tomcat各个版本区别

# Tocmat

Apache Tomcat 9.x

**Apache Tomcat上9.x的**是发展的当前焦点，它建立在Tomcat 8.0.x和实现了目前草案**的Servlet 4.0**规范，也将执行 **JSP 2.4？**，**EL 3.1？**，**目前对WebSocket的1.2？**和**JASPIC 1.1** 规范工作的一次更新上这些规范为Java EE 8除此之外启动时，它包括以下显著改进：

* 添加对HTTP / 2的支持（需要APR /本地库）
* 添加对TLS虚拟主机的支持
* 添加了对使用JSSE连接器（NIO和NIO2）使用OpenSSL for TLS支持的支持。

Apache Tomcat 8.x

**的Apache Tomcat 8.x的**建立在Tomcat的7.0.x并实施 **的Servlet 3.1**，**JSP 2.3**，**EL 3.0** 和**WebSocket的1.1**规格。除此之外，还包括以下重大改进：

* 单个公共资源实现来替换早期版本中提供的多个资源扩展特性。

**的Apache Tomcat 8.5.x的**支持相同的Servlet，JSP，EL和WebSocket规范的版本的Apache Tomcat 8.0.x. 除此之外，它也实现了**JASPIC 1.1**规范。还有在许多领域显著变化引擎盖下，从而提高了性能，稳定性和总拥有成本。有关详细信息，请参阅Apache Tomcat 8.5更改日志。

Apache Tomcat 7.x

**的Apache Tomcat 7.x的**建立在Tomcat中6.0.x的改进和实现**的Servlet 3.0**， **JSP 2.2**，**EL 2.2**和**WebSocket的1.1**规格。除此之外，它还包括以下改进：

* Web应用程序内存泄漏检测和预防
* 提高了Manager和Host Manager应用程序的安全性
* 通用CSRF保护
* 支持直接在Web应用程序中包含外部内容
* 重构（连接器，生命周期）和大量的内部代码清理

Apache Tomcat 6.x

**的Apache Tomcat 6.x的**建立在Tomcat中的5.5.x的改进和实现**的Servlet 2.5**和 **JSP 2.1**规范。除此之外，它还包括以下改进：

* 内存使用优化
* 高级IO功能
* 重构聚类

Tomcat的6的用户应该知道，Tomcat的团队已经公布了 [的生命日期为Tomcat 6.x的结束](http://blog.csdn.net/i_love_t/article/details/tomcat-60-eol.html)。Tomcat 6.x的用户应该计划在Tomcat 6.x到达生命周期之前进行升级。

Apache Tomcat 5.x

**的Apache Tomcat 5.x的**是可以从档案下载。

**的Apache Tomcat 5.5.X**支持相同的Servlet和JSP规范版本的的Apache Tomcat 5.0.x中 还有在许多领域显著变化引擎盖下，从而提高了性能，稳定性和总拥有成本。有关详细信息，请参阅Apache Tomcat 5.5 Changelog。

**的Apache Tomcat 5.0.x版**在很多方面在Apache Tomcat 4.1的改进，其中包括：

* 性能优化和减少的垃圾收集
* 重构的应用程序部署器，具有可选的独立部署器，允许在Web应用程序投入生产之前进行验证和编译
* 使用JMX和管理器Web应用程序完成服务器监视
* 可扩展性和可靠性增强
* 改进了Taglibs的处理，包括高级池和标签插件
* 改进的平台集成，与本机Windows和Unix包装器
* 使用JMX嵌入
* 增强的安全管理器支持
* 集成会话聚类
* 扩展文档

Apache Tomcat 4.x

**的Apache Tomcat 4.x版**可以从档案下载。

**的Apache Tomcat 4.x的**实现了基于全新架构的新的servlet容器（称为卡特琳娜）。4.x的版本中实现**的Servlet 2.3**和**JSP 1.2** 规范。

**的Apache Tomcat 4.1.x的**是的Apache Tomcat 4.0.x的的重构，并含有显著增强功能，包括：

* 基于JMX的管理功能
* JSP和Struts的管理Web应用程序
* 新的Coyote连接器（HTTP / 1.1，AJP 1.3和JNI支持）
* 重写Jasper JSP页面编译器
* 性能和内存效率提高
* 增强了与开发工具集成的管理应用程序支持
* 自定义Ant任务可以直接从build.xml脚本与管理器应用程序交互

**的Apache Tomcat 4.0.x的**。Apache Tomcat 4.0.6是旧的生产质量版本。4.0 servlet容器（卡塔利娜）已经从地上爬起来的灵活性和性能开发。4.0版实现了Servlet 2.3和JSP 1.2规范的最终发布版本。根据规范的要求，Apache Tomcat 4.0还支持为Servlet 2.2和JSP 1.1规范构建的Web应用程序，无需更改。

**Apache Tomcat 4.x**

不推荐使用，只要有可能，我们都强烈建议用户升级到tomcat 5.x 以上的版本

Apache Tomcat 3.x

**Apache Tomcat上3.X**可以从档案下载。

* 版本**3.3**是当前生产质量放行了Servlet 2.2和JSP 1.1规范。Apache Tomcat 3.3是Apache Tomcat 3.x体系结构的最新延续; 它比3.2.4更先进，这是“老”的生产质量释放。
* 版本3.2.4是“旧的”生产质量版本，现在仅在维护模式。
* 版本3.1.1是旧版本。

所有**的Apache Tomcat 3.X**版本跟踪其遗产回到原来的Servlet和JSP实现，Sun公司捐赠给Apache软件基金会。该3.X版本都实现了**支持Servlet 2.2**和**JSP 1.1**规范。

**的Apache Tomcat 3.3.X**。版本3.3.2是当前的生产质量版本。它继续在3.2版本中开始的重构，并将其转化为其逻辑结论。3.3版本提供了更多的模块化设计，允许通过添加和删除控制servlet请求处理的模块来定制servlet容器。此版本还包含许多性能改进。

**的Apache Tomcat 3.2.X**。版本3.2自3.1以来增加了几个新功能; 主要的努力是重构内部以提高性能和稳定性。3.2.1版本，如3.1.1，是一个安全补丁。版本3.2.2修复了大量的错误和所有已知的规范合规性问题。版本3.2.3是一个安全更新，关闭一个严重的安全漏洞。版本3.2.4是一个小错误修复版本。所有Apache Tomcat 3.2.3之前版本的用户都应该尽快升级。除了修复关键安全相关的错误，Apache Tomcat 3.2.x分支上的开发已停止。

**的Apache Tomcat 3.1.X**。3.1版本包含对Apache Tomcat 3.0的几个改进，包括servlet重新加载，WAR文件支持和为IIS和Netscape Web服务器添加的连接器。最新的维护版本3.1.1包含了对安全问题的修复。Apache Tomcat 3.1.x没有进行积极的开发。Apache Tomcat 3.1的用户应该更新到3.1.1以关闭安全漏洞，强烈建议他们迁移到当前的生产版本Apache Tomcat 3.3。

**的Apache Tomcat 3.0.x的**。初始Apache Tomcat版本。

# JDK

JDK doc - <https://docs.oracle.com/javase/8/docs/technotes/tools/unix/toc.html>

1.5  
1.自动装箱与拆箱：  
2.枚举(常用来设计单例模式)  
3.静态导入  
4.可变参数  
5.内省  
  
1.6  
1.Web服务元数据  
2.脚本语言支持  
3.JTable的排序和过滤  
4.更简单,更强大的JAX-WS  
5.轻量级Http Server  
6.嵌入式数据库 Derby  
  
1.7  
1，switch中可以使用字串了  
2.运用List<String> tempList = new ArrayList<>(); 即泛型实例化类型自动推断  
3.语法上支持集合，而不一定是数组  
4.新增一些取环境信息的工具方法  
5.Boolean类型反转，空指针安全,参与位运算  
6.两个char间的equals   
7.安全的加减乘除   
8.map集合支持并发请求，且可以写成 Map map = {name:"xxx",age:18};  
  
1.8  
1. 允许在接口中有默认方法实现  
2. Lambda表达式  
3. 函数式接口  
4. 方法和构造函数引用  
5. Lambda的范围  
6. 内置函数式接口  
7. Streams  
8. Parallel Streams  
9. Map  
10. 时间日期API  
11. Annotations  
  
1.9  
1. Jigsaw 项目;模块化源码  
2. 简化进程API   
3. 轻量级 JSON API   
4. 钱和货币的API   
5. 改善锁争用机制  
6. 代码分段缓存  
7. 智能Java编译, 第二阶段  
8. HTTP 2.0客户端  
9. Kulla计划: Java的REPL实现

Java 9的14个新特性总结- http://geek.csdn.net/news/detail/196632

---------------------------------------------------------------------------

JDK1.5新特性：

1.自动装箱与拆箱：

Integer iObj = 3;

System.out.println(iObj + 12);  
   Integer i1 = 137(-128--127范围时，为true);  
   Integer i2 = 137(-128--127范围时，为true);

   System.out.println(i1 == i2); //false，但是括号中时却返回ture，原因是Integer采用的是享元模式

   Integer i3 = Integer.valueOf(213);  
   Integer i4 = Integer.valueOf(213);  
   System.out.println(i3==i4);//同上，另一种包装形式

