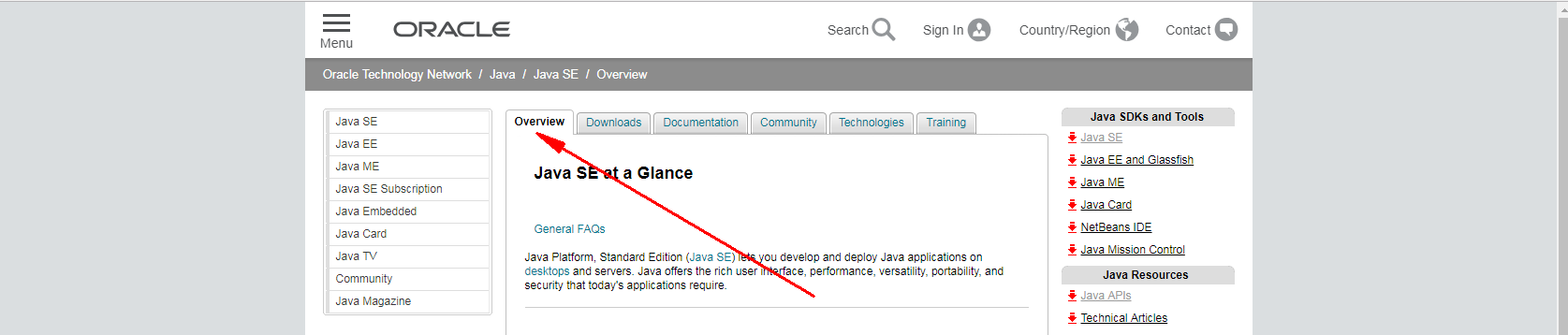
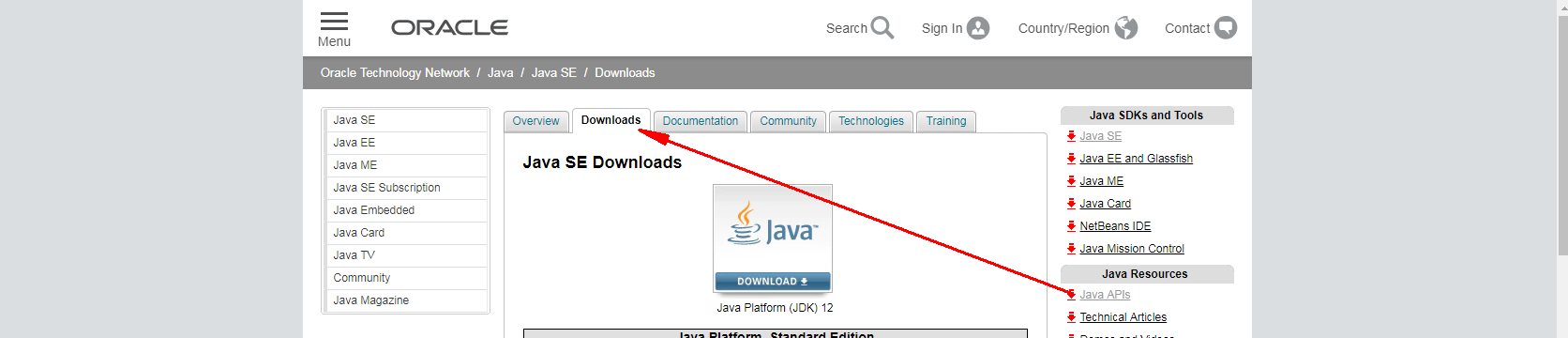
# API下载

javaSE：<https://www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html>

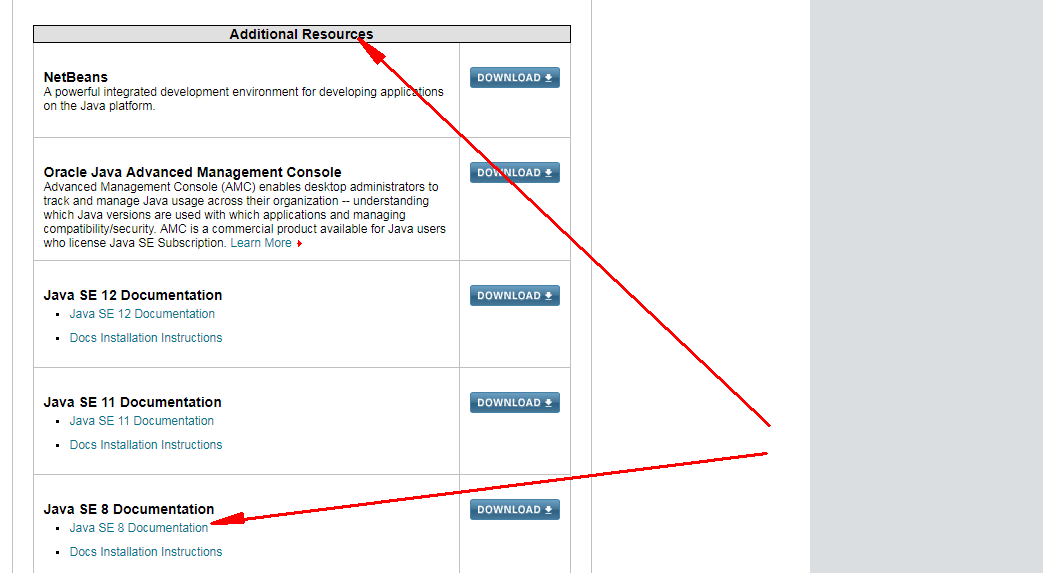
第一步：



第二步：



第三步：



javaEE：https://www.oracle.com/technetwork/java/javaee/documentation/apis-139520.html

OpenJDK源码下载：http://hg.openjdk.java.net

# 运算符

## 逻辑运算

**位运算符：**

  &（位逻辑与）：将操作数转换成二进制数，然后将两个二进制操作数对象从低位到高位对齐，每位求与。若操作数对象同一位都为1，则结果对应位为1，若操作数对象同一位为0。

 | （位逻辑或）：将操作数转换成二进制数 ，然后将两个二进制操作数对象从低位到高位对齐，每位求或。若操作数对象同一位都为0，则结果对应为0，否则结果对应为1。

^（位逻辑异或） ：将操作数转换成二进制数 ，然后将两个二进制操作数对象从低位到高位对齐，每位求异或。若操作数对象同一位不同为1，则结果对应位为1，否则结果结果中对应位为0。

