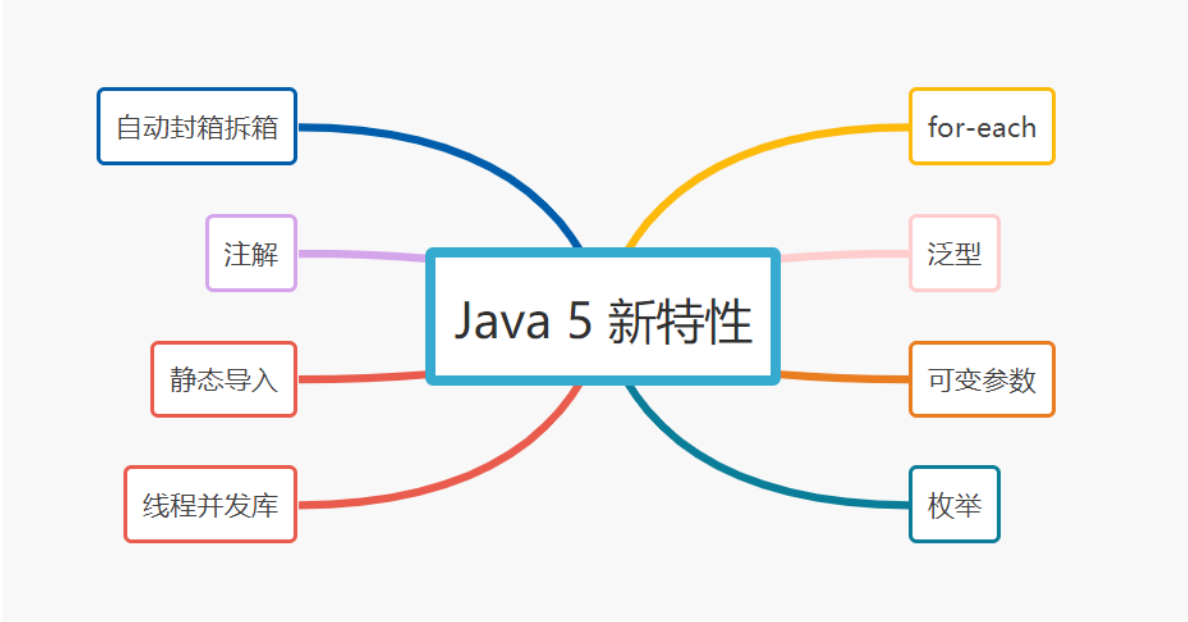
Java 5 新特性



1. 泛型

泛型本质是参数化类型，解决不确定具体对象类型的问题。

```
List<String> strList=new ArrayList<String>();
```

2. 增强循环（for-each）

for-each循环简化了集合的遍历。

```
String [] str = {"关注","公众号","捡田螺的小男孩"};
for (String temp:str) {
     System.out.println(temp);
}
```

3. 自动封箱拆箱

- 自动装箱: 就是将基本数据类型自动转换成对应的包装类。
- 自动拆箱：就是将包装类自动转换成对应的基本数据类型。

包装类型有：Integer,Double,Float,Long,Short,Character和Boolean

```
Integer i =666;  //自动装箱
int a= i;     //自动拆箱
```

4. 枚举

关键字enum可以将一组具名的值的有限集合创建为一种新的类型，而这些具名的值可以作为常规的程序组件使用，这就是枚举类型。

```
enum SeasonEnum {
    SPRING,SUMMER,FALL,WINTER;
}
```

5. 可变参数

我们在定义方法参数的时候不确定定义多少个，就可以定义为**「可变参数」**，它本质上是一个**「数组」**。

```
public static void main(String[] args) throws Exception {
    String [] str = {"关注","公众号","捡田螺的小男孩"};
    testVarargs(str);
    String str1 = "关注公众号，捡田螺的小男孩";
    testVarargs(str1);
}
//可变参数String... args
private static void testVarargs(String... args) {
    for (String arg : args) {
        System.out.println(arg);
    }
}
```

6. 注解

可以把注解理解为代码里的特殊标记，这些标记可以在编译，类加载，运行时被读取，并执行相应的处理。

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

7.静态导入

通过import static类，就可以使用类里的静态变量或方法。看一下例子哈~

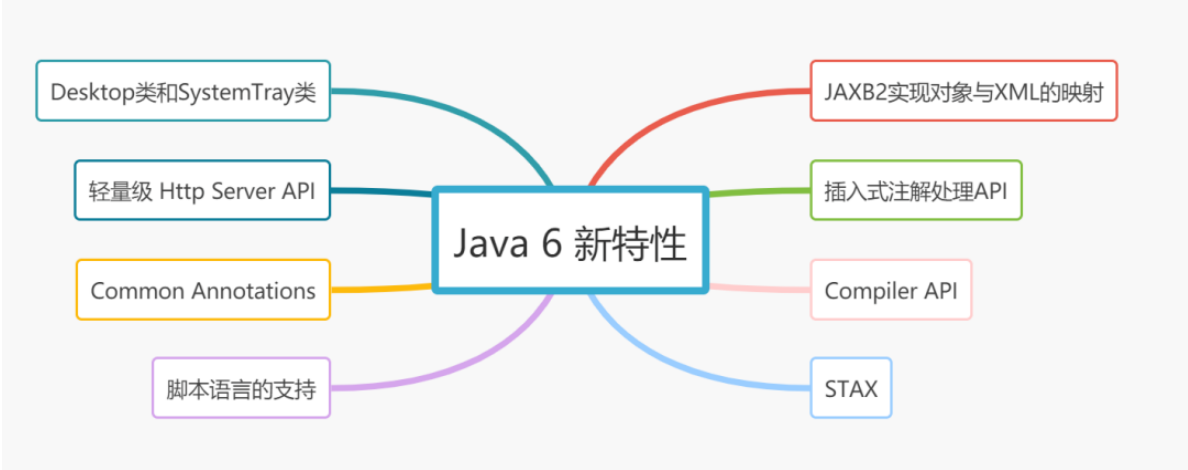
```
import static java.lang.System.out; //静态导入System类的静态变量out
public class Test {
    public static void main(String[] args) throws Exception {
        String str1 = "关注公众号，捡田螺的小男孩";
        System.out.println(str1); //常规写法
        out.println(str1);  //静态导入，可以直接使用out输出
    }
}
```

8. 线程并发库（JUC）

JDK5 丰富了线程处理功能，java.util.concurrent包提供了以下的类、接口：

> “
>
> - 线程池：ExecutorService接口
> - 线程护斥：Lock 类
> - 线程通信：Condition接口
> - 同步队列：ArrayBlockingQueue类
> - 同步集合：ConcurrentHashMap类
>
> ”

Java 6 新特性



1.Desktop类和SystemTray类

JDK 6在java.awt包下，新增了两个类：Desktop类和SystemTray类

> “
>
> - **「Desktop类」**：用来打开系统默认浏览器浏览指定的URL,打开系统默认邮件客户端发邮件等
> - **「SystemTray类」**:用来在系统托盘区创建一个托盘程序,如果在微软的Windows上，它被称为“任务栏”状态区域。
>
> ”

```
//获取Desktop实例
Desktop desktop = Desktop.getDesktop();
desktop.browse(URI.create("https://www.baidu.com"));
```

2. 使用JAXB2来实现对象与XML之间的映射

JAXB,即Java Architecture for XML Binding,可以实现对象与XML之间的映射，常用注解如下:

> “
>
> - @XmlRootElement：注解在类上面，对应xml的跟元素，使用name属性定义根节点的名称。
> - @XmlElement：指定一个字段或get/set方法映射到xml的节点，使用name属性定义这个根节点的名称。

> - @XmlAttribute：将JavaBean对象的属性映射为xml的属性,使用name属性为生成的xml属性指定别名。
> - @XmlAccessorType:定义映射这个类中的何种类型都需要映射到xml。
> - @XmlSchema: 将包映射到XML名称空间
>
> ”

