# API

API（**Application Programming Interface**）应用程序编程接口。

## Object类

### Object类简介

Object类是所有Java类的根父类，如果在类的声明中未使用extends关键字指明其父类，则默认父类为Object类,因此所有类都可以调用Object类中的方法。

Object中只有一个空参的构造器，并且有11个方法，其中wait方法有三个重载方法。

### Object类方法

|  |  |
| --- | --- |
| **方法摘要** | |
| protected Object | **clone**()创建并返回此对象的一个副本。 |
| boolean | **equals**(Object obj) 判断其他对象是否与当前对象相等。自定义类建议重写本方法。 |
| protected void | **finalize**()当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。 |
| Class | **getClass**()返回此对象的运行时对象。 |
| int | **hashCode**()返回该对象的哈希值。自定义类建议重写本方法。 |
| void | **notify**()唤醒在此对象监视器上等待的单个线程。 |
| void | **notifyAll**()唤醒在此对象监视器上等待的所有线程。 |
| String | **toString**()返回该对象的字符串表示。自定义类建议重写本方法。 |
| void | **wait**()在其他线程调用此对象的notify()方法或notifyAll()方法前，导致当前线程等待。 |
| void | **wait**(long timeout) 在其他线程调用此对象的 [notify()](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html#notify())方法或 [notifyAll()](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html#notifyAll())方法，或者超过指定的时间量前，导致当前线程等待。 |
| void | **wait**(long timeout,int nanos) 在其他线程调用此对象的 [notify()](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html#notify())方法或 [notifyAll()](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/lang/Object.html#notifyAll())方法，或者其他某个线程中断当前线程，或者已超过某个实际时间量前，导致当前线程等待。 |

#### equals方法

Object中equals方法源码：（比较的是两个对象的地址值）

public boolean equals(Object obj) {

return (this == obj);

}

系统提供的一些类，比如String,Date等等都已经重写了equals方法，用来比较内容。以后只要是自定义的类建议重写equals方法，用来比较内容。

当我们调用equals方法时，默认调用的是Object中的equals方法用来比较地址值。但是我们在调用equlas方法时希望比较的是内容。所以自定义类都要重写equals方法。

==和equals的区别？

== ： 如果是基本数据类型那么比较的是具体的值。如果是引用数据类型那么比较的是地址值。（判断两个引用是否指向同一块内存）

equals : 如果没有重写比较的是地址值。如果重写了那么比较的是内容。

**关系运算符==的使用**

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  // ==：  // 基本数据类型：根据基本数据类型的值判断是否。相等返回true，反之返回false  // 两端数据类型可以不同，在不同的情况下，会进行自动类型转  // 引用数据类型：比较引用类型变量的地址值是否相等。（首地址值）  **int** i = 12;  **int** j = 12;  System.***out***.println(i == j);// true  **char** c = 12;  System.***out***.println(i == c);// true  **float** f = 12.0f;  Object obj1 = **new** Object();  Object obj2 = **new** Object();  Object obj3 = obj1;  System.***out***.println(obj1 == obj3);// true  System.***out***.println(obj1 == obj2);// false  }  } |
| true  true  true  false |

**equals方法的使用**

equals比较时字符串常量一般放在前面，避免字符串空值的问题。定义的String类型可能为一个空对象，导致调用equals方法时会产生空指针异常。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  /\*  \* 因为equals是方法，只能由对象调用，所以equals方法只能比较引用数据类型  \* 查看Object类中源码，发现其equals方法也是 == 去比较首地址值  \*/  Object obj1 = **new** Object();  Object obj2 = **new** Object();  System.***out***.println(obj1.equals(obj2));  System.***out***.println(obj1==obj2);  /\*  \* 1、String类因为重写了Object类中的equals方法，因此比较的是内容  \* 2、很多类如包装类、File类、Date类重写了Object类的equals()方法，是比较对象的具体"实体内容"是否相同  \*/  String str1 = **new** String("AA");  String str2 = **new** String("AA");  System.***out***.println(str1 == str2);  //首先判断首地址是否相等，是，返回true，否，继续比较额每一个字符，直到全部相等才返回true ，否则，返回false  System.***out***.println(str1 .equals(str2));  }  } |
| false  false  false  true |

**equals方法的简单重写**

不同的类创建的对象具有不同的逻辑相等概念，只要其内容相等就可以判断其为相同的对象，而默认继承的Object类中的equals方法比较的是地址值而非内容，因此创建新的类时需要重写Object类中继承来的equals方法。

注意：重写equals方法时也要重写hashCode方法，保证不同的对象具有不同的哈希码，相同的对象具有相同的哈希码。

在重写equals方法时，要注意满足离散数学上的特性

（1）自反性：对任意x，x.equals(x)的返回值一定为true.