 ~ （取反运算符）：将操作数转换成二进制数 ，然后将各位二进制由0变成1，由1变成0。

**移位运算符：**

  <<（左移）：左移是将一个二进制操作数对象按指定的移动位数向左移，左边溢出的位数被丢弃，右边的空位用0补充。右移相当于乘以2的幂次。

>>（有符号右移）：有符号右移是将一个二进制操作数对象按指定的移动位数向右移，右边溢出的位数被丢弃，左边的空位用符号位填充，右移相当于除以2的幂次。

>>>（无符号右移）：右移是将一个二进制操作数对象按指定的移动位数向右移，右边溢出的位数被丢弃，左边的空位用0补充。

**&，|为啥没有短路：**

## 补码

java中负数在内存存储的方式，是以补码存放。

**为什么引用补码：**

计算机中没有减法器，减法运算只能通过转化加法来进行。引用补码的好处在于相同进制的减法运算中，减去一个数等于加加上这个数字的补码。

两个用补码表示的数相加时，如果最高位（符号位）有进位，则进位被舍弃。

计算机中是有规定的，正数的最高位为0，负数的最高位为1。

用负数表示0为：11111111，最高位1符号位，其余7位0000000取反1111111。

用正数表示0为：00000000，最高位0符号位，其余7位0000000不变。

[+0]原码=0000 0000， [-0]原码=1000 0000

[+0]反码=0000 0000， [-0]反码=1111 1111

[+0]补码=0000 0000， [-0]补码=0000 0000

+0和-0的补码是一样的。即 0的补码只有一种表示

八位二进制：

原码 -127~127 -128呢？？？不是说-128有原码 是1000 0000吗？？

反码 -127~127

补码 -128~127

-128原码=1000 0000 反码：1111 1111 你会发现-0和-128的反码相同，所以-128没有反码

科学家只是将原来表示-127的1000000用来表示-128，原来表示负数0的11111111用来表示-1。

-128～127在计算机存储的是10000000~01111111就是这样来的。

-128的补码为啥是1000 0000：

（1）从补码的意义上去理解

因为：256-128=256+(-128)的补码 --机器中只有加法。减法会变成补码的加法。而256-128=128 所以 256+（-128）的补码=128 所以 （-128）的补码=256-128=128 ，数学上， 128=1000 0000 ，故规定-128的补码为 1000 0000 。注意：只是规定而已

（2） 原码 补码 值

0111 1111 0111 1111 127

0111 1110 0111 1110 126

0111 1101 0111 1101 125

.... .... .... ....

0000 0001 0000 0001 1

0000 0000 0000 0000 0

1000 0000 0000 0000 0

1000 0001 1111 1111 -1

1000 0010 1111 1110 -2

.... .... .... ....

1111 1101 1000 0011 -125

1111 1110 1000 0010 -126

1111 1111 1000 0001 -127

无法表示 1000 0000 -128

# 集合框架

## Collection接口

### List接口

#### ArrayList

#### LinkedList

### Map接口

#### HashMap

##### 源码分析

在jdk1.7之前HashMap主要有“数组+链表”组成，在jdk1.8后HashMap进行了很大的优化，主要有“数组+链表+红黑树”组成。

Jdk1.8:

构造方法没有对HashMap进行初始化，而是在第一次put时，对HashMap进行初始化。

**成员变量：**

|  |
| --- |
| **static** **final** **int** ***DEFAULT\_INITIAL\_CAPACITY*** = 1 << 4; //默认初始化容量  **static** **final** **int** ***MAXIMUM\_CAPACITY*** = 1 << 30;//最大容量  **static** **final** **float** ***DEFAULT\_LOAD\_FACTOR*** = 0.75f;//加载因子  **static** **final** **int** ***TREEIFY\_THRESHOLD*** = 8;// 链表转化成红黑树的阈值  **static** **final** **int** ***UNTREEIFY\_THRESHOLD*** = 6;//红黑树转化成链表的阈值  **static** **final** **int** ***MIN\_TREEIFY\_CAPACITY*** = 64;//  **transient** Node<K,V>[] table;//存放元素的Node数组  **transient** **int** modCount; //内部维护的变量，用于记录修改次数  **int** threshold; //下一次resize的大小 |

**put方法：**

|  |
| --- |
| /\*  \* hash 要插入元素的hash值  \* key 要插入元素的key  \* value 要插入元素的value  \* onlyIfAbsent 如果为true，对已经存在的key，不改变已经存在的value  \* evict 如果为false，则hash表为创建模式  /  **final V putVal**(int hash, K key, V value, boolean onlyIfAbsent,boolean evict) {  Node<K,V>[] tab; Node<K,V> p; int n, i;  if ((tab = table) == null || (n = tab.length) == 0)  n = (tab = resize()).length; //初始化  if ((p = tab[i = (n - 1) & hash]) == null)  tab[i] = newNode(hash, key, value, null); //经过计算后的位置为null，则直接插入  else { //经过计算后的位置不为空  Node<K,V> e; K k;  if (p.hash == hash &&((k = p.key) == key || (key != null && key.equals(k))))  e = p; //插入位置元素的key与要插入元素的key相同，则直接替换  else if (p instanceof TreeNode)  e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value); //如果要插入的元素是红黑树，则直接插入到红黑树中  else { //插入的元素不是红黑树，则插入到链表中  for (int binCount = 0; ; ++binCount) {  if ((e = p.next) == null) { //遍历链表  p.next = newNode(hash, key, value, null);  if (binCount >= TREEIFY\_THRESHOLD - 1) // 如果链表的长度大于7，则直接转换成红黑树  treeifyBin(tab, hash);  break;  }  if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))  break; //遍历的过程中发现相等的key，则跳出循环  p = e;  }  }  if (e != null) { // 替换value  V oldValue = e.value;  if (!onlyIfAbsent || oldValue == null)  e.value = value;  afterNodeAccess(e); //回调  return oldValue; //返回旧value，可以为null  }  }  ++modCount;  if (++size > threshold)  resize();  afterNodeInsertion(evict);  return null;  } |

**resize方法：**

Jdk1.7:

