Java5:

**1、泛型 Generics:**  
        引用泛型之后，允许指定集合里元素的类型，免去了强制类型转换，并且能在编译时刻进行类型检查的好处。

        Parameterized Type作为参数和返回值，Generic是vararg、annotation、enumeration、collection的基石。

        A、类型安全

           抛弃List、Map，使用List<T>、Map<K,V>给它们添加元素或者使用Iterator<T>遍历时，编译期就可以给你检查出类型错误

        B、方法参数和返回值加上了Type

        抛弃List、Map，使用List<T>、Map<K,V>

          C、不需要类型转换

          List<String> list=new ArrayList<String>();

       String str=list.get(i);

         D、类型通配符“?”

       假设一个打印List<T>中元素的方法printList,我们希望任何类型T的List<T>都可以被打印：

     代码：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

1. **public** **void** printList(List<?> list,PrintStream out)**throws** IOException{
2. **for**(Iterator<?> i=list.iterator();i.hasNext();){
3. System.out.println(i.next.toString());
4. }
6. }

     如果通配符？让我们的参数类型过于广泛，我们可以把List<?>、Iterator<?> 修改为

      List<? Extends Number>、Iterator<? Extends Number>限制一下它。

**2、枚举类型 Enumeration:**

**3、自动装箱拆箱（自动类型包装和解包）autoboxing & unboxing:**

  简单的说是类型自动转换。

   自动装包：基本类型自动转为包装类（int ——Integer）

   自动拆包：包装类自动转为基本类型（Integer——int）

**4、可变参数varargs(varargs number of arguments)**

  参数类型相同时，把重载函数合并到一起了。

  如：public void test(object... objs){

for(Object obj:objs){

System.out.println(obj);

}

}

**5、Annotations 它是[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/samjustin1/article/details/_blank)中的metadata（注释）**

     A、Tiger中预定义的三种标准annotation

           a 、Override

           指出某个method覆盖了superclass 的method当你要覆盖的方法名拼写错时编译不通过

           b、Deprecated

           指出某个method或element类型的使用是被阻止的，子类将不能覆盖该方法

           c、SupressWarnings

           关闭class、method、field、variable 初始化的编译期警告，比如：List没有使用 Generic,则@SuppressWarnings("unchecked")去掉编译期警告。

     B、自定义annotation

          public @interface Marked{}

     C、meta-annotation

         或者说annotation的annotation

         四种标准的meta-annotation全部定义在java.lang.annotaion包中:  
          a, Target  
           指定所定义的annotation可以用在哪些程序单元上  
           如果Target没有指定，则表示该annotation可以使用在任意程序单元上  
      代码

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

1. @Retention(RetentionPolicy.RUNTIME)
2. @Target({ ElementType.FIELD, ElementType.METHOD })
3. **public** **@interface** RejectEmpty {
4. /\*\* hint title used in error message \*/
5. String value() **default** "";
6. }
8. @Retention(RetentionPolicy.RUNTIME)
9. @Target( { ElementType.FIELD, ElementType.METHOD })
10. **public** **@interface** AcceptInt {
11. **int** min() **default** Integer.MIN\_VALUE;
12. **int** max() **default** Integer.MAX\_VALUE;
13. String hint() **default** "";
14. }

    b, Retention  
     指出Java编译期如何对待annotation  
      annotation可以被编译期丢掉，或者保留在编译过的class文件中  
        在annotation被保留时，它也指定是否会在JVM加载class时读取该annotation  
代码

**[html]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

1. @Retention(RetentionPolicy.SOURCE)  // Annotation会被编译期丢弃
2. public @interface TODO1 {}
3. @Retention(RetentionPolicy.CLASS)   // Annotation保留在class文件中，但会被JVM忽略
4. public @interface TODO2 {}
5. @Retention(RetentionPolicy.RUNTIME) // Annotation保留在class文件中且会被JVM读取
6. public @interface TODO3 {}

       c, Documented  
       指出被定义的annotation被视为所熟悉的程序单元的公开API之一  
        被@Documented标注的annotation会在javadoc中显示，这在annotation对它标注的元素被客户端使用时有影响时起作用  
       d, Inherited  
       该meta-annotation应用于目标为class的annotation类型上，被此annotattion标注的class会自动继承父类的annotation  
    D, Annotation的反射  
    我们发现java.lang.Class有许多与Annotation的反射相关的方法，如getAnnotations、isAnnotationpresent  
    我们可以利用Annotation反射来做许多事情，比如自定义Annotation来做Model对象验证  
代码