2.枚举(常用来设计单例模式)

public class EnumTest {

public static void main(String[] args) {  
   WeekDay1 weekDay = WeekDay1.MON;  
   System.out.println(weekDay.nextDay());  
   WeekDay weekDay2 = WeekDay.FRI;  
   System.out.println(weekDay2);  
   System.out.println(weekDay2.name());  
   System.out.println(weekDay2.ordinal());   
   System.out.println(WeekDay.valueOf("SUN").toString());  
   System.out.println(WeekDay.values().length);  
   new Date(300){};  
}

public enum WeekDay{

   SUN(1),MON(),TUE,WED,THI,FRI,SAT;  
   private WeekDay(){System.out.println("first");}  
   private WeekDay(int day){System.out.println("second");}  
}  
  
public enum TrafficLamp{  
   RED(30){  
    public TrafficLamp nextLamp(){  
     return GREEN;  
    }  
   },  
   GREEN(45){  
    public TrafficLamp nextLamp(){  
     return YELLOW;  
    }     
   },  
   YELLOW(5){  
    public TrafficLamp nextLamp(){  
     return RED;  
    }     
   };  
   public abstract TrafficLamp nextLamp();  
   private int time;  
   private TrafficLamp(int time){this.time = time;}  
}  
}

3.静态导入

import static java.lang.Math.\*;

public class StaticImport {  
public static void main(String[] args){  
   int x = 1;  
   try {  
    x++;  
   } finally {  
    System.out.println("template");  
   }  
   System.out.println(x);  
    
    
   System.out.println(max(3, 6));  
   System.out.println(abs(3 - 6));  
    
}  
}  
4.可变参数

public class VarableParameter {

public static void main(String[] args) {  
  
   System.out.println(add(2,3));  
   System.out.println(add(2,3,5));    
}  
  
  
public static int add(int x,int... args){  
   int sum = x;  
  
    
   for(int arg : args){  
    sum += arg;  
   }  
   return sum;  
}

}  
5.内省

ReflectPoint pt1 = new ReflectPoint(3,5);

BeanInfo beanInfo = Introspector.getBeanInfo(pt1.getClass());  
   PropertyDescriptor[] pds = beanInfo.getPropertyDescriptors();  
   Object retVal = null;  
   for(PropertyDescriptor pd : pds){  
     Method methodGetX = pd.getReadMethod();  
     retVal = methodGetX.invoke(pt1);  
  
   }  
   jdk1.6新特性：

1.**Web服务元数据**

Java 里的Web服务元数据跟微软的方案基本没有语义上的区别,自从JDK5添加了元数据功能(Annotation)之后,SUN几乎重构了整个J2EE体 系, 由于变化很大,干脆将名字也重构为Java EE, Java EE(当前版本为5.0)将元数据纳入很多规范当中,这其中就包括Web Services的相关规范, 加入元数据之后的Web Services服务器端编程模型就跟上面看到的C#片断差不多了, 这显然比以前的JAX-RPC编程模型简单(当然, Axis的编程模型也很简单).这里要谈的Web服务元数据(JSR 181)只是Java Web 服务规范中的一个,它跟Common Annotations, JAXB2, StAX, SAAJ和JAX-WS等共同构成Java EE 5的Web Services技术堆栈.

package WebServices;  
  
import java.io.File;  
import java.io.IOException;  
import javax.jws.Oneway;  
import javax.jws.WebMethod;  
import javax.jws.WebParam;  
import javax.jws.WebResult;  
import javax.jws.WebService;  
import javax.xml.ws.Endpoint;  
  
  
@WebService(targetNamespace="http://blog.csdn.net/chinajash",serviceName="HelloService")  
public class WSProvider {  
     @WebResult(name="Greetings")//自定义该方法返回值在WSDL中相关的描述      
     @WebMethod  
     public String sayHi(@WebParam(name="MyName") String name){  
         return "Hi,"+name; //@WebParam是自定义参数name在WSDL中相关的描述  
     }      
     @Oneway //表明该服务方法是单向的,既没有返回值,也不应该声明检查异常  
     @WebMethod(action="printSystemTime",operationName="printSystemTime")//自定义该方法在WSDL中相关的描述  
     public void printTime(){  
         System.out.println(System.currentTimeMillis());  
     }  
     public static void main(String[] args) {  
         Thread wsPublisher = new Thread(new WSPublisher());  
         wsPublisher.start();  
     }      
     private static class WSPublisher implements Runnable{  
         public void run() {  
             //发布WSProvider到http://localhost:8888/chinajash/WSProvider这个地址,之前必须调用wsgen命令  
             //生成服务类WSProvider的支持类,命令如下:  
             //wsgen -cp . WebServices.WSProvider  
             Endpoint.publish("http://localhost:8888/chinajash/WSProvider",new WSProvider());  
         }          
     }  
}  
  
如果想看到Web Services Engine生成的WSDL文件是否遵守上面的元数据, 我们没有必要将上面的WSProvider部署到支持JSR-181的应用服务器或Servlet形式的Web Services Engine,现在JDK6已经提供了一个很简单的机制可以用来测试和发布Web Services,下面讲讲如何在JDK6环境下发布Web Services和查看生成的WSDL  
1.将<JDK\_HOME>/bin加入path环境变量  
2.在命令行下切换当前目录到WSProvider的class文件所在的目录,运行下面命令  
wsgen -cp . WebServices.WSProvider  
在这个例子中会生成以下3个类的源代码文件及class文件  
SayHi  
SayHiResponse  
PrintTime  
3.执行如下代码发布WSProvider到http://localhost:8888/chinajash/WSProvider,在这里可以执行WSProvider类的main方法就可以  
Endpoint.publish("http://localhost:8888/chinajash/WSProvider",new WSProvider());  
4.在浏览器输入http://localhost:8888/chinajash/WSProvider?wsdl就可以看到生成的WSDL文件，为了节省篇幅,这里就不把生成的WSDL文件贴上了，大家可以自己动手试试.  
2.**脚本语言支持**

JDK6增加了对脚本语言的支持(JSR 223)， 原理上是将脚本语言编译成bytecode，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，另外，由于现在是编译成 bytecode后再执行，所以比原来边解释边执行效率要高很多。加入对脚本语言的支持后，对Java语言也提供了以下好处。  
1、许多脚本语言都有动态特性，比如，你不需要用一个变量之前先声明它，你可以用一个变量存放完全不同类型的对象，你不需要做强制类型转换，因为转换都是自动的。现在Java语言也可以通过对脚本语言的支持间接获得这种灵活性。  
2、 可以用脚本语言快速开发产品原型，因为现在可以Edit-Run，而无需Edit-Compile-Run，当然，因为Java有非常好的IDE支持，我 们完全可以在IDE里面编辑源文件，然后点击运行(隐含编译)，以此达到快速开发原型的目的，所以这点好处基本上可以忽略。  
3、通过引入脚本语言可以轻松实现Java应用程序的扩展和自定义，我们可以把原来分布在在Java应用程序中的配置逻辑，数学表达式和业务规则提取出来，转用JavaScript来处理。  
  
Sun的JDK6实现包含了一个基于Mozilla Rhino的 脚本语言引擎，支持JavaScript，这并不是说明JDK6只支持JavaScript，任何第三方都可以自己实现一个JSR-223兼容的脚本引擎 使得JDK6支持别的脚本语言，比如，你想让JDK6支持Ruby，那你可以自己按照JSR 223 的规范实现一个Ruby的脚本引擎类，具体一点，你需要实现javax.script.ScriptEngine(简单起见，可以继承 javax.script.AbstractScriptEngine)和javax.script.ScriptEngineFactory两个接口。 当然，在你实现自己的脚本语言引擎之前，先到scripting.dev.java.net project 这里看看是不是有人已经帮你做了工作，这样你就可以直接拿来用就行。  
  
Scripting API  
  
--------------------------------------------------------------------------------  
Scripting API是用于在Java里面编写脚本语言程序的API， 在Javax.script中可以找到Scripting API，我们就是用这个API来编写JavaScript程序，这个包里面有一个ScriptEngineManager类，它是使用Scripting API的入口，ScriptEngineManager可以通过jar服务发现(service discovery)机制寻找合适的脚本引擎类(ScriptEngine)，使用Scripting API的最简单方式只需下面三步  
1、创建一个ScriptEngineManager对象  
2、通过ScriptEngineManager获得ScriptEngine对象  
3、用ScriptEngine的eval方法执行脚本  
  