#### TreeMap

### Set接口

#### HashSet

#### TreeSet

### Comparator接口

### Comparable接口

### 策略模式

# JDK1.5

## 泛型（Generics）

Java泛型是JDK5中引入的一个新特性，泛型提供了编译时类型安全检测机制，该机制允许开发者在编译时检测非法的类型。

泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。

### 泛型的通配符

我们在定义泛型类，泛型方法，泛型接口的时候经常会碰到很多不同的通配符，不如T，E，K，V等等。

本质上这些通配符没啥区别，只不过是编码时的一种约定俗成的东西。

？：表示不确定的java类型。

T(type)：表示具体的一个java类型。

K V(key value)：分别代表java键值中的key value。

E(element)：代表Element。

#### 无限制通配符

**public** **class** Demo

{

**public** **static** **void** test(List<? **extends** Animal> animals)

{

animals.stream().forEach(i->System.***out***.println(i));

}

**public** **static** **void** test2(List<Animal> animals)

{

animals.stream().forEach(i->System.***out***.println(i));

}

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

{

Animal a1 = **new** Animal();

Animal a2 = **new** Animal();

List<Animal> animals = Arrays.*asList*(a1,a2);

*test*(animals);

*test2*(animals);

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

Dog d1 = **new** Dog();

Dog d2 = **new** Dog();

List<Dog> dogs = Arrays.*asList*(d1,d2);

*test*(dogs);

*test2*(dogs);

}

}

**class** Animal{}

**class** Dog **extends** Animal{}

在编译器编译器就我们报错误test2(dogs)。

像 test方法中，限定了上届，但是不关心具体类型是什么，所以对于传入的 Animal 的所有子类都可以支持，并且不会报错。而 test1 就不行对于不确定或者不关心实际要操作的类型，可以使用无限制通配符（尖括号里一个问号，即 <?> ），表示可以持有任何类型

#### 上界通配符<? extends E>

上界通配符使用extends关键字声明，表示参数类型可能是所指定的类型E，也可能是所指定类型的子类。

好处：1.如果传入的类型不是E类型或者不是E的子类，则编译器就报错

2.泛型中可以使用E的方法，要不然还需要强转成E的类型。

**public** **class** Demo

{

**public** **static** **void** test(List<? **extends** Animal> animals)

{

animals.stream().forEach(i->{

i.run();

});

}

**public** **static** **void** test2(List<Dog> dogs)

{

dogs.stream().forEach(i->{

Animal animal = i;

i.run();

});

}

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

{

Animal a1 = **new** Animal();

Animal a2 = **new** Animal();

List<Animal> animals = Arrays.*asList*(a1,a2);

Dog d1 = **new** Dog();

Dog d2 = **new** Dog();

List<Dog> dogs = Arrays.*asList*(d1,d2);

*test*(animals);

*test2*(dogs);

}

}

**class** Animal

{

**public** **void** run()

{

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

}

}

**class** Dog **extends** Animal{}

在test方法中，我们可以直接调用父类的方法，不需要类型

#### 下界通配符<? Super E>

下界通配符使用super关键字，表示参数化的类型可以是E，也可以是E的父类，直到Object。

#### ？和T的区别

1.类型参数可以多重限定而通配符不可以

**public** **class** Demo2

{

**public** **static**<T **extends** A & B> **void** test(T t)

{

t.test();

}

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

{

A a = **new** AA();

B b = **new** BB();

CC c = **new** CC();

*test*(c);

}

}

**interface** A{**void** test();}

**class** AA **implements** A

{

**public** **void** test()

{

System.***out***.println("AA class");

}

}

**interface** B{**void** test();}

**class** BB **implements** B

{

**public** **void** test()

{

System.***out***.println("BB class");

}

}

**class** CC **implements** A,B

{

**public** **void** test()

{

System.***out***.println("implements A And B interface");

}

}

使用 & 符号设定多重边界（Multi Bounds)，指定泛型类型 T 必须是A和B的共有子类型，此时变量 t 就具有了所有限定的方法和属性。对于通配符来说，因为它不是一个确定的类型，所以不能进行多重限定。

2.通配符可以使用超类限定而类型参数不行。

类型参数T只具有一种类型限定方式：T extends A

通配符？可以进行两种限定方式：？ extends A 或 ? super A

## 增强的for循环（Enhanced For Loop）

## 自动装箱/自动拆箱（Autoboxing/Unboxing）

## 可变参数（Var Agus）

## 内省

内省时java对Bean类属性，事件的一种缺省处理方法，例如类A中有属性name，那我们可以通过getName,setName来得到其值或者设置新的值。通过setXXX和getXXX方法来访问name属性，这就是默认的规则。Java中提供了一套API用来访问某个属性的setXXX和getXXX方法，通过这些API可以是你不需要了解这个规则，这些API存放在javaj.beans中。

## Enum(Enumeration)

当使用“enum”定义枚举类型时，实质上是定义出来的类型继承自java.lang.Enum类型，而每个枚举类型的成员就是一个枚举类型的一个实例（Instance），换句话说，当我们定义一个枚举类型时，在jvm编译时刻就能确定该枚举类型有几个实例，分别是什么，在运行期间我们无法再使用该枚举类型创建新的实例。每一个实例都被预设为final，所以无法改变它们，每一个实例都是static的，所以我们可以通过类型名称直接使用他们，最重要的是它们都是公开的（public）

|  |
| --- |
| **public** **enum** Color  {  ***Yello***(1),***Red***(2),***Purple***(3),***Green***(4) ;  **public** **int** color ;  Color(**int** colore){  **this**.color = colore ;  }  **public** **static** **void** main(String[] args) {  System.***out***.println(Color.***Red***);  }  } |

## 静态导入（Static Import）

静态导入只能导入静态方法和成员变量。

Import static packagepath.classname.methodname

Import static packagepath.classname.fieldname

## 注解（Annotation）

Annotation并不会直接影响代码的语义，但是它能够工作的方式被看作类似程序的工具或者类库，它会反过来对正在运行的程序语义有所影响。Annotation可以从源文件，class文件或者在运行时以反射的多种方式读取。

当我们自定义Annotation类型时，实际上是自动继承java.lang.annotation.Annotation接口，有编译器自动完成其他的细节，在定义Annotation注释时，不能继承其他接口或Annotation类型。