「看个例子吧~」

```
public class JAXB2XmlTest {
    public static void main(String[] args) throws JAXBException, IOException {

        List<Singer> list = new ArrayList<>();
        list.add(new Singer("jay", 8));
        list.add(new Singer("eason", 10));
        SingerList singerList = new SingerList();
        singerList.setSingers(list);
        String str = JAXB2XmlTest.beanToXml(singerList, SingerList.class);
        String path = "C:\\jay.txt";
        BufferedWriter bfw = new BufferedWriter(new FileWriter(new File(path)));
        bfw.write(str);
        bfw.close();
    }
    private static String beanToXml(Object obj, Class<?> load) throws
JAXBException {
        JAXBContext context = JAXBContext.newInstance(load);
        Marshaller marshaller = context.createMarshaller();
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
        marshaller.setProperty(Marshaller.JAXB_ENCODING, "GBK");
        StringWriter writer = new StringWriter();
        marshaller.marshal(obj,writer);
        return writer.toString();
    }
}
public class Singer {
    private String name;
    private int age;
    public Singer(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @XmlAttribute(name="name")
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @XmlAttribute(name="age")
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
@XmlRootElement(name="list")
```

```
public class SingerList {
    private List<Singer> singers;

    @XmlElement(name="singer")
    public List<Singer> getSingers() {
        return singers;
    }
    public void setSingers(List<Singer> singers) {
        this.singers = singers;
    }
}
```

「运行效果：」

```
<?xml version="1.0" encoding="GBK" standalone="yes"?>
<list>
    <singer age="8" name="jay"/>
    <singer age="10" name="eason"/>
</list>
```

3.轻量级 Http Server API

JDK 6中提供了简单的Http Server API，可以构建嵌入式Http服务器,同时支持Http和Https协议。HttpServer会调用HttpHandler实现类的回调方法来处理客户端请求,这里用户只需实现HttpHandler接口就可以了。

```
/**
 * 根据Java提供的API实现Http服务器
 */
public class MyHttpServer {
    /**
     * @param args
     * @throws IOException
     */
    public static void main(String[] args) throws IOException {
        //创建HttpServer服务器
        HttpServer httpServer = HttpServer.create(new InetSocketAddress(8080),
10);
        //将 /jay请求交给MyHandler处理器处理
        httpServer.createContext("/", new MyHandler());
        httpServer.start();
    }
}
public class MyHandler implements HttpHandler {
    public void handle(HttpExchange httpExchange) throws IOException {
        //请求头
        Headers headers = httpExchange.getRequestHeaders();
        Set<Map.Entry<String, List<String>>> entries = headers.entrySet();
        StringBuffer response = new StringBuffer();
        for (Map.Entry<String, List<String>> entry : entries){
            response.append(entry.toString() + "\n");
        }
        //设置响应头属性及响应信息的长度
        httpExchange.sendResponseHeaders(200, response.length());
```

```
        //获得输出流
        OutputStream os = httpExchange.getResponseBody();
        os.write(response.toString().getBytes());
        os.close();
    }
}
```

4. 插入式注解处理API

> “
>
> JDK 6提供了插入式注解处理API，可以让我们定义的注解在编译期而不是运行期生效，从而可以在编译期修改字节码。lombok框架就是使用该特性来实现的，Lombok通过注解的方式，在编译时自动为属性生成构造器、getter/setter、equals、hashcode、toString等方法，大大简化了代码的开发。
>
> ”

5. STAX

STAX，是JDK6中一种处理XML文档的API。

```
public class STAXTest {
    public static void main(String[] args) throws Exception {
        XMLInputFactory xmlInputFactory = XMLInputFactory.newInstance();
        XMLEventReader xmlEventReader = xmlInputFactory.createXMLEventReader(new
FileInputStream("C:\\jay.xml"));
        XMLEvent event = null;
        StringBuffer stringBuffer = new StringBuffer();
        while (xmlEventReader.hasNext()) {
            event = xmlEventReader.nextEvent();
            stringBuffer.append(event.toString());
        }
        System.out.println("xml文档解析结果：");
        System.out.println(stringBuffer);
    }
}
```

「运行结果:」

```
xml文档解析结果：
<?xml version="1.0" encoding='GBK' standalone='yes'?><list>
    <singer name='jay' age='8'></singer>
    <singer name='eason' age='10'></singer>
</list>ENDDOCUMENT
```

6. Common Annotations

> “
>
> Common annotations原本是Java EE 5.0(JSR 244)规范的一部分，现在SUN把它的一部分放到了Java SE 6.0中。随着Annotation元数据功能加入到Java SE 5.0里面，很多Java 技术都会用Annotation部分代替XML文件来配置运行参数。
>
> ”

以下列举Common Annotations 1.0里面的几个Annotations:

- @Generated：用于标注生成的源代码
- @Resource： 用于标注所依赖的资源，容器据此注入外部资源依赖，有基于字段的注入和基于setter方法的注入两种方式 。
- @Resources：同时标注多个外部依赖，容器会把所有这些外部依赖注入
- @PostConstruct：标注当容器注入所有依赖之后运行的方法，用来进行依赖注入后的初始化工作，只有一个方法可以标注为PostConstruct 。
- @PreDestroy：当对象实例将要被从容器当中删掉之前，要执行的回调方法要标注为PreDestroy

7. Compiler API

javac编译器可以把.java的源文件编译为.class文件，JDK 6的新特性Compiler API(JSR 199)也可以动态编译Java源文件。

```
public class CompilerApiTest {
    public static void main(String[] args) throws Exception {
        JavaCompiler javaCompiler = ToolProvider.getSystemJavaCompiler();
        StandardJavaFileManager standardJavaFileManager =
javaCompiler.getStandardFileManager(null,null,null);
        Iterable<? extends JavaFileObject> javaFileObjects =
standardJavaFileManager.getJavaFileObjects("C:\\Singer.java");
        javaCompiler.getTask(null, standardJavaFileManager, null, null, null,
javaFileObjects).call();
        standardJavaFileManager.close();
    }
}
```

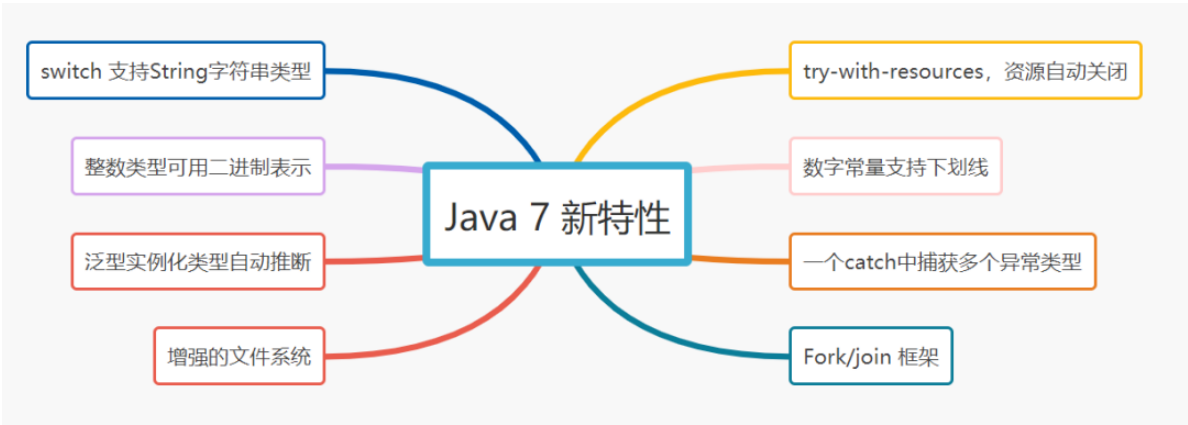
运行结果：会在C目录生成Singer.class文件

8. 对脚本语言的支持（如: ruby, groovy, javascript）

JDK6增加了对脚本语言的支持(JSR 223)，原理是将脚本语言编译成字节码，这样脚本语言也能享用Java平台的诸多优势，包括可移植性，安全等。JDK6实现包含了一个基于Mozilla Rhino的 脚本语言引擎,因此可以支持javascript，当然JDK也支持ruby等其他语言

```
public class JavaScriptTest {
    public static void main(String[] args) throws Exception {
        ScriptEngineManager factory = new ScriptEngineManager();
        ScriptEngine engine = factory.getEngineByName("JavaScript");
        String script;
        try {
            script = "print('Hello')";
            engine.eval(script);// 执行脚本
        }catch (Exception e) {
            e.printStackTrace();
        }
    }
}
//output
Hello
```