下面是一个Hello World程序  
  
public class HelloScript {public static void main(String[] args) throws Exception {         ScriptEngineManager factory = new ScriptEngineManager();//step 1         ScriptEngine engine = factory.getEngineByName("JavaScript");//Step 2             engine.eval\_r("print('Hello, Scripting')");//Step 3     }     }运行上面程序，控制台会输出Hello, Scripting上面这个简单的Scripting程序演示了如何在Java里面运行脚本语言，除此之外，我们还可以利用Scripting API实现以下功能1、暴露Java对象为脚本语言的全局变量2、在Java中调用脚本语言的方法3、脚本语言可以实现Java的接口4、脚本语言可以像Java一样使用JDK平台下的类下面的类演示了以上4种功能package Scripting;import java.io.File;import javax.script.Invocable;import javax.script.ScriptEngine;import javax.script.ScriptEngineManager;import javax.script.ScriptException;public class ScriptingAPITester {     public static void main(String[] args) throws Exception {         ScriptEngineManager manager = new ScriptEngineManager();         ScriptEngine engine = manager.getEngineByName("JavaScript");         testScriptVariables(engine);//演示如何暴露Java对象为脚本语言的全局变量          testInvokeScriptMethod(engine);//演示如何在Java中调用脚本语言的方法          testScriptInterface(engine);//演示脚本语言如何实现Java的接口          testUsingJDKClasses(engine);//演示脚本语言如何使用JDK平台下的类     }         public static void testScriptVariables(ScriptEngine engine) throws ScriptException{         File file = new File("test.txt");         engine.put("f", file);         engine.eval\_r("println('Total Space:'+f.getTotalSpace())");             }         public static void testInvokeScriptMethod(ScriptEngine engine) throws Exception{         String script = "function hello(name) { return 'Hello,' + name;}";         engine.eval\_r(script);         Invocable inv = (Invocable) engine;         String res = (String)inv.invokeFunction("hello", "Scripting" );         System.out.println("res:"+res);     }         public static void testScriptInterface(ScriptEngine engine) throws ScriptException{         String script = "var obj = new Object(); obj.run = function() { println('run method called'); }";         engine.eval\_r(script);         Object obj = engine.get("obj");         Invocable inv = (Invocable) engine;         Runnable r = inv.getInterface(obj,Runnable.class);         Thread th = new Thread(r);         th.start();     }         public static void testUsingJDKClasses(ScriptEngine engine) throws Exception{         //Packages是脚本语言里的一个全局变量,专用于访问JDK的package         String js = "function doSwing(t){var f=new Packages.javax.swing.JFrame(t);f.setSize(400,300);f.setVisible(true);}";         engine.eval\_r(js);         Invocable inv = (Invocable) engine;         inv.invokeFunction("doSwing", "Scripting Swing" );     }}Scripting Tool  
  
--------------------------------------------------------------------------------  
SUN 提供的JDK6中有一个命令行工具??jrunscript，你可以在<JDK6\_Home>/bin下面找到这个工 具，jrunscript是一个脚本语言的解释程序，它独立于脚本语言，但默认是用JavaScript，我们可以用jrunscript来测试自己写的 脚本语言是否正确，下面是一个在命令行运行jrunscript的简单例子  
jrunscript  
js>println("Hello,JrunScript");  
Hello,JrunScript  
js>9\*8  
72.0  
js>  
3.**JTable的排序和过滤**

原来的JTable基本上是只能显示数据，在JDK6新增了对JTable的排序和过滤功能，下面代码演示了这两个功能  
  
  
public class JTableTester {  
     static String data[][] = {  
         {"China","Beijing","Chinese"},  
         {"America","Washington","English"},  
         {"Korea","Seoul","Korean"},  
         {"Japan","Tokyo","Japanese"},  
         {"France","Paris","French"},  
         {"England","London","English"},  
         {"Germany","Berlin","German"},  
     };  
     static String titles[] = {"Country","Capital","Language"};  
     public static void main(String[] args) {          
         DefaultTableModel m = new DefaultTableModel(data,titles);  
         JTable t = new JTable(m);  
         final TableRowSorter sorter = new TableRowSorter(m);   
         t.setRowSorter(sorter); //为JTable设置排序器  
          
         JScrollPane sPane = new JScrollPane();  
         sPane.setViewportView(t);  
          
         JPanel p = new JPanel();  
         p.setLayout(new BoxLayout(p,BoxLayout.X\_AXIS));  
         JLabel l = new JLabel("Criteria:");  
         final JTextField tf = new JTextField();  
         JButton b = new JButton("Do Filter");  
         p.add(l);  
         p.add(tf);  
         p.add(b);  
         b.addActionListener(new ActionListener() {  
             public void actionPerformed(ActionEvent e) {  
                 if(tf.getText().length()==0){  
                     sorter.setRowFilter(null);  
                 }else{  
                     sorter.setRowFilter(RowFilter.regexFilter(tf.getText()));//为JTable设置基于正则表达式的过滤条件  
                 }  
             }  
         });  
          
         JFrame f = new JFrame("JTable Sorting and Filtering");  
         f.getContentPane().add(sPane,BorderLayout.CENTER);          
         f.getContentPane().add(p,BorderLayout.SOUTH);  
         f.setSize(400,300);  
         f.setVisible(true);  
     }  
}   
运行上面程序，单击JTable的某一个title，这个title对应的列就会按照升序/降序重新排列；在下面的Criteria文本框中输入"ese"，点击"Do Filter"按钮，JTable将只显示带有"ese"字符串的行，也就是China和Japan两行，如果文本框里面什么都没有，点击"Do Filter"按钮，这时JTable会显示所有的行。  
4.**更简单,更强大的JAX-WS**

JAX-WS2.0的来历  
  
--------------------------------------------------------------------------------  
JAX-WS(JSR-224) 是Java Architecture for XML Web Services的缩写,简单说就是一种用Java和XML开发Web Services应用程序的框架, 目前版本是2.0, 它是JAX-RPC 1.1的后续版本, J2EE 1.4带的就是JAX-RPC1.1, 而Java EE 5里面包括了JAX-WS 2.0,但为了向后兼容,仍然支持JAX-RPC. 现在,SUN又把JAX-WS直接放到了Java SE 6里面,由于JAX-WS会用到Common Annotation(JSR 250),Java Web Services Metadata(JSR 181), JAXB2(JSR 222), StAX(JSR 173), 所以SUN也必须把后几个原属于Java EE范畴的Components下放到Java SE, 现在我们可以清楚地理解了为什么Sun要把这些看似跟Java SE没有关系的Components放进来,终极目的就是要在Java SE里面支持Web Services.  
  
JAX-WS2.0的架构  
  
--------------------------------------------------------------------------------  
JAX-WS不是一个孤立的框架,它依赖于众多其他的规范,本质上它由以下几部分组成  
1.用来开发Web Services的Java API  
2.用来处理Marshal/Unmarshal的XML Binding机制,JAX-WS2.0用JAXB2来处理Java Object与XML之间的映射,Marshalling就是把Java Object映射到XML,Unmarshalling则是把XML映射到Java Object.之所以要做Java Object与XML的映射,是因为最终作为方法参数和返回值的Java Object要通过网络传输协议(一般是SOAP)传送,这就要求必须对Java Object做类似序列化和反序列化的工作,在SOAP中就是要用XML来表示Java object的内部状态  
3.众多元数据(Annotations)会被JAX-WS用来描述Web Services的相关类,包括Common Annotations, Web Services Metadata, JAXB2的元数据和JAX-WS2.0规范自己的元数据.  
4.Annotation Processing Tool(APT) 是JAX-WS重要的组成部分,由于JAX-WS2.0规范用到很多元数据,所以需要APT来处理众多的Annotations. 在<JDK\_HOME>/bin下有两个命令wsgen和wsimport,就是用到APT和Compiler API来处理碰到的Annotations,wsgen可以为Web Services Provider产生并编译必要的帮助类和相关支持文件,wsimport以WSDL作为输入为Web Service Consumer产生并编译必要的帮助类和相关支持文件.  
5.JAX-WS还包括JAX-WS Runtime与应用服务器和工具之间的契约关系  
  
JAX-WS2.0的编程模型  
  
--------------------------------------------------------------------------------  
现在用JAX-WS2.0来编写Web Services非常简单,不像JAX-RPC,JAX-WS可以把任意POJO暴露为Web Services,服务类不需要实现接口,服务方法也没有必要抛出RMI异常.下面介绍在JDK6环境下用JAX-WS2.0开发和测试Web Services的步骤  
1.编写服务类,并用Web Services Metadata(JSR-181)标注这个服务类,我用我的另一篇BlogJDK6的新特性之十:Web服务元数据中的WSProvider类作为服务类的例子,在此我重复贴一下WSProvider类的源代码:  
  
@WebService(targetNamespace="http://blog.csdn.net/chinajash",serviceName="HelloService")  
public class WSProvider {  
     @WebResult(name="Greetings")//自定义该方法返回值在WSDL中相关的描述      
     @WebMethod  
     public String sayHi(@WebParam(name="MyName") String name){  
         return "Hi,"+name; //@WebParam是自定义参数name在WSDL中相关的描述  
     }      
     @Oneway //表明该服务方法是单向的,既没有返回值,也不应该声明检查异常  
     @WebMethod(action="printSystemTime",operationName="printSystemTime")//自定义该方法在WSDL中相关的描述  
     public void printTime(){  
         System.out.println(System.currentTimeMillis());  
     }  
     public static void main(String[] args) {  
         Thread wsPublisher = new Thread(new WSPublisher());  
         wsPublisher.start();  
     }      
     private static class WSPublisher implements Runnable{  
         public void run() {  
             //发布WSProvider到http://localhost:8888/chinajash/WSProvider这个地址,之前必须调用wsgen命令  
             //生成服务类WSProvider的支持类,命令如下:  
             //wsgen -cp . WebServices.WSProvider  
             Endpoint.publish("http://localhost:8888/chinajash/WSProvider",new WSProvider());  
         }          
     }  
}  
2.用wsgen生成上面服务类的必要的帮助类,然后调用用EndPoint类的静态方法publish发布服务类(步骤请参考我的另一篇Blog JDK6的新特性之十:Web服务元数据),我在这里是将服务类发布到http://localhost:8888/chinajash/WSProvider  
3.用wsimport为服务消费者(也就是服务的客户端)生成必要的帮助类,命令如下:  
wsimport http://localhost:8888/chinajash/WSProvider?wsdl  
这会在<当前目录>\net\csdn\blog\chinajash下生成客户端的帮助类,在这个例子中会生成7个类  
HelloService.class  
ObjectFactory.class  
package-info.class  
PrintSystemTime.class  
SayHi.class  
SayHiResponse.class  
WSProvider.class  
4.在客户端用下面代码即可调用步骤1定义的Web Service  
HelloService hs = new HelloService();  
WSProvider ws = hs.getWSProviderPort();  
System.out.println(ws.sayHi("chinajash"));  
ws.printSystemTime();  
调用上述代码后客户端控制台输出  
hi,chinajash  
服务端控制台输出服务器当前系统时间  
5.**轻量级Http Server**