当我们定义一个接口，并且该接口继承了Annotation接口，那么我们定义的接口仍然是一个接口，而不是注解。

Jdk内置的注解：[1.@Override](mailto:1.@Override)：重写

[2.@Deprecated](mailto:2.@Deprecated)：不建议使用的

[3.@SuppressWarning({“unchecked”,”deprecated”})](mailto:3.@SuppressWarning(%7b)：抑制警告

[4.@Retention](mailto:4.@Retention):该注解可以在自定义注解时，指示编译器该如何对待你的自定义注解，默认上编译器会将Annotaion信息留在.class文件中，不被编译器读取，而仅用于编译程序和工具程序运行时提供信息。通常下该注解与枚举 RetentionPolicy结合使用，（1）RetentionPolicy.CLASS:可以保存到.class文件中，但不能被jvm读取。（2）RetentionPolicy.RUNTIME:可以保存到.class文件中，还能被jvm读取。（3）RetentionPolicy.SOURCE:注解不会被编译到.class文件内。

[5.@Target](mailto:5.@Target)：定义注解使用的时机，其传递的值类型为ElementType类型。（1）ElemententType.FIELD：只能修饰成员变量。（2）ElementType.METHOD：只能修饰方法。（3）ElementType.CONSTRUCTOR：只能修饰构造方法。等等…

[6.@Inherited](mailto:6.@Inherited)：子类是否继承父类，默认下父类的Annotation不会继承子类的Annotation。

特点：1.当注解的属相名为value是，则对其赋值时，可以不指定属性的名称直接赋值，而其他属性不行。

2.注解的属相可以赋予默认值，default 默认值。

案例一：

1.switch语句可以使用字符串。

# 反射（Reflection）

Java反射提供了如下功能：

# 代理模式

## 静态代理

作用：为其他对象提供一种代理以控制这个对象的访问。

角色：1.抽象角色：声明真实角色和代理角色的共同接口

2.代理角色：代理对象角色内部含有对真实角色的引用，从而可以操作真实对象，同事代理对象提供与真实对象相同的接口以便在任意时刻都可以代替真实对象，同事，代理对象可以在操作真是对象时，附加其他操作，相当于对真实对象进行封装。

3.真实角色：代理角色所代表的真实对象，是我们最终引用的对象。

|  |
| --- |
| /\*\*  \* 抽象角色  \*/  **public** **abstract** **class** Subject {  **public** **abstract** **void** request() ;  } |
| /\*\*  \*真实角色  \*/  **public** **class** RealSubject **extends** Subject{  @Override  **public** **void** request() {  System.***out***.println("from real subject");  }  } |
| /\*\*  \* 代理角色  \*/  **public** **class** ProxySubject **extends** Subject{  **private** RealSubject rs ;  @Override  **public** **void** request() {  preRequest(); //在真实角色操作前附加其他操作  **if**(rs == **null**) {  rs = **new** RealSubject();  }  rs.request();  postRequest();//在真实角色操作后附加其他操作  }  **private** **void** preRequest(){  System.***out***.println("pre request");  }  **private** **void** postRequest(){  System.***out***.println("post request");  }  } |
| **public** **class** Client  {  **public** **static** **void** main(String[] args) {  Subject subject = **new** ProxySubject() ;  subject.request();  }  } |

缺点：采用静态代理模式，则真实角色必须事先存在，并将其作为代理对象的内部属性。但是在实际使用时，一个真实角色必须对应一个代理角色，如果大量使用会导致类的极速的膨胀，此外，如果不知道真实角色，该如何代理呢？

## 动态代理

Java的动态代理类位于java.lang.reflect包下，一般主要涉及到连个类：

**--InvocationHandler**

该类是一个接口，该接口是由一个代理实例的代理处理器所实现，每一个代理实例都有一个与之关联的代理处理器，当调用代理实例的一个方式时，该方法会被编码，并转发给代理处理器的invoke方法。该类只有一个invoke方法。

public Object invoke(Object proxy ,Method method ,Object[] args)

proxy:包含代理方法的代理实例。

method：对用代理实例接口上的方法实例，

args：调用代理方法的参数，以数组的方式传递。

**--Proxy**

Proxy类提供了静态方法用于创建动态代理类，也是所有代理类的父类。

Proxy类中的newProxyInstance(ClassLoader classloader,Class<?>[] interfaces ,InvocationHandler hander)方法用于返回指定接口的代理类的实例，该方法可以将方法的调用指派到指定的方法处理程序。

动态类是运行时生成的class，在生成class时，你必须提供一组interface，然后该列就宣称实现了这些接口，所以我们可以把该class实例当做这些interface中的任意一个来使用，动态代理类其实就是一个Proxy，它不会替你做任何实质上的工作，在生成它时，你必须提供一个handle，有它接管所有实际的工作。

API说明：Proxy类提供了静态方法用于创建动态代理类和实例，并且该类也是所有动态代理类是父类（在下面的案例中证明）。

每一个代理类有如下属性：1.如果代理类实现的接口是public，则代理类是public，final和不是抽象的。

2.如果代理接口是非public的，则代理类是非public，final和不是抽象的。

3.代理类的全限定名没有被指定，代理类的类名是以$Proxy开头的

4.代理类继承了Proxy

5.代理类在被创建时显式的和有序的实现了指定的接口。

6.如果一个代理类实现了非public接口，然后

7.没一个代理类都有一个public的构造方法，该方法接受一个参数（InvocationHandler），参数是实现了InvocationHandler接口，设置invocation handler对着个代理实例。而不是通过反射API访问public的构造方法，没一个代理实例可以通过Proxy.newProxyInstance方法来创建。

没一个代理实例有如下属性：1.