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

1. @Retention(RetentionPolicy.RUNTIME)
2. @Target({ ElementType.FIELD, ElementType.METHOD })
3. **public** **@interface** RejectEmpty {
4. /\*\* hint title used in error message \*/
5. String value() **default** "";
6. }
8. @Retention(RetentionPolicy.RUNTIME)
9. @Target( { ElementType.FIELD, ElementType.METHOD })
10. **public** **@interface** AcceptInt {
11. **int** min() **default** Integer.MIN\_VALUE;
12. **int** max() **default** Integer.MAX\_VALUE;
13. String hint() **default** "";
14. }

使用@RejectEmpty和@AcceptInt标注我们的Model的field，然后利用反射来做Model验证

**6、新的迭代语句（for(int n:numbers)）**

**7、静态导入（import static ）**

**8、新的格式化方法（java.util.Formatter）**

formatter.format("Remaining account balance: $%.2f", balance);

**9、新的线程模型和并发库Thread Framework**

HashMap的替代者[ConcurrentHashMap](http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ConcurrentHashMap.html" \t "http://blog.csdn.net/samjustin1/article/details/_blank)和ArrayList的替代者[CopyOnWriteArrayList](http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/CopyOnWriteArrayList.html" \t "http://blog.csdn.net/samjustin1/article/details/_blank)  
在大并发量读取时采用java.util.concurrent包里的一些类会让大家满意BlockingQueue、Callable、Executor、Semaphore...

Java6:

1、引入了一个支持脚本引擎的新框架

2、UI的增强

3、对WebService支持的增强（JAX-WS2.0和JAXB2.0）

4、一系列新的安全相关的增强

5、JDBC4.0

6、Compiler API

7、通用的Annotations支持

Java7:

1，switch中可以使用字串了

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

1. String s = "test";
2. **switch** (s) {
3. **case** "test" :
4. System.out.println("test");
5. **case** "test1" :
6. System.out.println("test1");
7. **break** ;
8. **default** :
9. System.out.println("break");
10. **break** ;
11. }

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

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

1. File System.getJavaIoTempDir() // IO临时文件夹
2. File System.getJavaHomeDir() // JRE的安装目录
3. File System.getUserHomeDir() // 当前用户目录
4. 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

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

1. Boolean Booleans.negate(Boolean booleanObj)
2. True => False , False => True, Null => Null
3. **boolean** Booleans.and(**boolean**[] array)
4. **boolean** Booleans.or(**boolean**[] array)
5. **boolean** Booleans.xor(**boolean**[] array)
6. **boolean** Booleans.and(Boolean[] array)
7. **boolean** Booleans.or(Boolean[] array)
8. **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

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

1. **int** Math.safeToInt(**long** value)
2. **int** Math.safeNegate(**int** value)
3. **long** Math.safeSubtract(**long** value1, **int** value2)
4. **long** Math.safeSubtract(**long** value1, **long** value2)
5. **int** Math.safeMultiply(**int** value1, **int** value2)
6. **long** Math.safeMultiply(**long** value1, **int** value2)
7. **long** Math.safeMultiply(**long** value1, **long** value2)
8. **long** Math.safeNegate(**long** value)
9. **int** Math.safeAdd(**int** value1, **int** value2)
10. **long** Math.safeAdd(**long** value1, **int** value2)
11. **long** Math.safeAdd(**long** value1, **long** value2)
12. **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};

# JAVA8：

java8的新特新逐一列出，并将使用简单的代码示例来指导你如何使用默认接口方法，lambda表达式，方法引用以及多重Annotation，之后你将会学到最新的API上的改进，比如流，函数式接口，Map以及全新的日期API

**一、接口的默认方法**

**[Java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/samjustin1/article/details/_blank)**8允许我们给接口添加一个非抽象的方法实现，只需要使用 default关键字即可，这个特征又叫做扩展方法，示例如下：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

Formula接口在拥有calculate方法之外同时还定义了sqrt方法，实现了Formula接口的子类只需要实现一个calculate方法，默认方法sqrt将在子类上可以直接使用。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