Java 7 新特性



1.switch 支持String字符串类型。

```
String singer = "jay";
switch (singer) {
       case "jay" :
            System.out.println("周杰伦");
             break;
       case "eason" :
            System.out.println("陈奕迅");
            break ;
       default :
            System.out.println("其他");
            break ;
   }
```

2.try-with-resources，资源自动关闭

JDK 7 之前:

```
BufferedReader br = new BufferedReader(new FileReader("d:七里香.txt"));
try {
   return br.readLine();
} finally {
   br.close();
}
```

JDK 7 之后:

```
/*
 * 声明在try括号中的对象称为资源，在方法执行完毕后会被自动关闭
 */
try (BufferedReader br = new BufferedReader(new FileReader("d:七里香.txt")) {
   return br.readLine();
}
```

3. 整数类型如（byte，short，int，long）能够用二进制来表示

```
//0b或者0B表示二进制
int a = 0b010;
int b = 0B010;
```

4. 数字常量支持下划线

```
int a = 11_11;//a的值为1111，下划线不影响实际值，提升可读性
```

5. 泛型实例化类型自动推断,即”<>”

JDK 7 之前:

```
Map<String, List<String>> map = new HashMap<String, List<String>>();
```

JDK 7之后:

```
//不须声明类型,自动根据前面<>推断其类型
Map<String, List<String>> map = new HashMap<>();
```

6.一个catch中捕获多个异常类型，用（|）分隔开

JDK 7之前

```
try{
   //do something
} catch (FirstException e) {
     logger.error(e);
} catch (SecondException e) {
     logger.error(ex);
}
```

JDk 7之后

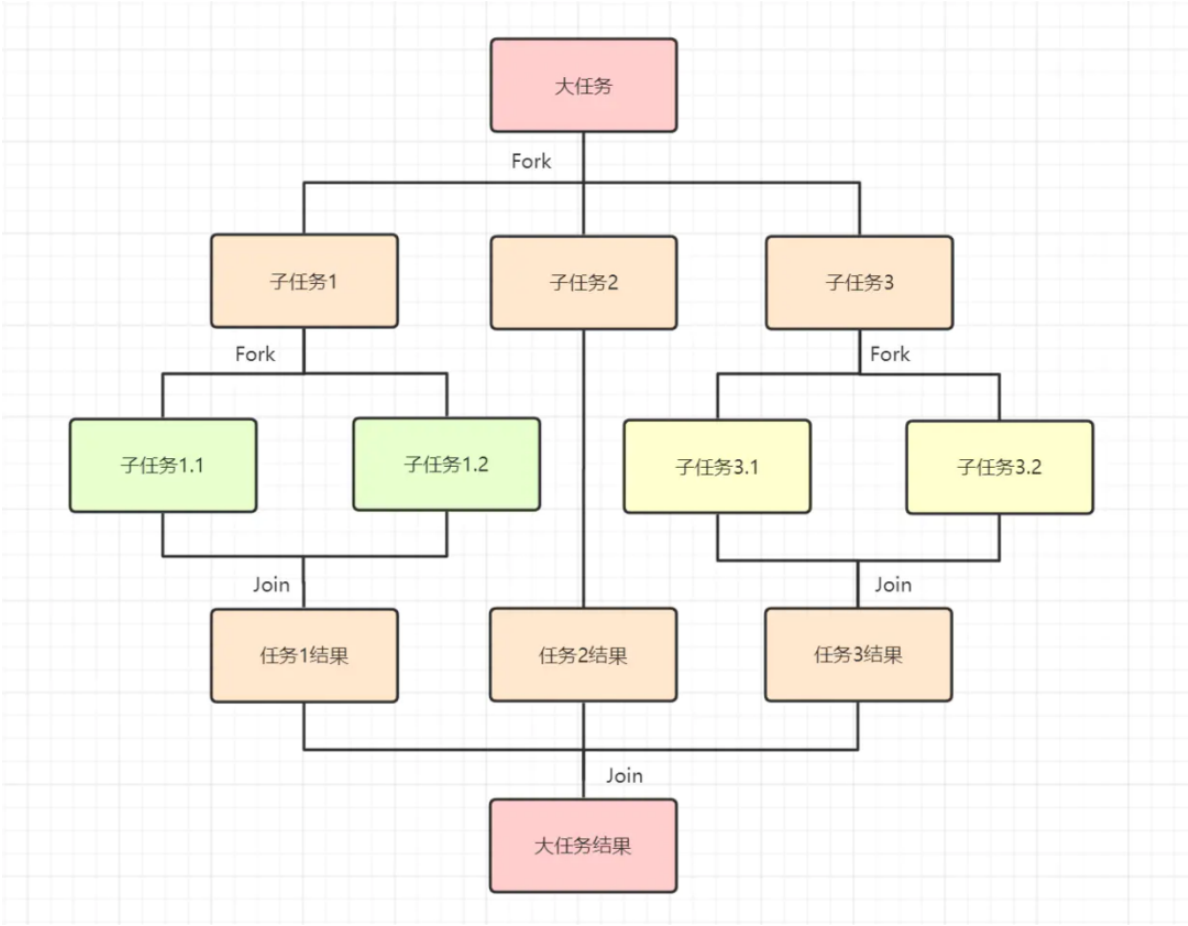
```
try{
   //do something
} catch (FirstException | SecondException e) {
     logger.error(e);
}
```

7. 增强的文件系统

Java7 提供了全新的NIO2.0 API，方便文件管理的编码。如，可以在java.nio.file包下使用Path、Paths、Files、WatchService等常用类型。

```
Path path = Paths.get("C:\\jay\\七里香.txt"); //创建Path对象
byte[] bytes= Files.readAllBytes(path);  //读取文件
System.out.println(path.getFileName()); //获取当前文件名称
System.out.println(path.toAbsolutePath()); // 获取文件绝对路径
System.out.println(new String(bytes, "utf-8"));
```