JDK6的新特性之五:轻量级Http Server  
  
JDK6提供了一个简单的Http Server API,据此我们可以构建自己的嵌入式Http Server,它支持Http和Https协议,提供了HTTP1.1的部分实现，没有被实现的那部分可以通过扩展已有的Http Server API 来实现,程序员必须自己实现HttpHandler接口,HttpServer会调用HttpHandler实现类的回调方法来处理客户端请求,在这里, 我们把一个Http请求和它的响应称为一个交换,包装成HttpExchange类,HttpServer负责将HttpExchange传给 HttpHandler实现类的回调方法.下面代码演示了怎样创建自己的Http Server  
  
  
public class HTTPServerAPITester {  
     public static void main(String[] args) {  
         try {  
             HttpServer hs = HttpServer.create(new InetSocketAddress(8888),0);//设置HttpServer的端口为8888  
             hs.createContext("/chinajash", new MyHandler());//用MyHandler类内处理到/chinajash的请求  
             hs.setExecutor(null); // creates a default executor  
             hs.start();  
         } catch (IOException e) {  
             e.printStackTrace();  
         }  
     }  
}  
  
class MyHandler implements HttpHandler {  
    public void handle(HttpExchange t) throws IOException {  
        InputStream is = t.getRequestBody();  
        String response = "<h3>Happy New Year 2007!--Chinajash</h3>";  
        t.sendResponseHeaders(200, response.length());  
        OutputStream os = t.getResponseBody();  
        os.write(response.getBytes());  
        os.close();  
    }  
}  
  
运行程序后,在浏览器内输入<http://localhost:8888/xx>,浏览器输出  
6.**嵌入式数据库 Derby**

Derby是IBM送给开源社区的又一个礼物，是一个pure java的数据库，现在已经被列入到java1.6中。  
不知道对于大数据量的性能如何，但传说中启动derby只会给JVM添加2M的内存，对那些小数据库应用，比如像用access那种应该是挺有诱惑力的。  
另外，麻雀虽小，五脏俱全，功能要比access多得多咯，包括事务处理，并发，触发器都有，管理又简单，因此自己用来做点工具正好合适。  
废话少说，介绍一下我折腾了半天的经验吧。  
我的Derby配置过程：  
1，下载db-derby-10.1.3.1-bin.tar.gz,derby\_core\_plugin\_10.1.3.zip和derby\_ui\_plugin\_1.1.0.zip，把两个插件安装到eclipse上  
2，打开ecllipse,新建一个project  
3，右键这个project，选择Apache Derby，再选择add apache derby native，发现只是给我的project添加了几个derby的jar，还不是在我看着顺眼的lib目录里，索性干掉，换上db-derby- 10.1.3.1-bin.tar.gz解压出来以后lib目录下的jar文件，在Build Path里设置一下；  
4，右键Project，在apache derby里选择start apache derby network server，控制台可以看到derby启动后打出的“服务器准备在端口 1527 上接受连接。”  
5，右键Project，在apache derby里选择ij(Interactive SQL)，启动SQL控制台；  
6，输入connect jdbc:derby:testdb;create=true; 注意要有单引号，可以在工程跟目录下创建testdb数据库，可以看到一个新建的目录testdb，那里的文件就是数据库咯；  
7，用标准的SQL语句来建一个数据库试试：  
create table test (a varchar(4) not null, b char(2) primary key);  
居然可以用，太神奇了，呵呵  
8，再插入一条语句试试呢，insert into test(a,b) values(a,11);，嗯，不错，可以用select 查出来的哦。  
9，再插一下：insert into test(a,b) values(a,11);，哦哦，报错了，“错误 23505：语句异常终止，因为它导致“TEST”上所定义的“SQL060710092132480”标识的唯一或主键约束或唯一索引中出现重复键值。” 呵呵。  
10，好了，现在可以像你控制的其他数据库一样来控制Derby了。  
  
  
如果上述方法不行，或者你习惯了在eclipse之外使用和管理数据库，那么可以很方便的把Derby“装”在系统里。下面我说一下步骤：  
1，把db-derby-10.1.3.1-bin.tar.gz解压到c:\derby,使lib和framework两个目录在c:\derby下边即可  
2，设置环境变量  
  
设置一个c:\derby\framework\embeded\bin或c:\derby\framework\NetworkServe\bin到Path中，这样我们就可以直接执行上边介绍的connect这样的命令而不用每次钻到那个目录下去执行了   
设置c:\derby\lib\derby.jar;c:\derby\lib\derbytoos.jar到CLASSPATH中，以便让这些java编成的命令能够正确执行；  
3，打开cmd  
4，敲入startNetworkServer，可以看到像在eclisp中提示的那样启动了server  
5，再打开一个cmd,敲入sysinfo,可以看到derby的环境信息了，注意在java user dir这一项，也许是java用户目录上和上边看到的会有所不同哦，这样在connect jdbc:derby:testdb;create=true;的建的数据库目录就不一样咯。  
6，敲入ij，好了，进入到上边的交互界面，可以建一个数据库看看了。  
7，最后在另外一个cmd中敲入stopNetworkServer就可以关闭数据库了。  
  
如果你两种方法都试过了，那么需要注意的，还是上边步骤5的问题，这个问题是你可能随时会启动一个数据库或新建一个数据库，但如果你刚刚使用derby，你可能还没有察觉。  
derby实际上有两种启动方式，一种是嵌入式的，一种是网络服务器的启动。  
1，我们在eclipse中右键start apache derby network server那个，就是网络服务器的启动方式，在这种方式下可以用另外一台计算机在ij中以：  
connect jdbc:derby://192.168.0.28:1527/testdb   
的方式进行链接。  
2，第二种启动方式是在ij里边就直接  
connect jdbc:derby:testdb  
这实际是在连当前配置环境下java user dir下那个目录的数据库。  
  
看到这里可能有点糊涂了，这么就会出问题了那？  
实际上derby的访问更像是一种使用derby driver对本地文件系统的访问，不管启动不启动网络服务器，都可以用driver访问本地的数据库。这样，在ij里边像第二种方式那样建立连接是完全可以的。启动了网络服务器，只不过是能够让其他主机访问罢了。  
  
另外一个问题是，在eclipse中和在系统中连接服务器，在connect的时候这个当前配置环境是不一样的，eclipse默认工程所在路径是数据库的所在路径，而在系统中“装”derby则会认为 c:\document and settings下边那个用户目录是数据库的所在路径。  
jdk1.7新特性：

1，switch中可以使用字串了  
String s = "test";     
switch (s) {     
case "test" :     
     System.out.println("test");   
case "test1" :     
    System.out.println("test1");   
    break ;     
default :     
    System.out.println("break");   
    break ;     
}

2.运用List<String> tempList = new ArrayList<>(); 即泛型实例化类型自动推断  
3.语法上支持集合，而不一定是数组

final List<Integer> piDigits = [ 1,2,3,4,5,8 ];     
4.新增一些取环境信息的工具方法

File System.getJavaIoTempDir() // IO临时文件夹

File System.getJavaHomeDir() // JRE的安装目录

File System.getUserHomeDir() // 当前用户目录

File System.getUserDir() // 启动java进程时所在的目录5

5.Boolean类型反转，空指针安全,参与位运算

Boolean Booleans.negate(Boolean booleanObj)

True => False , False => True, Null => Null

boolean Booleans.and(boolean[] array)

boolean Booleans.or(boolean[] array)

boolean Booleans.xor(boolean[] array)

boolean Booleans.and(Boolean[] array)

boolean Booleans.or(Boolean[] array)

boolean Booleans.xor(Boolean[] array)

6.两个char间的equals   
boolean Character.equalsIgnoreCase(char ch1, char ch2)  
7.安全的加减乘除   
int Math.safeToInt(long value)

int Math.safeNegate(int value)

long Math.safeSubtract(long value1, int value2)

long Math.safeSubtract(long value1, long value2)

int Math.safeMultiply(int value1, int value2)

long Math.safeMultiply(long value1, int value2)

long Math.safeMultiply(long value1, long value2)

long Math.safeNegate(long value)

int Math.safeAdd(int value1, int value2)

long Math.safeAdd(long value1, int value2)

long Math.safeAdd(long value1, long value2)

int Math.safeSubtract(int value1, int value2)