### 案例

案例一：

|  |
| --- |
| /\*\*  \*抽象角色，只能使用接口，不能使用抽象类  \*/  **public** **interface** Subject {  **public** **void** request();  } |
| /\*\*  \* 真实角色  \*/  **public** **class** RealSubject **implements** Subject{  @Override  **public** **void** request() {  System.***out***.println("real subject");  }  } |
| /\*\*  \* 动态类，含义一个真实类的引用，通过构造方法传递，通常动态类的内部属性类型为Object，  \* 这样可以传递任何类型。动态类必须实现InvocationHandler接口，重写invoke方法，  \* 当生成代理对象实例后，调用代理对象方法，则程序会跳到invoke方法内执行，invoke中  \* 的method.invoke()方法其实就是被代理对象将要执行的方法。  \*/  **public** **class** DynamicSubject **implements** InvocationHandler {  **private** Object rs ;  **public** DynamicSubject(Object rs) {  **this**.rs = rs ;  }  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {    System.***out***.println("before calling:"+method);  preProcess() ;  method.invoke(rs, args) ;  System.***out***.println("after calling:"+method);  postProcess();  **return** **null**;  }  **private** **void** preProcess()  {  System.***out***.println("pre invoked");  }  **private** **void** postProcess()  {  System.***out***.println("post invoked");  }  } |
| **public** **class** Client {  **public** **static** **void** main(String[] args) {  RealSubject rs = **new** RealSubject();  InvocationHandler handler = **new** DynamicSubject(rs) ;  Class rsClass = rs.getClass() ;  //生成代理对象  Subject subject = (Subject)Proxy.*newProxyInstance*(rsClass.getClassLoader(), rsClass.getInterfaces(), handler) ;  subject.request();//调用代理对象的方法  }  } |

我们可以调用subject.getClass()和subject.getClass().getSuperclass()方法查看代理类和代理类的父类。



案例二：

|  |
| --- |
| **public** **class** VectorProxy **implements** InvocationHandler{  **private** Object obj ;  **public** VectorProxy(Object obj) {  **this**.obj = obj ;  }  **public** **static** Object factory(Object obj)  {  Class<?> objClass = obj.getClass() ;  **return** Proxy.*newProxyInstance*(objClass.getClassLoader(), objClass.getInterfaces(), **new** VectorProxy(obj)) ;  }  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  **if**(args!=**null**)  {  **for**(Object obj1 :args)  {  System.***out***.println(obj1);  }  }  Object object = method.invoke(obj, args) ;  System.***out***.println("result:"+object);  **return** object;  }  **public** **static** **void** main(String[] args)  {  List v = (List)*factory*(**new** Vector<>());  v.add(10);  v.add("lisaihao");  }  } |

缺点：被代理的类必须有接口的实现，而没有实现接口的类无法被代理。

### 源码分析

Proxy构造方法

protected Proxy(InvocationHandler h) {  
 Objects.*requireNonNull*(h);  
 this.h = h;  
}

Proxy的newProxyInstance方法

public static Object newProxyInstance(ClassLoader loader,  
 Class<?>[] interfaces,  
 InvocationHandler h)  
 throws IllegalArgumentException  
{  
 Objects.*requireNonNull*(h);  
 final Class<?>[] intfs = interfaces.clone();  
 final SecurityManager sm = System.*getSecurityManager*();  
 if (sm != null) {  
 *checkProxyAccess*(Reflection.*getCallerClass*(), loader, intfs);  
 }  
  
 /\*  
 \*获取代理类的Class对象

\*/  
 Class<?> cl = *getProxyClass0*(loader, intfs);  
 try {  
 if (sm != null) {  
 *checkNewProxyPermission*(Reflection.*getCallerClass*(), cl);  
 }  
 //调用构造方法的Constructor对象  
 final Constructor<?> cons = cl.getConstructor(*constructorParams*);  
 final InvocationHandler ih = h;  
 if (!Modifier.*isPublic*(cl.getModifiers())) {  
 AccessController.*doPrivileged*(new PrivilegedAction<Void>() {  
 public Void run() {  
 cons.setAccessible(true);  
 return null;  
 }  
 });  
 }

//生成代理类实例  
 return cons.newInstance(new Object[]{h});  
 } catch (IllegalAccessException|InstantiationException e) {  
 throw new InternalError(e.toString(), e);  
 } catch (InvocationTargetException e) {  
 } catch (NoSuchMethodException e) {  
 throw new InternalError(e.toString(), e);  
 }  
}

该类主要生成了代理类的实例，从源码中我们可以看出 代理类有一个public的构造方法，并且构造方法接口一个InvocationHandler类型的参数。

private static Class<?> getProxyClass0(ClassLoader loader,  
 Class<?>... interfaces) {

//如果要被代理类的实现的接口个数大于65535则抛出异常（这是不可能的）  
 if (interfaces.length > 65535) {  
 throw new IllegalArgumentException("interface limit exceeded");  
 }  
  
 // If the proxy class defined by the given loader implementing  
 // the given interfaces exists, this will simply return the cached copy;  
 // otherwise, it will create the proxy class via the ProxyClassFactory  
 return *proxyClassCache*.get(loader, interfaces);  
}

首先从proxyClassCache获取要代理类的实例对象，如果没有则通过ProxyClassFactory创建。

#### ProxyClassFactory

该工厂类通过给定的classloader和接口数组来创建代理类。

类名：该静态内部类实现了函数接口BiFunctiaon，泛型类型参数为：ClassLoader和Class<?>[]和Class<?>

即接受一个ClassLoader和Class数组返回一个Class内部

private static final class ProxyClassFactory  
 implements BiFunction<ClassLoader, Class<?>[], Class<?>>

// prefix for all proxy class names  
private static final String *proxyClassNamePrefix* = "$Proxy";  
// next number to use for generation of unique proxy class names  
private static final AtomicLong *nextUniqueNumber* = new AtomicLong();

这里显示了为啥没一个代理都是一$Proxy开头。

/\*  
 \* 获取动态代理类的字节码  
 \*/  
byte[] proxyClassFile = ProxyGenerator.*generateProxyClass*(  
 proxyName, interfaces, accessFlags);

#### ProxyGenerator类

private static final boolean saveGeneratedFiles = (Boolean)AccessController.doPrivileged(new GetBooleanAction("sun.misc.ProxyGenerator.saveGeneratedFiles"));

saveGeneratedFiles是保存生成的代理类字节码文件的开关，如果我们main方法中设置true，则生成的字节码文件会保存到磁盘上。设置如下：

System.*getProperties*().put("sun.misc.ProxyGenerator.saveGeneratedFiles","true");

public static byte[] generateProxyClass(final String name,  
 Class<?>[] interfaces,  
 int accessFlags)  
{  
 ProxyGenerator gen = new ProxyGenerator(name, interfaces, accessFlags);

//生成代理类的字节码  
 final byte[] classFile = gen.generateClassFile();