8. Fork/join 框架



Java7提供的一个用于并行执行任务的框架，是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。

Fork/join计算1-1000累加值：

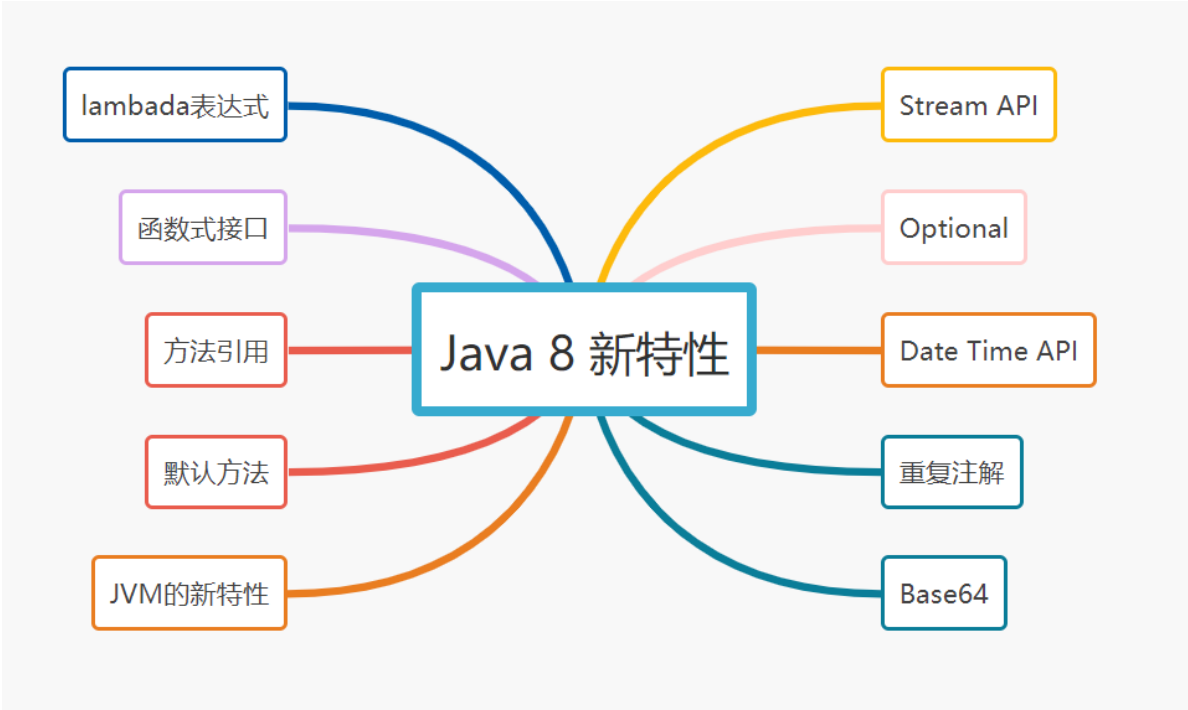
```
public class ForkJoinPoolTest {
    private static final Integer DURATION_VALUE = 100;
    static class ForkJoinSubTask extends RecursiveTask<Integer>{
        // 子任务开始计算的值
        private Integer startValue;
        // 子任务结束计算的值
        private Integer endValue;
        private ForkJoinSubTask(Integer startValue , Integer endValue) {
            this.startValue = startValue;
            this.endValue = endValue;
        }
        @Override
        protected Integer compute() {
            //小于一定值DURATION,才开始计算
            if(endValue - startValue < DURATION_VALUE) {
                System.out.println("执行子任务计算：开始值 = " + startValue + ";结束
值 = " + endValue);
```

```
                Integer totalValue = 0;
                for (int index = this.startValue; index <= this.endValue;
index++) {
                    totalValue += index;
                }
                return totalValue;
            } else {
                // 将任务拆分，拆分成两个任务
                ForkJoinSubTask subTask1 = new ForkJoinSubTask(startValue,
(startValue + endValue) / 2);
                subTask1.fork();
                ForkJoinSubTask subTask2 = new ForkJoinSubTask((startValue +
endValue) / 2 + 1 , endValue);
                subTask2.fork();
                return subTask1.join() + subTask2.join();
            }
        }
    }
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        // Fork/Join框架的线程池
        ForkJoinPool pool = new ForkJoinPool();
        ForkJoinTask<Integer> taskFuture =  pool.submit(new
ForkJoinSubTask(1,1000));
        Integer result = taskFuture.get();
        System.out.println("累加结果是:" + result);
    }
}
```

运行结果：

```
...
执行子任务计算：开始值 = 189;结束值 = 250
执行子任务计算：开始值 = 251;结束值 = 313
执行子任务计算：开始值 = 314;结束值 = 375
执行子任务计算：开始值 = 376;结束值 = 438
执行子任务计算：开始值 = 439;结束值 = 500
执行子任务计算：开始值 = 501;结束值 = 563
执行子任务计算：开始值 = 564;结束值 = 625
执行子任务计算：开始值 = 626;结束值 = 688
执行子任务计算：开始值 = 689;结束值 = 750
执行子任务计算：开始值 = 751;结束值 = 813
执行子任务计算：开始值 = 814;结束值 = 875
执行子任务计算：开始值 = 876;结束值 = 938
执行子任务计算：开始值 = 939;结束值 = 1000
累加结果是:500500
```

Java 8 新特性



1.lambada表达式

Lambda 允许把函数作为一个方法的参数，传递到方法中

语法格式：

```
(parameters) -> expression 或 (parameters) ->{ statements; }
```

代码示例：

```
Arrays.asList("jay", "Eason", "SHE").forEach(
       ( String singer ) -> System.out.print( singer + ",") );
```

2. 函数式接口

Lambda的设计者为了让现有的功能与Lambda表达式很好兼容，设计出函数式接口。

- 函数式接口是指只有一个函数的接口，可以隐式转换为lambada表达式。
- Java 8 提供了注解@FunctionalInterface，显示声明一个函数式接口。
- java.lang.Runnable和java.util.concurrent.Callable是函数式接口的例子~

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
```

3. 方法引用

方法引用提供了非常有用的语法，可以直接引用已有Java类或对象（实例）的方法或构造器。它与Lambda表达式配合使用，可以减少冗余代码，使代码更加简洁。

```
//利用函数式接口Consumer的accept方法实现打印，Lambda表达式如下
Consumer<String> consumer = x -> System.out.println(x);
consumer.accept("jay");
//引用PrintStream类（也就是System.out的类型）的println方法，这就是方法引用
consumer = System.out::println;
consumer.accept("关注公众号捡田螺的小男孩");
```

4. 默认方法

默认方法就是一个在接口里面有了一个实现的方法。它允许将新方法添加到接口，但不强制实现了该接口的类必须实现新的方法。

```
public interface ISingerService {
    // 默认方法
    default void sing(){
        System.out.println("唱歌");
    }
    void writeSong();
}
//JaySingerServiceImpl 不用强制实现ISingerService的默认sing()方法
public class JaySingerServiceImpl implements ISingerService {
    @Override
    public void writeSong() {
        System.out.println("写了一首七里香");
    }
}
```

5.Stream API

Stream API，支持对元素流进行函数式操作，它集成在Collections API 中，可以对集合进行批量操作。常用API：

- filter 筛选
- map流映射
- reduce 将流中的元素组合起来
- collect 返回集合
- sorted 排序
- flatMap 流转换
- limit返回指定流个数
- distinct去除重复元素

```
public class Singer {
    private String name;
    private Integer songNum;
    private Integer age;
    ...
}
List<Singer> singerList = new ArrayList<Singer>();
singerList.add(new Singer("jay", 11, 36));
singerList.add(new Singer("eason", 8, 31));
singerList.add(new Singer("JJ", 6, 29));
List<String> singerNameList = singerList.stream()
                .filter(singer -> singer.getAge() > 30)  //筛选年龄大于30
```

```
                .sorted(Comparator.comparing(Singer::getSongNum))  //根据歌曲数量排
序
                .map(Singer::getName)  //提取歌手名字
                .collect(Collectors.toList()); //转换为List
```

6. Optional

Java 8引入Optional类，用来解决NullPointerException。Optional代替if...else解决空指针问题，使代码更加简洁。

if...else 判空

```
Singer singer = getSingerById("666");
if (singer != null) {
    String name  = singer.getName();
    System.out.println(name);
}
```

Optional的判空

```
Optional<Singer> singer = Optional.ofNullable(getSingerById("666"));
singer.ifPresent(s -> System.out.println(s.getName()));
```

7. Date Time API

JDK 8之前的日期API处理存在非线程安全、时区处理麻烦等问题。Java 8 在 java.time包下提供了新的日期API，简化了日期的处理~

```
LocalDate today = LocalDate.now();
int year = today.getYear();
System.out.println("今年是" + year);
//是否闰年
System.out.println("今年是不是闰年:" + today.isLeapYear());
LocalDateTime todayTime = LocalDateTime.now();
System.out.println("当前时间" + todayTime);
//时区指定
System.out.println("美国时间:" +
ZonedDateTime.of(todayTime,ZoneId.of("America/Los_Angeles")));

LocalDate specailDate = LocalDate.of(2020, 6, 20);
LocalDate expectDate = specailDate.plus(100, ChronoUnit.DAYS);
System.out.println("比较特别的一天" + specailDate);
System.out.println("特殊日期的100天" + expectDate);
```

8. 重复注解

重复注解，即一个注解可以在一个类、属性或者方法上同时使用多次；用@Repeatable定义重复注解