（2）对称性：对任意x,y,当y.equals(x)返回值为true时，x.equals(y)的返回值一定为true;

（3）传递性：如果x.equals(y)=true, y.equals(z)=true,则x.equals(z)=true

（4）一致性：如果参与比较的对象没任何改变，则对象比较的结果也不应该有任何改变

（5）非空性：任何非空的引用值x，x.equals(null)的返回值一定为false

注意：当此方法被重写时，通常有必要重写 hashCode 方法，以维护 hashCode 方法的常规协定：

相等对象必须具有相等的哈希码，

两个对象的哈希码不相等，那么equals一定不相等。

两个对象的哈希码相等，那么equals结果可能相等也可能不相等

重写equals方法时要注意如果比较对象是null，返回false；如果比较对象不是相同类型数据，返回false；如果是同类型且需要比较的属性都相同，返回true。

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  Creature p5 = **new** Creature("xmm",24);  Creature p6 = **new** Creature("xmm",24);  System.***out***.println(p5.equals(p6));  Creature p1 = **new** Person("xmm",24);  Creature p2 = **new** Person("xmm",24);  //p1p2比较时因为多态的原因，导致编译时看左边，所以equals方法是Object类中的方法  //但是运行时看右边，所以运行时equals方法是Person重写的方法  System.***out***.println("\n"+p1.equals(p2));  Person p3 = **new** Person("xmm",24);  Person p4 = **new** Person("xmm",24);  System.***out***.println("\n"+p4.equals(p3));  }  }  **class** Creature{  String name;  **int** age;  //父类必须有空参的构造器，因为子类的构造器中的super默认会调用父类的空参构造器  Creature(){  System.***out***.print("default");  }  **public** Creature(String name,**int** age) {  **this**.name = name;  **this**.age = age;  }  }  **class** Person **extends** Creature{  String name;  **int** age;  **public** Person(String name,**int** age) {  //此处省略super();会调用父类的空参构造器  **this**.name = name;  **this**.age = age;  }  **public** **boolean** equals(Object obj) {  **if** (**this** == obj)  **return** **true**;  **if**(obj **instanceof** Person) {  Person p = (Person)obj;  **return** name.equals(p.name) && age == p.age;  }**else** {  **return** **false**;  }  }  } |
| false  defaultdefault  true  defaultdefault  true |

#### toString方法

Object中的toString方法源码

public String toString() {

return getClass().getName() + "@" + Integer.toHexString(hashCode());

}

像系统提供的一些类，比如String,Date等等都已经重写了toString方法。用来输出对象的内容。以后只要是自定义的类建议重写toString方法，用来输出对象的内容。

自定义类中如果没有重写toString方法，那么默认调用的是Object中的toString输出的是地址值。

|  |
| --- |
| **public** **class** ToStringTest {  **public** **static** **void** main(String[] args) {  Person p = **new** Person("aa",18);  //com.atguigu.java3.Person@659e0bfd  System.***out***.println(p.toString());  //com.atguigu.java3.Person@659e0bfd  //当我们输出对象的引用时，其实默认调用的就是该对象的toString方法  System.***out***.println(p);  //输出全类名  System.***out***.println(p.getClass().getName());  //String类重写了Object类的toString方法  String s = **new** String("abc");  System.***out***.println(s.toString());  //Date类 重写了Object类的toString方法  Date date = **new** Date();  System.***out***.println(date.toString());  }  }  **class** Person{  String name;  **int** age;  **public** Person(String name,**int** age) {  **this**.name = name;  **this**.age = age;  }  **public** String toString() {  **return** name + " " + age;  }  } |
| aa 18  aa 18  com.atguigu.java3.Person  abc  Wed Apr 17 19:36:07 CST 2019 |

#### hashCode方法

hashCode方法提高具有哈希结构的容器的效率，两个引用，如果指向的是同一个对象，则哈希值肯定是一样的，如果指向的是不同对象，则哈希值是不一样的。哈希值主要根据地址号来的，但是不能完全将哈希值等价于地址。

hashCode方法是相应对象整型的 hash 值。它常用于基于 hash 的集合类，如 Hashtable、HashMap、LinkedHashMap等等。它与 equals() 方法关系特别紧密。根据 Java 规范，使用 equal() 方法来判断两个相等的对象，必须具有相同的 hashcode。

将对象放入到集合中时，首先判断要放入对象的hashcode是否已经在集合中存在，不存在则直接放入集合。如果hashcode相等，然后通过equal()方法判断要放入对象与集合中的任意对象是否相等：如果equal()判断不相等，直接将该元素放入集合中，否则不放入。