文中的formula被实现为一个匿名类的实例，该代码非常容易理解，6行代码实现了计算 sqrt(a \* 100)。在下一节中，我们将会看到实现单方法接口的更简单的做法。译者注： 在Java中只有单继承，如果要让一个类赋予新的特性，通常是使用接口来实现，在C++中支持多继承，允许一个子类同时具有多个父类的接口与功能，在其他语言中，让一个类同时具有其他的可复用代码的方法叫做mixin。新的Java 8 的这个特新在编译器实现的角度上来说更加接近Scala的trait。 在C#中也有名为扩展方法的概念，允许给已存在的类型扩展方法，和Java 8的这个在语义上有差别。

**二、Lambda 表达式**

首先看看在老版本的Java中是如何排列字符串的：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

只需要给静态方法 Collections.sort 传入一个List对象以及一个比较器来按指定顺序排列。通常做法都是创建一个匿名的比较器对象然后将其传递给sort方法。在Java 8 中你就没必要使用这种传统的匿名对象的方式了，Java 8提供了更简洁的语法，lambda表达式：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

看到了吧，代码变得更段且更具有可读性，但是实际上还可以写得更短：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

看到了吧，代码变得更段且更具有可读性，但是实际上还可以写得更短：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

对于函数体只有一行代码的，你可以去掉大括号{}以及return关键字，但是你还可以写得更短点：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

Java编译器可以自动推导出参数类型，所以你可以不用再写一次类型。接下来我们看看lambda表达式还能作出什么更方便的东西来：

**三、函数式接口**

Lambda表达式是如何在java的类型系统中表示的呢？每一个lambda表达式都对应一个类型，通常是接口类型。而“函数式接口”是指仅仅只包含一个抽象方法的接口，每一个该类型的lambda表达式都会被匹配到这个抽象方法。因为 默认方法 不算抽象方法，所以你也可以给你的函数式接口添加默认方法。

我们可以将lambda表达式当作任意只包含一个抽象方法的接口类型，确保你的接口一定达到这个要求，你只需要给你的接口添加 @FunctionalInterface 注解，编译器如果发现你标注了这个注解的接口有多于一个抽象方法的时候会报错的。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

需要注意如果@FunctionalInterface如果没有指定，上面的代码也是对的。

译者注 将lambda表达式映射到一个单方法的接口上，这种做法在**[java](http://lib.csdn.net/base/java" \o "Java 知识库" \t "http://blog.csdn.net/samjustin1/article/details/_blank)**8之前就有别的语言实现，比如Rhino **[JavaScript](http://lib.csdn.net/base/javascript" \o "JavaScript知识库" \t "http://blog.csdn.net/samjustin1/article/details/_blank)**解释器，如果一个函数参数接收一个单方法的接口而你传递的是一个function，Rhino 解释器会自动做一个单接口的实例到function的适配器，典型的应用场景有 org.w3c.dom.events.EventTarget 的addEventListener 第二个参数 EventListener。

四、方法与构造函数引用

前一节中的代码还可以通过静态方法引用来表示：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

Java 8 允许你使用 :: 关键字来传递方法或者构造函数引用，上面的代码展示了如何引用一个静态方法，我们也可以引用一个对象的方法：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

1. converter = something::startsWith;
2. String converted = converter.convert("Java");
3. System.out.println(converted);    // "J"

接下来看看构造函数是如何使用::关键字来引用的，首先我们定义一个包含多个构造函数的简单类：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

接下来我们指定一个用来创建Person对象的对象工厂接口：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

这里我们使用构造函数引用来将他们关联起来，而不是实现一个完整的工厂：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

我们只需要使用 Person::new 来获取Person类构造函数的引用，Java编译器会自动根据PersonFactory.create方法的签名来选择合适的构造函数。

**五、Lambda 作用域**  
在lambda表达式中访问外层作用域和老版本的匿名对象中的方式很相似。你可以直接访问标记了final的外层局部变量，或者实例的字段以及静态变量。

**六、访问局部变量**

我们可以直接在lambda表达式中访问外层的局部变量：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

但是和匿名对象不同的是，这里的变量num可以不用声明为final，该代码同样正确：

**[html]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

不过这里的num必须不可被后面的代码修改（即隐性的具有final的语义），例如下面的就无法编译：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

在lambda表达式中试图修改num同样是不允许的。  
七、访问对象字段与静态变量  
和本地变量不同的是，lambda内部对于实例的字段以及静态变量是即可读又可写。该行为和匿名对象是一致的：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

八、访问接口的默认方法  
还记得第一节中的formula例子么，接口Formula定义了一个默认方法sqrt可以直接被formula的实例包括匿名对象访问到，但是在lambda表达式中这个是不行的。  
Lambda表达式中是无法访问到默认方法的，以下代码将无法编译：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