8.map集合支持并发请求，且可以写成 Map map = {name:"xxx",age:18};

--------------------------------------------------------------------------------------------------------------

欢迎阅读我编写的[Java 8](https://jdk8.java.net/)介绍。本教程将带领你一步一步地认识这门语言的新特性。通过简单明了的代码示例，你将会学习到如何使用默认接口方法，Lambda表达式，方法引用和重复注解。看完这篇教程后，你还将对最新推出的[API](http://download.java.net/jdk8/docs/api/)有一定的了解，例如：流控制，函数式接口，map扩展和新的时间日期API等等。

#### 目 录 [[ - ]](http://www.iteye.com/magazines/129-Java-8-Tutorial)

1. [允许在接口中有默认方法实现](http://www.iteye.com/magazines/129-Java-8-Tutorial#548)
2. [Lambda表达式](http://www.iteye.com/magazines/129-Java-8-Tutorial#549)
3. [函数式接口](http://www.iteye.com/magazines/129-Java-8-Tutorial#550)
4. [方法和构造函数引用](http://www.iteye.com/magazines/129-Java-8-Tutorial#551)
5. [Lambda的范围](http://www.iteye.com/magazines/129-Java-8-Tutorial#552)
6. [内置函数式接口](http://www.iteye.com/magazines/129-Java-8-Tutorial#553)
7. [Streams](http://www.iteye.com/magazines/129-Java-8-Tutorial#554)
8. [Parallel Streams](http://www.iteye.com/magazines/129-Java-8-Tutorial#555)
9. [Map](http://www.iteye.com/magazines/129-Java-8-Tutorial#556)
10. [时间日期API](http://www.iteye.com/magazines/129-Java-8-Tutorial#557)
11. [Annotations](http://www.iteye.com/magazines/129-Java-8-Tutorial#558)
12. [总结](http://www.iteye.com/magazines/129-Java-8-Tutorial#559)

## 允许在接口中有默认方法实现 [Top](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)

Java 8 允许我们使用default关键字，为接口声明添加非抽象的方法实现。这个特性又被称为扩展方法。下面是我们的第一个例子： 

**Java代码**

1. **interface** Formula {
2. **double** calculate(**int** a);
4. **default** **double** sqrt(**int** a) {
5. **return** Math.sqrt(a);
6. }
7. }

在接口Formula中，除了抽象方法caculate以外，还定义了一个默认方法sqrt。Formula的实现类只需要实现抽象方法caculate就可以了。默认方法sqrt可以直接使用。 

**Java代码**

1. Formula formula = **new** Formula() {
2. @Override
3. **public** **double** calculate(**int** a) {
4. **return** sqrt(a \* 100);
5. }
6. };
8. formula.calculate(100);     // 100.0
9. formula.sqrt(16);           // 4.0

formula对象以匿名对象的形式实现了Formula接口。代码很啰嗦：用了6行代码才实现了一个简单的计算功能：a\*100开平方根。我们在下一节会看到，Java 8 还有一种更加优美的方法，能够实现包含单个函数的对象。 

## Lambda表达式 [Top](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)

让我们从最简单的例子开始，来学习如何对一个string列表进行排序。我们首先使用Java 8之前的方法来实现： 

**Java代码**

1. List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");
3. Collections.sort(names, **new** Comparator<String>() {
4. @Override
5. **public** **int** compare(String a, String b) {
6. **return** b.compareTo(a);
7. }
8. });

静态工具方法Collections.sort接受一个list，和一个Comparator接口作为输入参数，Comparator的实现类可 以对输入的list中的元素进行比较。通常情况下，你可以直接用创建匿名Comparator对象，并把它作为参数传递给sort方法。   
  
除了创建匿名对象以外，Java 8 还提供了一种更简洁的方式，Lambda表达式。 

**Java代码**

1. Collections.sort(names, (String a, String b) -> {
2. **return** b.compareTo(a);
3. });

你可以看到，这段代码就比之前的更加简短和易读。但是，它还可以更加简短： 

**Java代码**

1. Collections.sort(names, (String a, String b) -> b.compareTo(a));

只要一行代码，包含了方法体。你甚至可以连大括号对{}和return关键字都省略不要。不过这还不是最短的写法： 

**Java代码**

1. Collections.sort(names, (a, b) -> b.compareTo(a));

Java编译器能够自动识别参数的类型，所以你就可以省略掉类型不写。让我们再深入地研究一下lambda表达式的威力吧。 

## 函数式接口 [Top](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)

Lambda表达式如何匹配Java的类型系统？每一个lambda都能够通过一个特定的接口，与一个给定的类型进行匹配。一个所谓的函数式接口必须要有 且仅有一个抽象方法声明。每个与之对应的lambda表达式必须要与抽象方法的声明相匹配。由于默认方法不是抽象的，因此你可以在你的函数式接口里任意添 加默认方法。   
  
任意只包含一个抽象方法的接口，我们都可以用来做成lambda表达式。为了让你定义的接口满足要求，你应当在接口前加上@FunctionalInterface 标注。编译器会注意到这个标注，如果你的接口中定义了第二个抽象方法的话，编译器会抛出异常。   
  
举例： 

**Java代码**

1. @FunctionalInterface
2. **interface** Converter<F, T> {
3. T convert(F from);
4. }
6. Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
7. Integer converted = converter.convert("123");
8. System.out.println(converted);    // 123

注意，如果你不写@FunctionalInterface 标注，程序也是正确的。 

## 方法和构造函数引用 [Top](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)

上面的代码实例可以通过静态方法引用，使之更加简洁： 

**Java代码**

1. Converter<String, Integer> converter = Integer::valueOf;
2. Integer converted = converter.convert("123");
3. System.out.println(converted);   // 123

Java 8 允许你通过::关键字获取方法或者构造函数的的引用。上面的例子就演示了如何引用一个静态方法。而且，我们还可以对一个对象的方法进行引用： 

**Java代码**

1. **class** Something {
2. String startsWith(String s) {
3. **return** String.valueOf(s.charAt(0));
4. }
5. }
7. Something something = **new** Something();
8. Converter<String, String> converter = something::startsWith;
9. String converted = converter.convert("Java");
10. System.out.println(converted);    // "J"

让我们看看如何使用::关键字引用构造函数。首先我们定义一个示例bean，包含不同的构造方法： 

**Java代码**

1. **class** Person {
2. String firstName;
3. String lastName;
5. Person() {}
7. Person(String firstName, String lastName) {
8. **this**.firstName = firstName;
9. **this**.lastName = lastName;
10. }
11. }

接下来，我们定义一个person工厂接口，用来创建新的person对象： 

**Java代码**

1. **interface** PersonFactory<P **extends** Person> {
2. P create(String firstName, String lastName);
3. }

然后我们通过构造函数引用来把所有东西拼到一起，而不是像以前一样，通过手动实现一个工厂来这么做。 

**Java代码**

1. PersonFactory<Person> personFactory = Person::**new**;
2. Person person = personFactory.create("Peter", "Parker");

我们通过Person::new来创建一个Person类构造函数的引用。Java编译器会自动地选择合适的构造函数来匹配PersonFactory.create函数的签名，并选择正确的构造函数形式。 

## Lambda的范围 [Top](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)

对于lambdab表达式外部的变量，其访问权限的粒度与匿名对象的方式非常类似。你能够访问局部对应的外部区域的局部final变量，以及成员变量和静态变量。   
  
**访问局部变量**   
  
我们可以访问lambda表达式外部的final局部变量： 

**Java代码**

1. **final** **int** num = 1;
2. Converter<Integer, String> stringConverter =
3. (from) -> String.valueOf(from + num);
5. stringConverter.convert(2);     // 3

但是与匿名对象不同的是，变量num并不需要一定是final。下面的代码依然是合法的： 

**Java代码**

1. **int** num = 1;
2. Converter<Integer, String> stringConverter =
3. (from) -> String.valueOf(from + num);
5. stringConverter.convert(2);     // 3

然而，num在编译的时候被隐式地当做final变量来处理。下面的代码就不合法： 

**Java代码**

1. **int** num = 1;
2. Converter<Integer, String> stringConverter =
3. (from) -> String.valueOf(from + num);
4. num = 3;

在lambda表达式内部企图改变num的值也是不允许的。   
  
**访问成员变量和静态变量**   
  
与局部变量不同，我们在lambda表达式的内部能获取到对成员变量或静态变量的读写权。这种访问行为在匿名对象里是非常典型的。 

**Java代码**

1. **class** Lambda4 {
2. **static** **int** outerStaticNum;
3. **int** outerNum;
5. **void** testScopes() {
6. Converter<Integer, String> stringConverter1 = (from) -> {
7. outerNum = 23;
8. **return** String.valueOf(from);
9. };
11. Converter<Integer, String> stringConverter2 = (from) -> {
12. outerStaticNum = 72;
13. **return** String.valueOf(from);
14. };
15. }
16. }

**访问默认接口方法**   
  
还记得第一节里面formula的那个例子么？ 接口Formula定义了一个默认的方法sqrt，该方法能够访问formula所有的对象实例，包括匿名对象。这个对lambda表达式来讲则无效。   
  