```java
@Repeatable(ScheduleTimes.class)
public @interface ScheduleTime {
    String value();
}
public @interface ScheduleTimes {
    ScheduleTime[] value();
}
public class ScheduleTimeTask {
    @ScheduleTime("10")
    @ScheduleTime("12")
    public void doSomething() { }
}
```

9. Base64

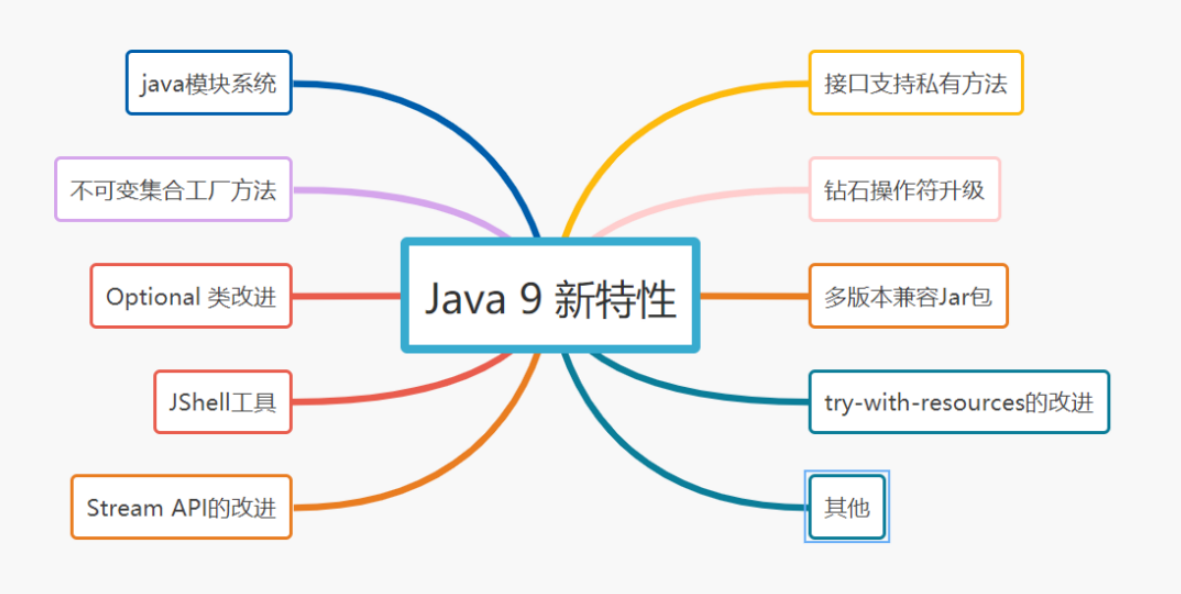
Java 8把Base64编码的支持加入到官方库中~

```java
String str = "公众号:捡田螺的小男孩";
String encoded = Base64.getEncoder().encodeToString(str.getBytes(
StandardCharsets.UTF_8));
String decoded = new String(Base64.getDecoder().decode(encoded),
StandardCharsets.UTF_8);
```

10. JVM的新特性

使用元空间Metaspace代替持久代（PermGen space），JVM参数使用-XX:MetaSpaceSize和-XX:MaxMetaspaceSize设置大小。

Java 9 新特性



1. java模块系统

什么是模块化?

> “

> 一个大型系统，比如一个商城网站，它会包含很多模块的，如：订单模块，用户信息模块，商品信息模块，广告位模块等等。各个模块之间会相互调用。如果每个模块单独运行都会带动其他所有模块，性能非常低效。但是，如果某一模块运行时，只会启动它所依赖的模块，性能大大提升。这就是JDK 9模块化的思想。
>
> ”

什么是JDK 9模块化?

> “
>
> Java 平台模块系统，即Project Jigsaw，把模块化开发实践引入到了Java平台中。在引入了模块系统之后，JDK 被重新组织成94个模块。Java 应用可以通过新增的jlink 工具，创建出只包含所依赖的JDK模块的自定义运行时镜像。这样可以极大的减少Java运行时环境的大小。
>
> ”

Java 9 模块的重要特征:

> “
>
> - 在其工件（artifact）的根目录中包含了一个描述模块的 module-info.class 文 件。
> - 工件的格式可以是传统的 JAR 文件或是 Java 9 新增的 JMOD 文件。
> - 这个文件由根目录中的源代码文件 module-info.java 编译而来。
> - 该模块声明文件可以描述模块的不同特征。
>
> ”

在 module-info.java 文件中，我们可以用新的关键词module来声明一个模块，如下所示。下面给出了一个模块com.mycompany.mymodule的最基本的模块声明

```
module com.jay.sample {   //关键词module来声明一个模块
    exports com.jay.sample; //使用 exports可以声明模块对其他模块所导出的包。
    requires com.jay.common; //使用requires可以声明模块对其他模块的依赖关系。
}
```

2. 不可变集合工厂方法

为了创建不可变集合，JDK9之前酱紫的:

```
List<String> stringList = new ArrayList<>();
stringList.add("关注公众号:");
stringList.add("捡田螺的小男孩");
List<String> unmodifiableList = Collections.unmodifiableList(stringList);
```

JDK 9 提供了List.of()、Set.of()、Map.of()和Map.ofEntries()等工厂方法来创建不可变集合:

```
List<String> unmodifiableList = List.of("关注公众号:","捡田螺的小男孩");
```

3. 接口支持私有方法

JDK 8支持在接口实现默认方法和静态方法，但是不能在接口中创建私有方法，为了避免了代码冗余和提高阅读性，JDK 9在接口中支持私有方法。

```
public interface IPrivateInterfaceTest {
    //JDK 7 之前
    String a = "jay";
    void method7();
    //JDK 8
    default void methodDefault8(){
        System.out.println("JDK 8新特性默认方法");
    }
    static void methodStatic8() {
        System.out.println("JDk 8新特性静态方法");
    }

    //Java 9 接口支持私有方法
    private void method9(){}
}
```

4. 钻石操作符升级

- 钻石操作符是在 java 7 中引入的，可以让代码更易读，但它不能用于匿名的内部类。
- 在 java 9 中， 它可以与匿名的内部类一起使用，从而提高代码的可读性。

```
//JDK 5,6
Map<String, String> map56 = new HashMap<String,String>();
//JDk 7,8
Map<String, String> map78 = new HashMap<>();
//JDK 9 结合匿名内部类的实现
Map<String, String> map9 = new HashMap<>(){};
```

5. Optional 类改进

java 9 中，java.util.Optional 添加了很多新的有用方法，如：

- stream()
- ifPresentOrElse()
- or()

ifPresentOrElse 方法的改进就是有了 else，接受两个参数 Consumer 和 Runnable。

```
import java.util.Optional;

public class OptionalTest {
   public static void main(String[] args) {
      Optional<Integer> optional = Optional.of(1);

      optional.ifPresentOrElse( x -> System.out.println("Value: " + x),() ->
         System.out.println("Not Present."));

      optional = Optional.empty();
```

```
        optional.ifPresentOrElse( x -> System.out.println("Value: " + x),() ->
            System.out.println("Not Present."));
    }
}
```

6. 多版本兼容Jar包

> “
>
> 很多公司使用的JDK都是老版本的，JDK6、JDk5 ，甚至JDk4的，不是他们不想升级JDk版本，而是担心兼容性问题。JDK 9的一个新特性，多版本兼容Jar包解决了这个问题。举个例子：假设你一直用的是小米8，已经非常习惯它的运行流程了，突然出来小米9，即使小米9很多新功能引人入胜，但是有些人不会轻易买小米9，因为已经已经习惯小米8的流程。同理，为什么很多公司不升级JDK，就是在此。但是呢，JDK 9的这个功能很强大，它可以让你的版本升级到JDK 9，但是还是老版本的运行流程，即在老的运行流程继承新的功能~
>
> ”

7. JShell工具

jShell工具相当于cmd工具，然后呢，你可以像在cmd工具操作一样，直接在上面运行Java方法，Java语句等~