//如果为true，则将字节码文件保存到磁盘上  
 if (saveGeneratedFiles) {  
 java.security.AccessController.*doPrivileged*(  
 new java.security.PrivilegedAction<Void>() {  
 public Void run() {  
 try {  
 int i = name.lastIndexOf('.');  
 Path path;  
 if (i > 0) {  
 Path dir = Paths.*get*(name.substring(0, i).replace('.', File.*separatorChar*));  
 Files.*createDirectories*(dir);  
 path = dir.resolve(name.substring(i+1, name.length()) + ".class");  
 } else {  
 path = Paths.*get*(name + ".class");  
 }  
 Files.*write*(path, classFile);  
 return null;  
 } catch (IOException e) {  
 throw new InternalError(  
 "I/O exception saving generated file: " + e);  
 }  
 }  
 });  
 }  
 return classFile;  
}

## CGLIB动态代理

# 异常（Exception）

Java中异常分为两种：1.checked exception（Not Runtime Exception）

2.unchecked exception（Runtime Exception）

Java中的异常都直接或间接的继承了Exception，RuntimeException直接继承了Exception类，叫做运行时异常，直接继承Exception，而不是RuntimeException都不是运行时异常。

对于非运行时异常，必须对其处理，处理的方式有两种：第一种：使用try…catch..finally语句进行捕获。第二种：在会产生异常方法中声明throws Exception

对于运行时异常，可以进行处理也可以不进行处理，建议不进行处理。

如果try语句块中，有return语句，那么首先现将finally语句执行完毕，然后方法返回。如果try语句块有System.exit()，则finally语句不会被执行，此时虚拟机已经停止工作。

自定义异常，通常我们都直接继承Exception类，而不是继承某个运行时异常，继承Exception类我们都叫做自定义异常。

# 内部类

## 静态内部类

静态内部类可以被编译成完全独立的.class文件

静态内部类只能访问外部类的静态属性和方法

创建实例：外部类.内部类 变量名=new 外部类.内部类();

|  |
| --- |
| **public** **class** StaticInnerClass {  **public** **static** **void** main(String[] args)  {  A.B b = **new** A.B();  b.*testA*();  b.testA2();  }  }  **class** A  {  **private** **static** **int** *a* = 10 ;  **private** **int** b = 20 ;  **public** **static** **void** testAA()  {  System.***out***.println("A method testAA");  }  **private** **void** testAA2()  {  System.***out***.println("A private method");  }  /\*\*  \* 静态内部类，可以访问外部类的静态变量和静态方法  \*/  **public** **static** **class** B  {  **public** **static** **void** testA()  {  System.***out***.println(*a*);  *testAA*();  }  **public** **void** testA2()  {  System.***out***.println(*a*);  *testAA*();  }  }  } |

## 成员内部类

定义在一个内部类，不适用static修饰，像一个实例变量一样。

成员内部类的非静态方法可以访问所有外部类的方法和成员变量。

成员内部类不能定义静态方法。

如果成员内部类定义了一个和外部类相同（相同类型和相同变量名）的非静态成员变量，并且成员内部类方法想调用外部类的那个成员变量，则使用：**外部类名.this.变量名**的方式调用。

成员内部类定义静态的成员变量，必须使用final修饰。

外部类非静态方法想调用成员内部类的方法使用:this.内部类名().方法名()的方式。

外部类静态方法调用成员内部类的方法需要先实例化外部类，再实例化内部类，最后调用内部类方法： **new 外部类名().内部类名().方法名();**

创建实例：外部类.内部类 变量名=new 外部类().new 内部类();

|  |
| --- |
| **public** **class** MemberInnerClass  {  **public** **static** **void** main(String[] args)  {  /\*B.BB b = new B().new BB();  b.testBB();\*/  B.*testB*();  }  }  **class** B  {  **private** **static** **int** *a* = 10 ;  **private** **int** b = 20 ;  **public** **static** **void** testB()  {  System.***out***.println("B method testB");  **new** B().**new** BB().testBB();;  }  **public** **void** testBB1()  {  System.***out***.println("B method testBB");  //非静态方法调用内部类方法  **this**.**new** BB().testBB();  }  **public** **class** BB  {  **static** **final** **int** ***a*** = 101 ;  **int** b = 30 ;  **public** **void** testBB()  {  System.***out***.println(***a***); //101  System.***out***.println(b); //30  System.***out***.println(B.**this**.b);//20  System.***out***.println(B.*a*); //10  System.***out***.println();  /\*testBB1();  testB();\*/  }  }  } |

## 局部内部类

定义在方法内部的类叫做局部内部类，使用很少。

局部内部类就像一个局部变量一样，不能声明为public ,private ,protected和static的。

局部内部类只能访问方法内定义的final类型的局部变量。

在方法内部的局部变量不能使用static修饰。

|  |
| --- |
| **public** **class** LocalInnerClass  {  **public** **static** **void** main(String[] args) {  **new** InnerClass().inner();  }  }  **class** InnerClass  {  **public** **void** inner()  {  **final** **int** a = 10 ;  **final** **int** b = 20 ;  //局部内部类  **class** Inner  {  **public** **void** doSoming()  {  System.***out***.println("inner dosomging a="+a+" b="+b);  }  }  //调用局部内部类方法  **new** Inner().doSoming();  }  } |

## 匿名内部类

没有名字的内部类叫做匿名内部类，是一个特许的局部内部类。

没有关键字class，extends和implements，没有构造方法。

当我们定义一个匿名内部类时，会隐式的继承一个父类或者实现一个接口。

|  |
| --- |
| **public** **class** AnonymousInnerClass  {  @SuppressWarnings("deprecation")  **public** String get(Date date)  {  **return** date.~~toLocaleString~~();  }  **public** **static** **void** main(String[] args) {  AnonymousInnerClass test = **new** AnonymousInnerClass() ;  //普通传递参数  System.***out***.println(test.get(**new** Date()));  //匿名传递参数，即生成了继承了Date类型的子类  System.***out***.println(test.get(**new** Date() {  @Override  **public** String toLocaleString()  {  **return** "lisaihao ";  }  }));  }  } |