默认方法无法在lambda表达式内部被访问。因此下面的代码是无法通过编译的： 

**Java代码**

1. Formula formula = (a) -> sqrt( a \* 100);

## 内置函数式接口 [Top](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)

JDK 1.8 API中包含了很多内置的函数式接口。有些是在以前版本的Java中大家耳熟能详的，例如Comparator接口，或者Runnable接口。对这些现 成的接口进行实现，可以通过@FunctionalInterface 标注来启用Lambda功能支持。   
  
此外，Java 8 API 还提供了很多新的函数式接口，来降低程序员的工作负担。有些新的接口已经在[Google Guava](https://code.google.com/p/guava-libraries/)库中很有名了。如果你对这些库很熟的话，你甚至闭上眼睛都能够想到，这些接口在类库的实现过程中起了多么大的作用。   
  
**Predicates**   
  
Predicate是一个布尔类型的函数，该函数只有一个输入参数。Predicate接口包含了多种默认方法，用于处理复杂的逻辑动词（and, or，negate）： 

**Java代码**

1. Predicate<String> predicate = (s) -> s.length() > 0;
3. predicate.test("foo");              // true
4. predicate.negate().test("foo");     // false
6. Predicate<Boolean> nonNull = Objects::nonNull;
7. Predicate<Boolean> isNull = Objects::isNull;
9. Predicate<String> isEmpty = String::isEmpty;
10. Predicate<String> isNotEmpty = isEmpty.negate();

**Functions**   
  
Function接口接收一个参数，并返回单一的结果。默认方法可以将多个函数串在一起（compse, andThen）： 

**Java代码**

1. Function<String, Integer> toInteger = Integer::valueOf;
2. Function<String, String> backToString = toInteger.andThen(String::valueOf);
4. backToString.apply("123");     // "123"

**Suppliers**   
  
Supplier接口产生一个给定类型的结果。与Function不同的是，Supplier没有输入参数。 

**Java代码**

1. Supplier<Person> personSupplier = Person::**new**;
2. personSupplier.get();   // new Person

**Consumers**   
  
Consumer代表了在一个输入参数上需要进行的操作。 

**Java代码**

1. Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);
2. greeter.accept(**new** Person("Luke", "Skywalker"));

**Comparators**   
  
Comparator接口在早期的Java版本中非常著名。Java 8 为这个接口添加了不同的默认方法。 

**Java代码**

1. Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);
3. Person p1 = **new** Person("John", "Doe");
4. Person p2 = **new** Person("Alice", "Wonderland");
6. comparator.compare(p1, p2);             // > 0
7. comparator.reversed().compare(p1, p2);  // < 0

**Optionals**   
  
Optional不是一个函数式接口，而是一个精巧的工具接口，用来防止NullPointerEception产生。这个概念在下一节会显得很重要，所以我们在这里快速地浏览一下Optional的工作原理。   
  
Optional是一个简单的值容器，这个值可以是null，也可以是non-null。考虑到一个方法可能会返回一个non-null的值，也可能返回一个空值。为了不直接返回null，我们在Java 8中就返回一个Optional。 

**Java代码**

1. Optional<String> optional = Optional.of("bam");
3. optional.isPresent();           // true
4. optional.get();                 // "bam"
5. optional.orElse("fallback");    // "bam"
7. optional.ifPresent((s) -> System.out.println(s.charAt(0)));     // "b"

## Streams [Top](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)

java.util.Stream表示了某一种元素的序列，在这些元素上可以进行各种操作。Stream操作可以是中间操作，也可以是完结操作。完结操作 会返回一个某种类型的值，而中间操作会返回流对象本身，并且你可以通过多次调用同一个流操作方法来将操作结果串起来（就像StringBuffer的 append方法一样————译者注）。Stream是在一个源的基础上创建出来的，例如java.util.Collection中的list或者 set（map不能作为Stream的源）。Stream操作往往可以通过顺序或者并行两种方式来执行。   
  
我们先了解一下序列流。首先，我们通过string类型的list的形式创建示例数据： 

**Java代码**

1. List<String> stringCollection = **new** ArrayList<>();
2. stringCollection.add("ddd2");
3. stringCollection.add("aaa2");
4. stringCollection.add("bbb1");
5. stringCollection.add("aaa1");
6. stringCollection.add("bbb3");
7. stringCollection.add("ccc");
8. stringCollection.add("bbb2");
9. stringCollection.add("ddd1");

Java 8中的Collections类的功能已经有所增强，你可以之直接通过调用Collections.stream()或者Collection.parallelStream()方法来创建一个流对象。下面的章节会解释这个最常用的操作。   
  
**Filter**   
  
Filter接受一个predicate接口类型的变量，并将所有流对象中的元素进行过滤。该操作是一个中间操作，因此它允许我们在返回结果的基 础上再进行其他的流操作（forEach）。ForEach接受一个function接口类型的变量，用来执行对每一个元素的操作。ForEach是一个 中止操作。它不返回流，所以我们不能再调用其他的流操作。 

**Java代码**

1. stringCollection
2. .stream()
3. .filter((s) -> s.startsWith("a"))
4. .forEach(System.out::println);
6. // "aaa2", "aaa1"

**Sorted**   
  
Sorted是一个中间操作，能够返回一个排过序的流对象的视图。流对象中的元素会默认按照自然顺序进行排序，除非你自己指定一个Comparator接口来改变排序规则。 

**Java代码**

1. stringCollection
2. .stream()
3. .sorted()
4. .filter((s) -> s.startsWith("a"))
5. .forEach(System.out::println);
7. // "aaa1", "aaa2"

一定要记住，sorted只是创建一个流对象排序的视图，而不会改变原来集合中元素的顺序。原来string集合中的元素顺序是没有改变的。 

**Java代码**

1. System.out.println(stringCollection);
2. // ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1

**Map**   
  
map是一个对于流对象的中间操作，通过给定的方法，它能够把流对象中的每一个元素对应到另外一个对象上。下面的例子就演示了如何把每个 string都转换成大写的string. 不但如此，你还可以把每一种对象映射成为其他类型。对于带泛型结果的流对象，具体的类型还要由传递给map的泛型方法来决定。 

**Java代码**

1. stringCollection
2. .stream()
3. .map(String::toUpperCase)
4. .sorted((a, b) -> b.compareTo(a))
5. .forEach(System.out::println);
7. // "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"

**Match**   
  
匹配操作有多种不同的类型，都是用来判断某一种规则是否与流对象相互吻合的。所有的匹配操作都是终结操作，只返回一个boolean类型的结果。 

**Java代码**

1. **boolean** anyStartsWithA =
2. stringCollection
3. .stream()
4. .anyMatch((s) -> s.startsWith("a"));
6. System.out.println(anyStartsWithA);      // true
8. **boolean** allStartsWithA =
9. stringCollection
10. .stream()
11. .allMatch((s) -> s.startsWith("a"));
13. System.out.println(allStartsWithA);      // false
15. **boolean** noneStartsWithZ =
16. stringCollection
17. .stream()
18. .noneMatch((s) -> s.startsWith("z"));
20. System.out.println(noneStartsWithZ);      // true

**Count**   
  
Count是一个终结操作，它的作用是返回一个数值，用来标识当前流对象中包含的元素数量。 

**Java代码**

1. **long** startsWithB =
2. stringCollection
3. .stream()
4. .filter((s) -> s.startsWith("b"))
5. .count();
7. System.out.println(startsWithB);    // 3

**Reduce**   
  
该操作是一个终结操作，它能够通过某一个方法，对元素进行削减操作。该操作的结果会放在一个Optional变量里返回。 

**Java代码**

1. Optional<String> reduced =
2. stringCollection
3. .stream()
4. .sorted()
5. .reduce((s1, s2) -> s1 + "#" + s2);
7. reduced.ifPresent(System.out::println);
8. // "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"

## Parallel Streams [Top](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)

像上面所说的，流操作可以是顺序的，也可以是并行的。顺序操作通过单线程执行，而并行操作则通过多线程执行。   
  
下面的例子就演示了如何使用并行流进行操作来提高运行效率，代码非常简单。   
  
首先我们创建一个大的list，里面的元素都是唯一的： 

**Java代码**

1. **int** max = 1000000;
2. List<String> values = **new** ArrayList<>(max);
3. **for** (**int** i = 0; i < max; i++) {
4. UUID uuid = UUID.randomUUID();
5. values.add(uuid.toString());
6. }

现在，我们测量一下对这个集合进行排序所使用的时间。   
  
**顺序排序** 

**Java代码**

1. **long** t0 = System.nanoTime();
3. **long** count = values.stream().sorted().count();
4. System.out.println(count);
6. **long** t1 = System.nanoTime();
8. **long** millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
9. System.out.println(String.format("sequential sort took: %d ms", millis));
11. // sequential sort took: 899 ms

**并行排序** 

**Java代码**

1. **long** t0 = System.nanoTime();
3. **long** count = values.parallelStream().sorted().count();
4. System.out.println(count);
6. **long** t1 = System.nanoTime();
8. **long** millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
9. System.out.println(String.format("parallel sort took: %d ms", millis));
11. // parallel sort took: 472 ms