```
jshell> System.out.println("关注公众号：捡田螺的小男孩");
关注公众号：捡田螺的小男孩
```

8. try-with-resources的改进

JDK 9对try-with-resources异常处理机制进行了升级~

```
//JDK 7,8
try (BufferedReader br = new BufferedReader(new FileReader("d:七里香.txt")) {
   br.readLine();
}catch(IOException e){
  log.error("IO 异常，e:{}",e);
}
//JDk 9
BufferedReader br = new BufferedReader(new FileReader("d:七里香.txt")
try(br){
  br.readLine();
}catch(IOException e){
  log.error("IO 异常，e:{}",e);
}
```

9. Stream API的改进

JDK 9 为Stream API引入以下这些方法，丰富了流处理操作：

- takeWhile（）
- dropWhile（）
- iterate
- ofNullable

「takeWhile」

使用一个断言（Predicate 接口）作为参数，返回给定Stream的子集直到断言语句第一次返回 false

```
// 语法格式
default Stream<T> takeWhile(Predicate<? super T> predicate)
//代码示例
Stream.of(1,2,3).takeWhile(s-> x<2)
          .forEach(System.out::println);
 //输出
 1
```

「dropWhile」

与 takeWhile（）作用相反，使用一个断言（Predicate 接口）作为参数，直到断言语句第一次返回 true，返回给定Stream的子集

```
//语法
default Stream<T> dropWhile(Predicate<? super T> predicate)
//代码示例
Stream.of(1,2,3).dropWhile(s-> x<2)
          .forEach(System.out::println);
//输出
2
3
```

「iterate」

iterate() 方法能够返回以seed（第一个参数）开头，匹配 Predicate（第二个参数）直到返回false，并使用第三个参数生成下一个元素的元素流。

```
//语法
static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext,
UnaryOperator<T> next)
//代码示例
IntStream.iterate(2, x -> x < 10, x -> x*x).forEach(System.out::println);
//输出
2
4
```

「ofNullable」

如果指定元素为非null，则获取一个元素并生成单个元素流，元素为null则返回一个空Stream。

```
//语法
static <T> Stream<T> ofNullable(T t)
//代码示例
Stream<Integer> s1= Stream.ofNullable(100);
s1.forEach(System.out::println)
Stream<Integer> s2 = Stream.ofNullable(null);
s2.forEach(System.out::println)
//输出
100
```

10.其他

> “
>
> - HTTP 2客户端 (支持 WebSocket和 HTTP2 流以及服务器推送)
> - 进程API（控制和管理操作系统进程）
> - String底层存储结构更改(char[]替换为byte[])
> - 标识符添加限制(String _ ="hello"不能用)
> - 响应式流 API (支持Java 9中的响应式编程)
>
> ”

Java 10 新特性

1.局部变量类型推断

JDK 10增加了局部变量类型推断（Local-Variable Type Inference）功能，让 Java 可以像Js里的var一样可以自动推断数据类型。Java中的var是一个保留类型名称，而不是关键字。

JDK 10之前

```
List<String> list = new ArrayList<String>();
Stream<Integer> stream = Stream.of(1, 2, 3);
```

JDK 10 之后

```
var list = new ArrayList<String>(); // ArrayList<String>
var stream =  Stream.of(1, 2, 3);
```

var 变量类型推断的使用也有局限性，仅**「局限」**于以下场景：

- 具有初始化器的局部变量
- 增强型for循环中的索引变量
- 传统for循环中声明的局部变量

而**「不能用于」**

- 推断方法的参数类型
- 构造函数参数类型推断
- 推断方法返回类型
- 字段类型推断
- 捕获表达式

2. 不可变集合的改进

JDK 10中，List，Set，Map 提供了一个新的静态方法copyOf(Collection<? extends E> coll)，它返回Collection集合一个不可修改的副本。

JDK 源码：

```
static <E> List<E> copyOf(Collection<? extends E> coll) {
    return ImmutableCollections.listCopy(coll);
}
```

使用实例：

```
var oldList = new ArrayList<String>();
oldList.add("欢迎关注公众号：");
oldList.add("捡田螺的小男孩");
var copyList = List.copyOf(oldList);
oldList.add("在看、转载、点赞三连");
copyList.add("双击666");  //UnsupportedOperationException异常
```

3. 并行全垃圾回收器 G1

> “
>
> JDK 9引入 G1 作为默认垃圾收集器，执行GC 时采用的是基于单线程标记扫描压缩算法（mark-sweep-compact）。为了最大限度地减少 Full GC 造成的应用停顿的影响，Java 10 中将为 G1 引入多线程并行 GC，同时会使用与年轻代回收和混合回收相同的并行工作线程数量，从而减少了 Full GC 的发生，以带来更好的性能提升、更大的吞吐量。
>
> ”

4. 线程本地握手

Java 10 中线程管控引入JVM安全点的概念，将允许在不运行全局JVM安全点的情况下实现线程回调，由线程本身或者JVM线程来执行，同时保持线程处于阻塞状态，这将会很方便使得停止单个线程或不停止线程成为可能。

5. Optional新增orElseThrow()方法

Optional、OptionalDouble等类新增一个方法orElseThrow()，在没有值时抛出异常

6. 其他新特性

- 基于 Java 的 实验性 JIT 编译器
- 类数据共享
- Unicode 语言标签扩展
- 根证书
- 基于时间（Time-Based）的版本控制模型

Java 11 新特性

1.字符串操作

String类是Java最常用的类，JDK 11增加了一系列好用的字符串处理方法

- isBlank() 判空。
- strip() 去除首尾空格
- stripLeading() 去除字符串首部空格
- stripTrailing() 去除字符串尾部空格
- lines() 分割获取字符串流。
- repeat() 复制字符串

```
// 判断字符串是否为空白
"  ".isBlank();    // true
// 去除首尾空格
" jay ".strip();  // "jay"
// 去除首部空格
" jay ".stripLeading();   // "jay "
去除字符串尾部空格
" jay ".stripLeading();   // " jay"
// 行数统计
"a\nb\nc".lines().count();    // 3
// 复制字符串
"jay".repeat(3);   // "jayjayjay"
```

2.用于 Lambda 参数的局部变量语法

局部变量类型推断是Java 10引入的新特性，但是不能在Lambda 表达式中使用。Java 11再次创新，它允许开发者在 Lambda 表达式中使用 var 进行参数声明。

```
var map = new HashMap<String, Object>();
map.put("公众号", "捡田螺的小男孩");
map.forEach((var k, var v) -> {
    System.out.println(k + ": " + v);
});
```

3.标准化HTTP Client

Java 9 引入Http Client API,Java 10对它更新，Java 11 对它进行标准化。这几个版本后，Http Client几乎被完全重写，支持HTTP/1.1和HTTP/2 ，也支持 websockets。

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create("https://github.com/whx123/JavaHome"))
            .GET()
            .build();
// 同步
HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
// 异步
client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

4. 单个命令编译运行源代码

Java 11增强了Java 启动器，使之能够运行单一文件的Java 源代码。

- Java 11之前,要运行一个 Java 源代码必须先编译，再运行

```
// 编译
javac Jay.java
// 运行
java Jay
```

- Java 11之后,只要一个java命令就搞定

```
java Jay.java
```

5. ZGC：可伸缩低延迟垃圾收集器

ZGC ，即 Z Garbage Collector（垃圾收集器或垃圾回收器）。它是一个可伸缩的、低延迟的垃圾收集器。 ZGC 主要为了满足如下目标进行设计：

- GC 停顿时间不超过 10ms
- 既能处理几百 MB 的小堆，也能处理几个 TB 的大堆
- 应用吞吐能力不会下降超过 15%（与 G1 回收算法相比）
- 方便在此基础上引入新的 GC 特性和利用 colord
- 针以及 Load barriers 优化奠定基础
- 当前只支持 Linux/x64 位平台

6.其他一些特性

- 添加 Epsilon 垃圾收集器。
- 支持 TLS 1.3 协议
- 飞行记录器分析工具
- 动态类文件常量
- 低开销的 Heap Profiling

Java 12 新特性

1. Switch 表达式扩展（预览功能）

传统的switch语句，容易漏写break而出错，同时写法并不简洁优雅。

Java 12之前