# I/O流

## 字节输入流

## 字节输出流

## 字符输入流

## 字符输出流

## 装饰模式

**装饰模式以对客户端透明的方式扩展对象的功能，是继承关系的一种替代方案，提供比继承更多的灵活性。动态地给一个对象添加功能，这些功能可以再动态地撤消，可增加有一些基本功能的排列组合而产生的非常大量的功能。装饰模式是通过创建一个包装对象来实现的，也就是用装饰来包裹真是对象。**

## AutoCloseable类

在JDK1.7中，为我们提供了一个AutoCloseable类。

API：

**public** **class** AutoCloseableTest **implements** AutoCloseable

{

BufferedInputStream bis = **null** ;

**public** **void** readFileByPath(String path) **throws** Exception

{

File file = **new** File(path);

bis = **new** BufferedInputStream(**new** FileInputStream(file));

**int** readLen = 0;

**byte**[] readByte = **new** **byte**[1024];

StringBuffer sb = **new** StringBuffer();

**while**(-1!=(readLen=bis.read(readByte)))

{

System.***out***.println(**new** String(readByte));

}

}

@Override

**public** **void** close() **throws** Exception

{

bis.close();

System.***out***.println("close method invoked");

}

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

{

**try**(AutoCloseableTest act = **new** AutoCloseableTest())

{

act.readFileByPath("");

} **catch** (Exception e) {

e.printStackTrace();

}

}

}

# NIO

NIO全称 non-blocking IO，是指jdk1.4 及以上版本里提供的新API（New IO），为所有的原始类型（boolean类型除外）提供[缓存](https://baike.baidu.com/item/%E7%BC%93%E5%AD%98/100710)支持的数据容器，使用它可以提供非阻塞式的高伸缩性网络。

## IO vs NIO

IO是基于流（Stream）实现的，这就意味着每次只能从流中读取一个或多个字节，直到流中的字节被完全读取，然而这种方式的很容易为流创建一个或多个过滤器，链接几个过滤器很容易对流数据进行处理，不过这种方式对流的处理速度很慢，无法满足我们需求。

NIO是基于块实现的，使用分块的方式处理数据，每次操作消耗或生成一个或多个数据块。NIO的核心对象是：通道（Channel）和缓存（Buffer）组成。

## 通道（Channel）

通道是用于打开一个链接，例如：硬件设备，文件，网络链接或者操作程序组件。通道也是一个对象，我即可以从通道中读取数据，又可以将数据写入通道中，但是通道没有任何操纵数据的能力，如果要操纵数据，首先要将通道中的数据要先写入到一个Buffer中。

通道有类似于流，但又有些不同：（1）通道既可以写入数据，又可以读取数据，而流的读，通常是单向的。

（2）通道可以异步的读写，而流不可以。

（3）通道中的数据总是从一个Buffer中读取。

在Java NIO中提供了多个通道对象，每一个对象都直接或间接实现了Channel接口。

### SocketChanel

#### ServerSocketChannel

ServerSocketChannel是一个面向流的监听套接字的可选择通道。该通道不是监听网络套接字的完全抽象，绑定和套接字选项的操作必须通过socket方法获得与之关联的ServerSocket对象，不可能为

任意的，已经存在的服务套接字创建通道。

|  |
| --- |
| **public** **class** ServerSocket  {  **public** **static** **void** main(String[] args) **throws** IOException {  //开启服务器socket，但是没有绑定  ServerSocketChannel ssc = ServerSocketChannel.*open*() ;  //创建绑定地址  SocketAddress address = **new** InetSocketAddress("localhost",8888);  //绑定  ssc.bind(address) ;  //配置非堵塞模式  ssc.configureBlocking(**false**) ;  **while**(**true**){  //接受连接请求  SocketChannel channel = ssc.accept() ;  }  }  } |

#### SocketChannel

一个面向流的连接套接字的可选通道，改通道不是一个监听网络套接字的完全抽象，绑定，关闭和操作套接字选项必须同socket方法，获得与之关联的Socket对象，对于任意的，已存在的socket不能为它创建通道，socket通道支持非堵塞的连接

非堵塞模式下：read()方法只传输立即可用数据，如果没有立即可用数据，则返回0，同理，

数据没有写入Socket，则write()方法返回0，对于server socket如果当前没有客户端进行

连接，则accept()方法返回null

|  |
| --- |
| **public** **class** ClientSocket {  **public** **static** **void** main(String[] args) **throws** IOException{  //开启SocketChanel服务  SocketChannel sc = SocketChannel.*open*();  //创建连接地址  SocketAddress address = **new** InetSocketAddress("localhost",8888);  //阻塞  sc.configureBlocking(**true**) ;  //连接  **boolean** isConnect = sc.connect(address) ;    }  } |

## 缓存区（Buffer）

Buffer包含了对基本数据类型的操作，实际上一个Buffer就是一个容器对象（其实就是数组），在NIO中所有对数据的操作都要在Buffer中进行，Buffer不仅是一个容器，而且它还提供了对数据结构化的访问，还可以追踪系统中读写进程。