如你所见，所有的代码段几乎都相同，唯一的不同就是把stream()改成了parallelStream(), 结果并行排序快了50%。 

## Map [Top](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)

正如前面已经提到的那样，map是不支持流操作的。而更新后的map现在则支持多种实用的新方法，来完成常规的任务。 

**Java代码**

1. Map<Integer, String> map = **new** HashMap<>();
3. **for** (**int** i = 0; i < 10; i++) {
4. map.putIfAbsent(i, "val" + i);
5. }
7. map.forEach((id, val) -> System.out.println(val));

上面的代码风格是完全自解释的：putIfAbsent避免我们将null写入；forEach接受一个消费者对象，从而将操作实施到每一个map中的值上。   
  
下面的这个例子展示了如何使用函数来计算map的编码： 

**Java代码**

1. map.computeIfPresent(3, (num, val) -> val + num);
2. map.get(3);             // val33
4. map.computeIfPresent(9, (num, val) -> **null**);
5. map.containsKey(9);     // false
7. map.computeIfAbsent(23, num -> "val" + num);
8. map.containsKey(23);    // true
10. map.computeIfAbsent(3, num -> "bam");
11. map.get(3);             // val33

接下来，我们将学习，当给定一个key值时，如何把一个实例从对应的key中移除： 

**Java代码**

1. map.remove(3, "val3");
2. map.get(3);             // val33
4. map.remove(3, "val33");
5. map.get(3);             // null

另一个有用的方法： 

**Java代码**

1. map.getOrDefault(42, "not found");  // not found

将map中的实例合并也是非常容易的： 

**Java代码**

1. map.merge(9, "val9", (value, newValue) -> value.concat(newValue));
2. map.get(9);             // val9
4. map.merge(9, "concat", (value, newValue) -> value.concat(newValue));
5. map.get(9);             // val9concat

合并操作先看map中是否没有特定的key/value存在，如果是，则把key/value存入map，否则merging函数就会被调用，对现有的数值进行修改。 

## 时间日期API [Top](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)

Java 8 包含了全新的时间日期API，这些功能都放在了java.time包下。新的时间日期API是基于Joda-Time库开发的，但是也不尽相同。下面的例子就涵盖了大多数新的API的重要部分。   
  
**Clock**   
  
Clock提供了对当前时间和日期的访问功能。Clock是对当前时区敏感的，并可用于替代 System.currentTimeMillis()方法来获取当前的毫秒时间。当前时间线上的时刻可以用Instance类来表示。Instance 也能够用于创建原先的java.util.Date对象。 

**Java代码**

1. Clock clock = Clock.systemDefaultZone();
2. **long** millis = clock.millis();
4. Instant instant = clock.instant();
5. Date legacyDate = Date.from(instant);   // legacy java.util.Date

**Timezones**   
  
时区类可以用一个ZoneId来表示。时区类的对象可以通过静态工厂方法方便地获取。时区类还定义了一个偏移量，用来在当前时刻或某时间与目标时区时间之间进行转换。 

**Java代码**

1. System.out.println(ZoneId.getAvailableZoneIds());
2. // prints all available timezone ids
4. ZoneId zone1 = ZoneId.of("Europe/Berlin");
5. ZoneId zone2 = ZoneId.of("Brazil/East");
6. System.out.println(zone1.getRules());
7. System.out.println(zone2.getRules());
9. // ZoneRules[currentStandardOffset=+01:00]
10. // ZoneRules[currentStandardOffset=-03:00]

**LocalTime**   
  
本地时间类表示一个没有指定时区的时间，例如，10 p.m.或者17：30:15，下面的例子会用上面的例子定义的时区创建两个本地时间对象。然后我们会比较两个时间，并计算它们之间的小时和分钟的不同。 

**Java代码**

1. LocalTime now1 = LocalTime.now(zone1);
2. LocalTime now2 = LocalTime.now(zone2);
4. System.out.println(now1.isBefore(now2));  // false
6. **long** hoursBetween = ChronoUnit.HOURS.between(now1, now2);
7. **long** minutesBetween = ChronoUnit.MINUTES.between(now1, now2);
9. System.out.println(hoursBetween);       // -3
10. System.out.println(minutesBetween);     // -239

LocalTime是由多个工厂方法组成，其目的是为了简化对时间对象实例的创建和操作，包括对时间字符串进行解析的操作。 

**Java代码**

1. LocalTime late = LocalTime.of(23, 59, 59);
2. System.out.println(late);       // 23:59:59
4. DateTimeFormatter germanFormatter =
5. DateTimeFormatter
6. .ofLocalizedTime(FormatStyle.SHORT)
7. .withLocale(Locale.GERMAN);
9. LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);
10. System.out.println(leetTime);   // 13:37

**LocalDate**   
  
本地时间表示了一个独一无二的时间，例如：2014-03-11。这个时间是不可变的，与LocalTime是同源的。下面的例子演示了如何通过加减日，月，年等指标来计算新的日期。记住，每一次操作都会返回一个新的时间对象。 

**Java代码**

1. LocalDate today = LocalDate.now();
2. LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
3. LocalDate yesterday = tomorrow.minusDays(2);
5. LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);
6. DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();
7. System.out.println(dayOfWeek);    // FRIDAY<span style="font-family: Georgia, 'Times New Roman', 'Bitstream Charter', Times, serif; font-size: 13px; line-height: 19px;">Parsing a LocalDate from a string is just as simple as parsing a LocalTime:</span>

解析字符串并形成LocalDate对象，这个操作和解析LocalTime一样简单。 

**Java代码**

1. DateTimeFormatter germanFormatter =
2. DateTimeFormatter
3. .ofLocalizedDate(FormatStyle.MEDIUM)
4. .withLocale(Locale.GERMAN);
6. LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter);
7. System.out.println(xmas);   // 2014-12-24

**LocalDateTime**   
  
LocalDateTime表示的是日期-时间。它将刚才介绍的日期对象和时间对象结合起来，形成了一个对象实例。LocalDateTime是不可变的，与LocalTime和LocalDate的工作原理相同。我们可以通过调用方法来获取日期时间对象中特定的数据域。 

**Java代码**

1. LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31, 23, 59, 59);
3. DayOfWeek dayOfWeek = sylvester.getDayOfWeek();
4. System.out.println(dayOfWeek);      // WEDNESDAY
6. Month month = sylvester.getMonth();
7. System.out.println(month);          // DECEMBER
9. **long** minuteOfDay = sylvester.getLong(ChronoField.MINUTE\_OF\_DAY);
10. System.out.println(minuteOfDay);    // 1439

如果再加上的时区信息，LocalDateTime能够被转换成Instance实例。Instance能够被转换成以前的java.util.Date对象。 

**Java代码**

1. Instant instant = sylvester
2. .atZone(ZoneId.systemDefault())
3. .toInstant();
5. Date legacyDate = Date.from(instant);
6. System.out.println(legacyDate);     // Wed Dec 31 23:59:59 CET 2014

格式化日期-时间对象就和格式化日期对象或者时间对象一样。除了使用预定义的格式以外，我们还可以创建自定义的格式化对象，然后匹配我们自定义的格式。 

**Java代码**

1. DateTimeFormatter formatter =
2. DateTimeFormatter
3. .ofPattern("MMM dd, yyyy - HH:mm");
5. LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13", formatter);
6. String string = formatter.format(parsed);
7. System.out.println(string);     // Nov 03, 2014 - 07:13

不同于java.text.NumberFormat，新的DateTimeFormatter类是不可变的，也是线程安全的。   
  
更多的细节，请看[这里](http://download.java.net/jdk8/docs/api/java/time/format/DateTimeFormatter.html) 

## Annotations [Top](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)

Java 8中的注解是可重复的。让我们直接深入看看例子，弄明白它是什么意思。   
  
首先，我们定义一个包装注解，它包括了一个实际注解的数组 

**Java代码**

1. **@interface** Hints {
2. Hint[] value();
3. }
5. @Repeatable(Hints.**class**)
6. **@interface** Hint {
7. String value();
8. }

只要在前面加上注解名：@Repeatable，Java 8 允许我们对同一类型使用多重注解：   
  
变体1：使用注解容器（老方法）： 

**Java代码**

1. @Hints({@Hint("hint1"), @Hint("hint2")})
2. **class** Person {}

变体2：使用可重复注解（新方法）： 

**Java代码**

1. @Hint("hint1")
2. @Hint("hint2")
3. **class** Person {}

使用变体2，Java编译器能够在内部自动对@Hint进行设置。这对于通过反射来读取注解信息来说，是非常重要的。 

**Java代码**

1. Hint hint = Person.**class**.getAnnotation(Hint.**class**);
2. System.out.println(hint);                   // null
4. Hints hints1 = Person.**class**.getAnnotation(Hints.**class**);
5. System.out.println(hints1.value().length);  // 2
7. Hint[] hints2 = Person.**class**.getAnnotationsByType(Hint.**class**);
8. System.out.println(hints2.length);          // 2

尽管我们绝对不会在Person类上声明@Hints注解，但是它的信息仍然可以通过getAnnotation(Hints.class)来读 取。并且，getAnnotationsByType方法会更方便，因为它赋予了所有@Hints注解标注的方法直接的访问权限。 

**Java代码**

1. @Target({ElementType.TYPE\_PARAMETER, ElementType.TYPE\_USE})
2. **@interface** MyAnnotation {}