```
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        System.out.println(6);
        break;
    case TUESDAY:
        System.out.println(7);
        break;
    case THURSDAY:
    case SATURDAY:
        System.out.println(8);
        break;
    case WEDNESDAY:
        System.out.println(9);
        break;
}
```

JDk 12 之后，Switch表达式得到增强，能接受语句和表达式。

```
switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> System.out.println(6);
    case TUESDAY                -> System.out.println(7);
    case THURSDAY, SATURDAY     -> System.out.println(8);
    case WEDNESDAY              -> System.out.println(9);
}
```

2. 紧凑的数据格式

JDK 12 新增了NumberFormat对复杂数字的格式化

```
NumberFormat numberFormat = NumberFormat.getCompactNumberInstance(Locale.CHINA,
NumberFormat.Style.SHORT);
System.out.println(numberFormat.format(100000));
//output
10万
```

3. 字符串支持transform、indent操作

- transform 字符串转换，可以配合函数式接口Function一起使用

```
List<String> list1 = List.of("jay", " 捡田螺的小男孩");
List<String> list2 = new ArrayList<>();
list1.forEach(element ->
            list2.add(element.transform(String::strip)
                    .transform((e) -> "Hello," + e))
    );
list2.forEach(System.out::println);
//输出
Hello,jay
Hello,捡田螺的小男孩
```

- indent 缩进，每行开头增加空格space和移除空格

```
String result = "Java\n Python\nC".indent(3);
System.out.println(result);
//输出
   Java
    Python
   C
```

4. Files.mismatch(Path, Path)

Java 12 新增了mismatch方法，此方法返回第一个不匹配的位置，如果没有不匹配，则返回 -1L。

```
public static long mismatch(Path path, Path path2) throws IOException;
```

代码示例：

```
Path file1 = Paths.get("c:\\jay.txt");
Path file2 = Paths.get("c: \\捡田螺的小男孩.txt");
try {
 long fileMismatch = Files.mismatch(file1, file2);
 System.out.println(fileMismatch);
} catch (IOException e) {
 e.printStackTrace();
}
```

5. Teeing Collector

Teeing Collector 是 Streams API 中引入的新的收集器实用程序，它的作用是 merge 两个 collector 的结果,API格式如下：

```
public static <T, R1, R2, R>
    Collector<T, ?, R> teeing(Collector<? super T, ?, R1> downstream1,
    Collector<? super T, ?, R2> downstream2,
    BiFunction<? super R1, ? super R2, R> merger)
```

直接看代码例子吧，如下为求学生的平均分和总分的例子

```
  List<Student> studentList= Arrays.asList(
                new Student("jay", 90),
                new Student("捡田螺的小男孩", 100),
                new Student("捡表情的小男孩", 80)
        );
        String teeingResult=studentList.stream().collect(
                Collectors.teeing(
                        Collectors.averagingInt(Student::getScore),
                        Collectors.summingInt(Student::getScore),
                        (s1,s2)-> s1+ ":"+ s2
                )
        );
 System.out.println(teeingResult); //90:270
```

6.其他特性

- 支持unicode 11（684个新字符、11个新blocks、7个新脚本）
- JVM 常量 API （主要在新的java.lang.invoke.constant包中定义了一系列基于值的符号引用类型，能够描述每种可加载常量。）
- Shenandoah GC（低暂停时间垃圾收集器）
- G1 收集器提升 （可中止的混合收集集合、及时返回未使用的已分配内存）
- 默认CDS档案
- JMH 基准测试

Java 13 新特性

Switch 表达式扩展（引入 yield 关键字）

传统的switch：

```
private static String getText(int number) {
    String result = "";
    switch (number) {
        case 1, 2:
        result = "one or two";
        break;
        case 3:
        result = "three";
        break;
        case 4, 5, 6:
        result = "four or five or six";
        break;
        default:
        result = "unknown";
        break;
```

Java 13之后，value break 语句不再被编译，而是用 yield 来进行值返回

```
private static String getText(int number) {
    return switch (number) {
        case 1, 2:
            yield "one or two";
        case 3:
            yield "three";
        case 4, 5, 6:
            yield "four or five or six";
        default:
            yield "unknown";
    };
}
```

2.文本块升级

Java 13之前，字符串不能够多行使用，需要通过换行转义或者换行连接符等等，反正就是好麻烦、好难维护。

```
String html = "<html>\n" +
              "    <body>\n" +
              "        <p>Hello, 捡田螺的小男孩</p>\n" +
              "    </body>\n" +
              "</html>\n";
```

Java 13之后，清爽多了~

```
String html = """
                <html>
                    <body>
                        <p>Hello,  捡田螺的小男孩</p>
                    </body>
                </html>
                """;
```

3. SocketAPI 重构

- 传统的Java Socket API（java.net.ServerSocket 和 java.net.Socket）依赖于SocketImpl 的内部实现
- 在 Java 13之前，通过使用 PlainSocketImpl 作为 SocketImpl 的具体实现。
- Java 13 中的新底层实现，引入 NioSocketImpl 的实现用以替换 SocketImpl 的 PlainSocketImpl 实现，此实现与 NIO（新 I/O）实现共享相同的内部基础结构，并且与现有的缓冲区高速缓存机制集成在一起。

一个Socket简单例子：

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
public class SocketAPITest {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(8080)){
            boolean runFlag = true;
            while(runFlag){
                Socket clientSocket = serverSocket.accept();
                //搞事情
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

运行以上的实例，看下是否有以下关键词输出~

```
[class,load] sun.nio.ch.NioSocketImpl
```

4.FileSystems.newFileSystem新方法

FileSystems 类中添加了以下三种新方法，以便更容易地使用将文件内容视为文件系统的文件系统提供程序：

- 1、newFileSystem(Path)
- 2、newFileSystem(Path, Map<String, ?>)
- 3、newFileSystem(Path, Map<String, ?>, ClassLoader)

5. 增强 ZGC 释放未使用内存

- ZGC 是Java 11 中引入的最为瞩目的垃圾回收特性，是一种可伸缩、低延迟的垃圾收集器。但是实际使用中，它不能够主动将未使用的内存释放给操作系统。
- Java 13 中对 ZGC 的改进，包括释放未使用内存给操作系统、支持最大堆大小为 16TB、JVM参数-XX:SoftMaxHeapSize 来软限制堆大小

6.其他特性

- 动态 CDS 存档， 扩展了 Java 10 中引入的类数据共享功能， 使用CDS 存档变得更容易。
- 文本块的字符串类新方法，如formatted(Object...args)，stripIndent()等。

Java 14 新特性

1. instanceof模式匹配

instanceof 传统使用方式：

```
if (person instanceof Singer) {
    Singer singer = (Singer) person;
    singer.sing();
} else if (person instanceof Writer) {
    Writer writer = (Writer) person;
    writer.write();
}
```

Java 14 对 instanceof 进行模式匹配改进之后

```
if (person instanceof Singer singer) {
    singer.sing();
} else if (person instanceof Writer writer) {
   writer.write();
}
```

2.Record 类型（预览功能）

Java 14将Record 类型作为预览特性而引入，有点类似于Lombok 的@Data注解，看个例子吧：

```
public record Person(String name, int age) {
    public static String address;
    public String getName() {
        return name;
    }
}
```

反编译结果：

```
public final class Person extends java.lang.Record {
    private final java.lang.String name;
    private final java.lang.String age;
    public Person(java.lang.String name, java.lang.String age) { /* compiled
code */ }
    public java.lang.String getName() { /* compiled code */ }
    public java.lang.String toString() { /* compiled code */ }
    public final int hashCode() { /* compiled code */ }
    public final boolean equals(java.lang.Object o) { /* compiled code */ }
    public java.lang.String name() { /* compiled code */ }
    public java.lang.String age() { /* compiled code */ }
}
```

可以发现，当用 Record 来声明一个类时，该类将自动拥有下面特征：

- 构造方法
- hashCode() 方法
- euqals() 方法
- toString() 方法
- 类对象被final 关键字修饰，不能被继承。

3. Switch 表达式-标准化

switch 表达式在之前的 Java 12 和 Java 13 中都是处于预览阶段，终于在 Java 14 标准化，成为稳定版本。

- Java 12 为switch 表达式引入Lambda 语法
- Java 13 使用yield代替 break 关键字来返回表达式的返回值。