Buffer提供了四个重要属性：容量（Capacity），上界（Limit），位置（Position）和标记（Mark）。

容量：缓存区能够容纳数据的最大数据，这个容量在缓存区创建时就已经确定，不可以改变。

上界：缓存区第一个不能被读或写的元素，是一个索引，且永远不能为负数。

位置：下一个要被读出或写入的元素的索引。位置会自动由相应的 get( )和 put( )函数更新。

标记：在调用reset方式时将其位置重置到的索引位置。

0 <= mark <= position <= limit <= capacity

Buffer常用方法:Buffer.rewind() pos=0,mark丢弃，重绕。

Buffer.clear() pos=0,lim=cap,mark丢弃，清空。

Buffer.flip() lim=pos ,pos=0,mark丢弃

Buffer.reset() pos=mark

Buffer.slice() 切片，从当前缓冲区中划出前n个元素构造新的缓冲区，n等于当前buffer的reminding大小。

# 正则表达式

正则表达式列表：

|  |  |
| --- | --- |
| 字符 | 描述 |
| \ | 将下一个字符标记为一个特殊字符，一个原义字符，一个向后引用或一个八进制转义符。 |
| ^ | 匹配输入字符串的开始位置。如果设置了正则表达式对象的Mulitiline属性，$也匹配\n或\r之后的位置。 |
| $ | 匹配输入字符串的结束位置。 |
| \* | 匹配前面的子表达式零次或多次。 |
| + | 匹配前面的子表达式零次和一次。 |
| ? | 匹配前面的子表达式零次或一次。 |
| {n} | n是一个非负整数。匹配确定的n次 |
| {n,} | n是一个非负整数。匹配至少n次 |
| {n,m} |  |
| . |  |
| (pattern) | 匹配pattern并获取这一匹配。锁获取的匹配可以从产生的Matches集合得到。要匹配圆括号字符，请使用\()或\ |
| (?:pattern) |  |
| (?=pattern) |  |
| (?!pattern) |  |
| x|y |  |
| [xyz] |  |
| [^xyx] |  |
| [a-z] |  |
| [^a-z] |  |
| \b |  |
| \B |  |
| \cx |  |
| \d | 匹配一个数字字符。 |
| \D | 匹配一个非数字字符 |
| \f | 匹配一个换页符 |
| \n | 匹配一个换行符 |
| \r | 匹配一个回车符 |
| \s | 匹配任何空白字符，包括空格，制表符，换页符等，等价于[\f\n\r\t\v] |
| \S | 匹配任何非空白字符，等价于[^\f\v\r\n\t] |
| \t | 匹配一个制表符，等价于”\x09”和”\cI” |
| \v | 匹配一个垂直制表符。等价于”\x0b”和”\cK” |
| \w | 匹配包括下划线的任何单词字符。等价于[a-zA-z0-9\_] |
| \W | 匹配任何非单词字符，等价于[^A-Za-z0-9] |
| \xn |  |
| \num |  |
| \n |  |
| \nm |  |
| \nml |  |
| \un | 匹配n，其中n是一个用四个十六进制数字表示的Unicode字符，例如，”\u00A9”匹配版权符号© |

**以多个条件分隔字符串：**

Pattern p = Pattern.*compile*("[,|\\s]") ;

String[] strs = p.split("Java Hello World,LI Sai Hao") ;

**for**(String ss :strs)

{

System.***out***.println(ss);

}

**文字替换：**

Pattern p = Pattern.*compile*("Java") ;

Matcher m = p.matcher("Java Hello World,LI Java Hao") ;

System.***out***.println(m.replaceFirst("Hao"));

System.out.println(m.replaceAll("Hao"));

**验证是否为邮箱地址：**

String emilRegu = "[\\w\\.\\-]+@([\\w\\-]+\\.)+[\\w\\-]+" ;

Pattern eP = Pattern.*compile*(emilRegu, Pattern.***CASE\_INSENSITIVE***) ;

Matcher eM = eP.matcher("1622443440@qq.com");

System.***out***.println(eM.matches());

**截去**[**http://地址**](http://地址)**：**

Pattern p = Pattern.*compile*("(http://|https://){1}[\\w\\.\\-/:]+");

Matcher m = p.matcher("<img href=\"http://www.haodashen.com/img/hao.png\"/>");

StringBuffer buffer = **new** StringBuffer();

**while**(m.find())

{

buffer.append(m.group());

buffer.append("\r\n");

}

System.***out***.println(buffer.toString());

**查找html对应的字符串：**

Pattern p = Pattern.*compile*("href=(\"|\').+?.html(\"|\')",Pattern.***DOTALL***);

Matcher matcher = p.matcher("<div style=\"widht:100px;height=200px;border:1px solid red;\">"

+ "<span><a href=\"index.html\">首页</a>www.baidu.com/index.html</sapn></div>");

**if**(matcher.find())

{

System.***out***.println(matcher.group(0));

}

**替换{}中的文字：**

**public** **static** String replace(String sourceString , Map<String ,Object> object)

{

String temp = sourceString ;

Collection<Object> values = object.values();

Set<String> keys = object.keySet() ;

Iterator<String> iterator = keys.iterator();

**while**(iterator.hasNext())

{

String key = iterator.next() ;

String s = (String)object.get(key) ;

Pattern p = Pattern.*compile*(key) ;

Matcher m = p.matcher(temp) ;

temp = m.replaceAll(s);

}

**return** temp ;

}

**以正则条件查询指定目录下的文件：**

|  |
| --- |
| **public** **class** FilesAnalyze  {  **private** ArrayList files = **new** ArrayList<>();    **private** String \_path ;    **private** String \_regexp ;    **class** MyFileFilter **implements** FileFilter  {  @Override  **public** **boolean** accept(File file)  {  Pattern pattern = Pattern.*compile*(\_regexp);  Matcher matcher = pattern.matcher(file.getName());  **return** matcher.matches();  }  }    **public** FilesAnalyze(String path ,String regexp)  {  getFileName(path,regexp);  }    **private** **void** getFileName(String path ,String regexp)  {  **this**.\_path = path ;  **this**.\_regexp = regexp ;  File directory = **new** File(\_path);  File[] filesFile = directory.listFiles(**new** MyFileFilter());  **if**(filesFile == **null**) **return** ;  **for**(**int** j = 0 ;j<filesFile.length;j++)  {  files.add(filesFile[j]);  }  **return** ;  }    **public** **void** print(PrintStream out)  {  Iterator elements = files.iterator();  **while**(elements.hasNext())  {  File file = (File) elements.next();  out.println(file.getPath());  }  }  **public** **static** **void** main(String[] args) {  FilesAnalyze fa = **new** FilesAnalyze("D:\\", "[A-z|.}]\*");  fa.print(System.***out***);  }  } |

# 日期API

## Calendar

**API说明：**Calendar是一个抽象类，该类提供了特定时间点与一组日志字段，如YEAR，MONTH，DAY\_OF\_MONTH,HOUR等等相互转换的方法，并为操作日志字段提供了一些方法，如获取下一周的日期。一个时间点可以使用毫秒表示，它是从1970年1月1日00:00:00.000（格林威治标准时间）到给时间点的偏移量。

该类还为包外的具体日历系统提供了其他字段和方法，这些字段和方法被定义为protected。

与其他locale-sensitive类一样，Calendar提供了getInstance类方法，用于获取该类型的通用对象。Calendar的getInstance方法返回一个Calendar对象，并且该对象的字段将使用当前日期和时间来初始化。

对于一个特殊的语言和日历格式，Calendar可以生成实现日期-时间格式的日历字段值。Calendar定义了返回具体日历字段的范围值和这些值的意义。比如，日历系统中第一个月的值是MONTH=JANUARY，其他值被子类所定义。