## 总结 [Top](http://www.iteye.com/magazines/129-Java-8-Tutorial#top)

Java 8编程指南就到此告一段落。当然，还有很多内容需要进一步研究和说明。这就需要靠读者您来对JDK 8进行探究了，例如：Arrays.parallelSort, StampedLock和CompletableFuture等等 ———— 我这里只是举几个例子而已。   
  
我希望这个博文能够对您有所帮助，也希望您阅读愉快。完整的教程源代码放在了[GitHub](https://github.com/winterbe/java8-tutorial)上。您可以尽情地[fork](https://github.com/winterbe/java8-tutorial/fork)，并请通过[Twitter](https://twitter.com/benontherun)告诉我您的反馈。   
  
原文链接： [winterbe](http://winterbe.com/posts/2014/03/16/java-8-tutorial/) 翻译： [ImportNew.com](http://www.importnew.com/)- [黄小非](http://www.importnew.com/author/huangxiaofei)   
译文链接： <http://www.importnew.com/10360.html>

------------------------------------------------------------------------------------------------------------------------------------

加快OpenJDK的开发速度：继2014年3月份发布了Java 8之后，我们进入下一个两年的发布周期。 Java 9预计在2016年发布，并且已经公布了JEP(JDK改进提议)中的前期列表。同时，我们已经把一些新特性整理到了JSR(Java规范请求)，还有提 出了一些希望包括在新版本中的其他特性。

这些重要的特性都包括在Jigsaw项目中。显著的性能改善和期待已久的API包括：进程API更新，JSON将成为java.util的一部分，货币处理API对于想处在技术最前沿的你，可从这里获得Java 9的初期版本。

**被接受的特性**

1. Jigsaw 项目;模块化源码

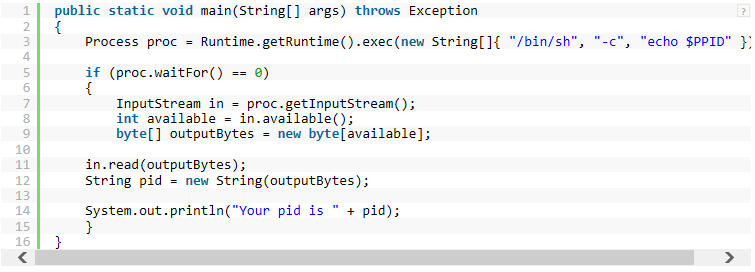
Jigsaw项目是为了模块化Java代码、将JRE分成可相互协作的组件，这也是Java 9 众多特色种的一个。JEP是迈向Jigsaw四步中的第一步，它不会改变JRE和JDK的真实结构。JEP是为了模块化JDK源代码，让编译系统能够模块 编译并在构建时检查模块边界。这个项目原本是随Java 8发布的，但由于推迟，所以将把它加到Java 9.

一旦它完成，它可能允许根据一个项目需求自定义组件从而减少rt.jar的大小。在JDK 7 和JDK 8的rt.jar包中有大约20,000个类，但有很多类在一些特定的环境里面并没有被用到(即使在Java 8的紧凑分布特性中已经包含了一部分解决方法也存在着类冗余)。这么做是为了能让Java能够容易应用到小型计算设备(比如网络设备)中，提高它的安全和 性能，同时也能让开发者更容易构建和维护这些类库。

**2. 简化进程API**

截止到目前，Java控制与管理系统进程的能力是有限的。举个例子，现在为了简便获取你程序的进程PID，你要么调用本地程序要么要自己使用一些变通方案。更多的是，每个（系统）平台需要有一个不同实现来确保你能获得正确的结果。

期望代码能获取Linux PIDS，现在是如下方式：



在Java 9中，可以变换成如下方式（支持所有的操作系统）：

http://dl2.iteye.com/upload/attachment/0101/1722/c4eb8366-25db-3ff1-b827-1a80b5230719.png

这次更新将会扩展Java与操作系统的交互能力：新增一些新的直接明了的方法去处理PIDs，进程名字和状态以及枚举多个JVM和进程以及更多事情。

**3. 轻量级 JSON API**

目前有多种处理JSON的Java工具，但JSON API 独到之处在于JSON API将作为Java语言的一部分，轻量并且运用Java 8的新特性。它将放在java.util包里一起发布(但在[JSR 353](https://jsonp.java.net/)里面的JSON是用第三方包或者[其他的方法](https://code.google.com/p/google-gson/)处理的).

**4. 钱和货币的API**

在Java 8引进了日期和时间的API之后, Java 9引入了新的货币API, 用以表示货币, 支持币种之间的转换和各种复杂运算. 关于这个项目的具体情况, 请访问<https://github.com/JavaMoney>,里面已经给出了使用说明和示例, 以下是几个重要的例子:



[更多关于 JSR 354的内容](https://jcp.org/en/jsr/detail?id=354)

**5. 改善锁争用机制**

锁争用是限制许多Java多线程应用性能的瓶颈. 新的机制在改善Java对象监视器的性能方面已经得到了多种基准(benchmark)的验证, 其中包括[Volano](http://www.volano.com/). 测试中通讯服务器开放了海量的进程来连接客户端, 其中有很多连接都申请同一个资源, 以此模拟重负荷日常应用.

通过诸如此类的压力测试我们可以估算JVM的极限吞吐量(每秒的消息数量). JEP在22种不同的测试中都得到了出色的成绩, 新的机制如果能在Java 9中得到应用的话, 应用程序的性能将会大大提升.

[关于JEP 143的更多内容](http://openjdk.java.net/jeps/143)

**6. 代码分段缓存**

Java 9的另一个性能提升来自于JIT(Just-in-time)编译器. 当某段代码被大量重复执行的时候, 虚拟机会把这段代码编译成机器码(native code)并储存在代码缓存里面, 进而通过访问缓存中不同分段的代码来提升编译器的效率.

和原来的单一缓存区域不同的是, 新的代码缓存根据代码自身的生命周期而分为三种:

* 永驻代码(JVM 内置 / 非方法代码)
* 短期代码(仅在某些条件下适用的配置性(profiled)代码)
* 长期代码(非配置性代码)

缓存分段会在各个方面提升程序的性能, 比如做垃圾回收扫描的时候可以直接跳过非方法代码(永驻代码), 从而提升效率.

[更多关于JEP 197的内容](http://openjdk.java.net/jeps/197)

**7. 智能Java编译, 第二阶段**

智能Java编译工具sjavac的第一阶段开始于[JEP 139](http://openjdk.java.net/jeps/139)这个项目, 用于在多核处理器上提升JDK的编译速度. 现在这个项目已经进入第二阶段([JEP 199](http://openjdk.java.net/jeps/199)), 目的是改进sjavac并让其成为取代目前JDK编译工具javac的Java默认的通用编译工具.

**其他值得期待的内容:**

**8. HTTP 2.0客户端**

HTTP 2.0标准虽然还没正式发布, 但是已经进入了最终审查阶段, 预计可以在Java 9发布之前审查完毕. [JEP 110](http://openjdk.java.net/jeps/110)将会重新定义并实现一个全新的Java HTTP客户端, 用来取代现在的HttpURLConnection, 同时也会实现HTTP 2.0和网络接口(原文websockets). 它现在还没被JEP正式认可但我们希望在Java 9中包含这一项目的内容.

官方的HTTP 2.0 RFC([Request for Comments](http://www.ietf.org/rfc.html), 官方技术讨论/会议记录等等的一系列文档记录)预订于2015年2月发布, 它是基于Google发布的SPDY(Speedy, 快速的)协议. 基于SPDY协议的网络相对于基于HTTP 1.1协议的网络有11.81%到47.7%之间的显著提速, 现在已经有浏览器实现了这个协议.

**9. Kulla计划: Java的REPL实现**

这个取名为Kulla的项目最近宣布将于2015年4月整合测试, 虽然已经不太有希望能赶上Java 9的发布, 但如果进度快的话或许刚好能赶上. 现在Java并没有来自官方的REPL(Read-Eval-Print-Loop)方式, 也就是说现在如果你想要跑几行Java代码做一个快速的测试, 你仍然需要把这几行代码封装在项目或者方法里面. 虽然在一些流行的IDE里面有Java REPL工具, 但它们并没有官方支持, 而Kulla项目或许就能成为Java官方发布的REPL解决方案.

[更多关于Kulla计划的内容](http://mail.openjdk.java.net/pipermail/announce/2014-August/000181.html)

**这些新功能出自何处？**

JEP和JSR并不是无中生有，下面就介绍一下Java发展的生态环境：

**小组** - 对特定技术内容, 比如安全、网络、Swing、HotSpot、有共同兴趣的组织和个人。

**项目** - 编写代码, 文档以及其他工作，至少由一个小组赞助和支持，比如最近的Lambda计划，Jigsaw计划和Sumatra计划。

**JDK改进提案(JEP)** - 每当需要有新的尝试的时候, JEP可以在JCP(Java Community Process)之前或者同时提出非正式的规范(specification)，被正是认可的JEP正式写进JDK的发展路线图并分配版本号。

**Java规范提案(JSR)** - 新特性的规范出现在这一个阶段，可以来自于小组 / 项目、JEP、 JCP成员或者Java社区(community)成员的提案，每个Java版本都由相应的JSR支持, Java 9暂时还没有。