```
String result = switch (day) {
    case "M", "W", "F" -> "MWF";
    case "T", "TH", "S" -> "TTS";
    default -> {
        if (day.isEmpty()) {
            yield "Please insert a valid day.";
        } else {
            yield "Looks like a Sunday.";
        }
    }
};
System.out.println(result);
```

4. 改进 NullPointerExceptions提示信息

Java 14 之前：

```
String name = song.getSinger().getSingerName()

//堆栈信息
Exception in thread "main" java.lang.NullPointerException
    at NullPointerExample.main(NullPointerTest.java:6)
```

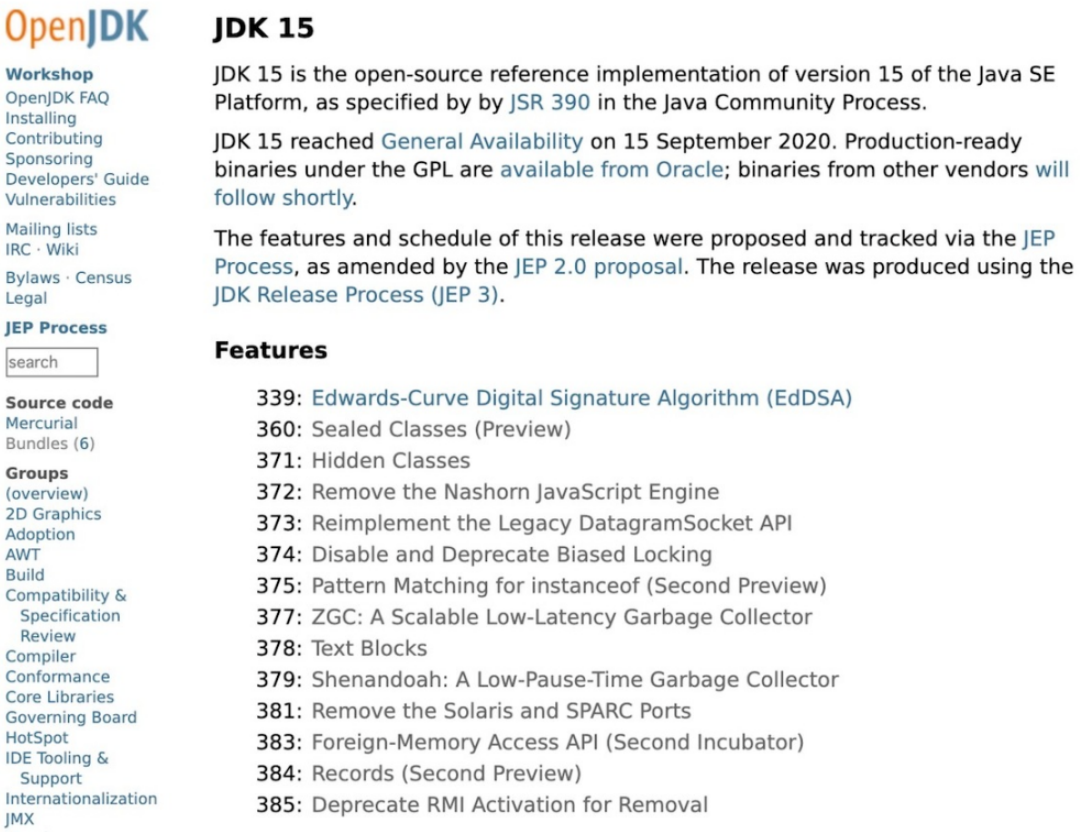
Java 14，通过引入JVM 参数-XX:+ShowCodeDetailsInExceptionMessages，可以在空指针异常中获取更为详细的调用信息。

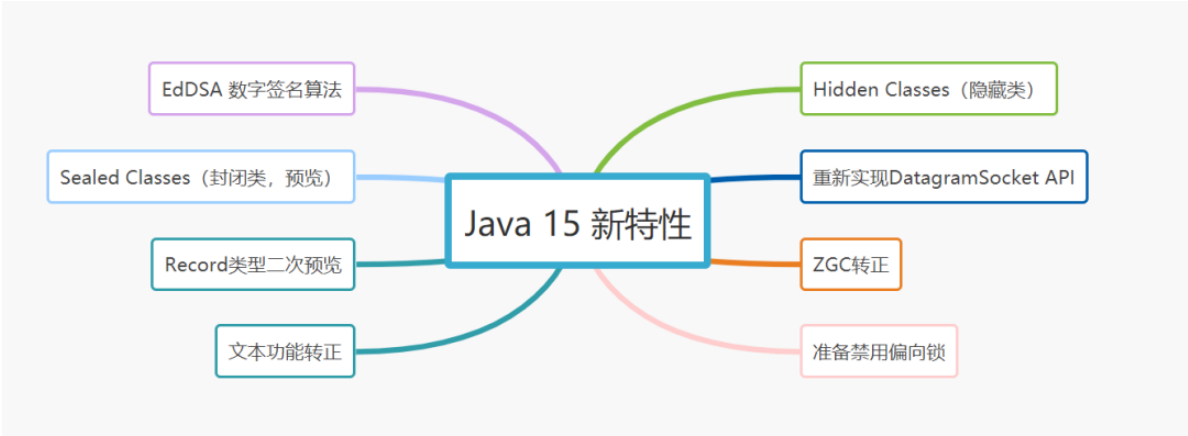
```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"Singer.getSingerName()"
because the return value of "rainRow.getSinger()" is null
    at NullPointerExample.main(NullPointerTest.java:6)
```

5. 其他特性

- G1 的 NUMA 可识别内存分配
- 删除 CMS 垃圾回收器
- GC 支持 MacOS 和 Windows 系统

Java 15 新特性





1.EdDSA 数字签名算法

- 使用 Edwards-Curve 数字签名算法（EdDSA）实现加密签名。
- 与其它签名方案相比，EdDSA 具有更高的安全性和性能。
- 得到许多其它加密库（如 OpenSSL、BoringSSL）的支持。

2.Sealed Classes（封闭类，预览）

封闭类，可以是封闭类、封闭接口，防止其他类或接口扩展或实现它们。

```
public abstract sealed class Singer
    permits Jay, Eason{
    ...
}
```

类Singer被sealed 修饰，是封闭类，只能被2个指定子类（Jay, Eason）继承。

3. Hidden Classes（隐藏类）

- 隐藏类天生为框架设计的。
- 隐藏类只能通过反射访问，不能直接被其他类的字节码。

4. Remove the Nashorn JavaScript Engine

- Nashorn太难维护了，移除 Nashorn JavaScript引擎成为一种必然
- 其实早在JDK 11 中就已经被标记为 deprecated 了。

5.Reimplement the Legacy DatagramSocket API（重新实现 DatagramSocket API）

- 重新实现老的DatagramSocket API
- 更改java.net.DatagramSocket 和 java.net.MulticastSocket 为更加简单、现代化的底层实现。

6.其他

- Disable and Deprecate Biased Locking（准备禁用偏向锁）
- instanceof 自动匹配模式（预览）
- ZGC，一个可伸缩、低延迟的垃圾回收器。（转正）
- Text Blocks，文本功能转正（JDK 13和14预览，14终于转正）
- Remove the Solaris and SPARC Ports（删除 Solaris 和 SPARC 端口）
- 外部存储器访问 API（允许Java 应用程序安全有效地访问 Java 堆之外的外部内存。）

- Record类型二次预览（在Java 14就预览过啦）

  来源： https://mp.weixin.qq.com/s/8_bwXonagarppt61FPdlDg