1. Formula formula = (a) -> sqrt( a \* 100);
2. Built-in Functional Interfaces

JDK 1.8 API包含了很多内建的函数式接口，在老Java中常用到的比如Comparator或者Runnable接口，这些接口都增加了@FunctionalInterface注解以便能用在lambda上。  
Java 8 API同样还提供了很多全新的函数式接口来让工作更加方便，有一些接口是来自Google Guava库里的，即便你对这些很熟悉了，还是有必要看看这些是如何扩展到lambda上使用的。

**Predicate接口**

Predicate 接口只有一个参数，返回boolean类型。该接口包含多种默认方法来将Predicate组合成其他复杂的逻辑（比如：与，或，非）：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**Function 接口**  
Function 接口有一个参数并且返回一个结果，并附带了一些可以和其他函数组合的默认方法（compose, andThen）：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**Supplier 接口**  
Supplier 接口返回一个任意范型的值，和Function接口不同的是该接口没有任何参数

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**Consumer 接口**  
Consumer 接口表示执行在单个参数上的操作。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**Comparator 接口**

Comparator 是老Java中的经典接口， Java 8在此之上添加了多种默认方法：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**Optional 接口**

Optional 不是函数是接口，这是个用来防止NullPointerException异常的辅助类型，这是下一届中将要用到的重要概念，现在先简单的看看这个接口能干什么：

Optional 被定义为一个简单的容器，其值可能是null或者不是null。在Java 8之前一般某个函数应该返回非空对象但是偶尔却可能返回了null，而在Java 8中，不推荐你返回null而是返回Optional。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**Stream 接口**

java.util.Stream 表示能应用在一组元素上一次执行的操作序列。Stream 操作分为中间操作或者最终操作两种，最终操作返回一特定类型的计算结果，而中间操作返回Stream本身，这样你就可以将多个操作依次串起来。Stream 的创建需要指定一个数据源，比如 java.util.Collection的子类，List或者Set， Map不支持。Stream的操作可以串行执行或者并行执行。

首先看看Stream是怎么用，首先创建实例代码的用到的数据List：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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扩展了集合类，可以通过 Collection.stream() 或者 Collection.parallelStream() 来创建一个Stream。下面几节将详细解释常用的Stream操作：

**Filter 过滤**  
过滤通过一个predicate接口来过滤并只保留符合条件的元素，该操作属于中间操作，所以我们可以在过滤后的结果来应用其他Stream操作（比如forEach）。forEach需要一个函数来对过滤后的元素依次执行。forEach是一个最终操作，所以我们不能在forEach之后来执行其他Stream操作。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**Sort 排序**  
排序是一个中间操作，返回的是排序好后的Stream。如果你不指定一个自定义的Comparator则会使用默认排序。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

需要注意的是，排序只创建了一个排列好后的Stream，而不会影响原有的数据源，排序之后原数据stringCollection是不会被修改的：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**Map 映射**  
中间操作map会将元素根据指定的Function接口来依次将元素转成另外的对象，下面的示例展示了将字符串转换为大写字符串。你也可以通过map来讲对象转换成其他类型，map返回的Stream类型是根据你map传递进去的函数的返回值决定的。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**Match 匹配**  
Stream提供了多种匹配操作，允许检测指定的Predicate是否匹配整个Stream。所有的匹配操作都是最终操作，并返回一个boolean类型的值。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**Count 计数**

计数是一个最终操作，返回Stream中元素的个数，返回值类型是long。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**Reduce 规约**  
这是一个最终操作，允许通过指定的函数来讲stream中的多个元素规约为一个元素，规越后的结果是通过Optional接口表示的：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**并行Streams**

前面提到过Stream有串行和并行两种，串行Stream上的操作是在一个线程中依次完成，而并行Stream则是在多个线程上同时执行。

下面的例子展示了是如何通过并行Stream来提升性能：

首先我们创建一个没有重复元素的大表：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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. }

然后我们计算一下排序这个Stream要耗时多久，

串行排序：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

// 串行耗时: 899 ms  
并行排序：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

// 并行排序耗时: 472 ms  
上面两个代码几乎是一样的，但是并行版的快了50%之多，唯一需要做的改动就是将stream()改为parallelStream()。  
**Map**  
前面提到过，Map类型不支持stream，不过Map提供了一些新的有用的方法来处理一些日常任务。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

1. Map<Integer, String> map = **new** HashMap<>();
2. **for** (**int** i = 0; i < 10; i++) {
3. map.putIfAbsent(i, "val" + i);
4. }

map.forEach((id, val) -> System.out.println(val));  
以上代码很容易理解， putIfAbsent 不需要我们做额外的存在性检查，而forEach则接收一个Consumer接口来对map里的每一个键值对进行操作。

下面的例子展示了map上的其他有用的函数：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

接下来展示如何在Map里删除一个键值全都匹配的项：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

另外一个有用的方法：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

对Map的元素做合并也变得很容易了：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

Merge做的事情是如果键名不存在则插入，否则则对原键对应的值做合并操作并重新插入到map中。  
九、Date API  
Java 8 在包java.time下包含了一组全新的时间日期API。新的日期API和开源的Joda-Time库差不多，但又不完全一样，下面的例子展示了这组新API里最重要的一些部分：

**Clock 时钟**

Clock类提供了访问当前日期和时间的方法，Clock是时区敏感的，可以用来取代 System.currentTimeMillis() 来获取当前的微秒数。某一个特定的时间点也可以使用Instant类来表示，Instant类也可以用来创建老的java.util.Date对象。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**Timezones 时区**

在新API中时区使用ZoneId来表示。时区可以很方便的使用静态方法of来获取到。 时区定义了到UTS时间的时间差，在Instant时间点对象到本地日期对象之间转换的时候是极其重要的。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**LocalTime 本地时间**

LocalTime 定义了一个没有时区信息的时间，例如 晚上10点，或者 17:30:15。下面的例子使用前面代码创建的时区创建了两个本地时间。之后比较时间并以小时和分钟为单位计算两个时间的时间差：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

LocalTime 提供了多种工厂方法来简化对象的创建，包括解析时间字符串。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**LocalDate 本地日期**

LocalDate 表示了一个确切的日期，比如 2014-03-11。该对象值是不可变的，用起来和LocalTime基本一致。下面的例子展示了如何给Date对象加减天/月/年。另外要注意的是这些对象是不可变的，操作返回的总是一个新实例。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

1. LocalDate today = LocalDate.now();
2. LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
3. LocalDate yesterday = tomorrow.minusDays(2);
4. LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);
5. DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();

System.out.println(dayOfWeek);    // FRIDAY  
从字符串解析一个LocalDate类型和解析LocalTime一样简单：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

**LocalDateTime 本地日期时间**  
LocalDateTime 同时表示了时间和日期，相当于前两节内容合并到一个对象上了。LocalDateTime和LocalTime还有LocalDate一样，都是不可变的。LocalDateTime提供了一些能访问具体字段的方法。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

只要附加上时区信息，就可以将其转换为一个时间点Instant对象，Instant时间点对象可以很容易的转换为老式的java.util.Date。

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

格式化LocalDateTime和格式化时间和日期一样的，除了使用预定义好的格式外，我们也可以自己定义格式：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

和java.text.NumberFormat不一样的是新版的DateTimeFormatter是不可变的，所以它是线程安全的。  
关于时间日期格式的详细信息：http://download.java.net/jdk8/docs/api/java/time/format/DateTimeFormatter.html

**十、Annotation 注解**

在Java 8中支持多重注解了，先看个例子来理解一下是什么意思。  
首先定义一个包装类Hints注解用来放置一组具体的Hint注解：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

Java 8允许我们把同一个类型的注解使用多次，只需要给该注解标注一下@Repeatable即可。例 1: 使用包装类当容器来存多个注解（老方法）

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

例 2：使用多重注解（新方法）

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

第二个例子里java编译器会隐性的帮你定义好@Hints注解，了解这一点有助于你用反射来获取这些信息：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

即便我们没有在Person类上定义@Hints注解，我们还是可以通过 getAnnotation(Hints.class) 来获取 @Hints注解，更加方便的方法是使用 getAnnotationsByType 可以直接获取到所有的@Hint注解。  
另外Java 8的注解还增加到两种新的target上了：

**[java]** [view plain](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "view plain" \t "http://blog.csdn.net/samjustin1/article/details/_blank) [copy](http://blog.csdn.net/qq_30641447/article/details/49853067" \o "copy" \t "http://blog.csdn.net/samjustin1/article/details/_blank)

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

关于Java 8的新特性就写到这了，肯定还有更多的特性等待发掘。JDK 1.8里还有很多很有用的东西，比如Arrays.parallelSort, StampedLock和CompletableFuture等等。