**注意**：

两个不相等的对象是有可能具有相同的哈希值的，这种情况很少出现；但是两个相等的对象哈希值必定是相等的，因此hashCode方法中不能使用随机数字。

**重写hashcode方法的基本原则**

程序运行时，同一个对象对此调用hashCode方法应该返回相同的值；

**\***当两个对象的equals方法比较返回true时，这两个对象的hashCode方法的返回值必须相等；

对象中在equals方法内比较的属性，都应该用来计算hashCode值。

|  |
| --- |
| //自定义类中重写的hashCode方法，其中每个属性都用来计算哈希值  **public** **int** hashCode() {  **final** **int** prime = 31;  **int** result = 1;  result = prime \* result + age;  //name是String类型的，其String类也重写了hashCode方法  result = prime \* result + ((name == **null**) ? 0 : name.hashCode());  **return** result;  } |

**数字31的由来？**

选择系数的时候要选择尽量大的系数，因为哈希值越大，冲突越少，查找效率也会提高（减少冲突）。

31只占用5bits，相乘造成数据溢出的概率较小（减少冲突）。

31可以由i\*31== (i<<5)-1来表示,现在很多虚拟机里面都有做相关优化.（提高算法效率）。

31是一个素数，素数作用就是如果我用一个数字来乘以这个素数，那么最终的出来的结果只能被素数本身和被乘数还有1来整除(减少冲突)。

#### finalize方法

1. 当对象被回收时，系统自动调用该对象的finalize方法。子类可以重写该方法，做一些释放资源的操作。
2. 什么时候被回收：当某个对象没有任何引用时，则jvm就认为这个对象是一个垃圾对象，就会使用垃圾回收机制来销毁该对象，在销毁该对象前，会先调用finalize方法。
3. 垃圾回收机制的调用，是由系统来决定, 也可以通过System.gc() 主动触发垃圾回收机制。

#### getClass方法

获取任意对象运行时对象，即其所属字节码文件对象。

#### clone方法

java.lang.Cloneable 是一个标示性接口，不包含任何方法，clone 方法在 object 类中定义。并且需要知道 clone() 方法是一个本地方法，这意味着它是由 c 或 c++ 或 其他本地语言实现的。

该方法有一个默认行为：super.clone();因为首先要把父类中的成员复制到位，然后才是复制自己的成员。

**深拷贝和浅拷贝的区别**

浅拷贝：被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用仍然指向原来的对象。换言之，浅拷贝仅仅复制对象的引用，而不复制它所引用的对象。

深拷贝：被复制对象的所有变量都含有与原来的对象相同的值，而那些引用其他对象的变量将指向被复制过的新对象，而不再是原有的那些被引用的对象。换言之，深拷贝把要复制的对象所引用的对象都复制了一遍。

## System类

System 类包含一些有用的类字段和方法。它不能被实例化。

标准输入、标准输出和错误输出流对象

PrintStream err：“标准”错误输出流。

InputStream in：“标准”输入流。

PrintStream out：“标准”输出流。

System.currentTimeMillis()获取当前时间到基准时间（1970年1月1日0时0分0秒）的时间长度，单位为毫秒，一般该方法用于计算时间差。

|  |  |
| --- | --- |
| **System方法摘要** | |
| static void | **exit**(int status) 终止当前正在运行的 Java 虚拟机 |
| static void | **arraycopy**(Object src,int srcPos,Object dest,int destPos,int length) 复制数组元素，比较适合底层调用，一般使用Arrays.copyOf完成复制数组 |
| static long | **currentTimeMillens**()返回当前时间距离基准时间的时间长度 |
| static void | **gc**()运行垃圾回收机制，但是真正开始的时间不确定 |
| static void | **setErr**(PrintStream err) 重新分配标准错误输出流 |
| static void | **setIn**(InputStream in) 重新分配标准输入流 |
| static void | **setOut**(PrintStream out) 重新分配标准输出流 |
| static void | **gerProperties**(Properties props) 将系统属性设置为Properties参数 |
| static String | **setProperty**(String key,String value) 设置指定键指示的系统属性 |

System类中的gc方法可用于垃圾回收，当使用System.gc()回收某个对象所占用的内存之前，通过要求程序调用适当的方法来清理资源。在没有明确指定资源清理的情况下，Java提高了默认机制来清理该对象的资源，就是调用Object类的finalize方法。finalize方法的作用是释放一个对象占用的内存空间时，会被JVM调用。而子类重写该方法，就可以清理对象占用的资源，该方法有没有链式调用，所以必须手动实现。

从程序的运行结果可以发现，执行System.gc()前，系统会自动调用finalize方法清除对象占有的资源，通过super.finalize()方式可以实现从下到上的finalize方法的调用，即先释放自己的资源，再去释放父类的资源。

但是，不要在程序中频繁的调用垃圾回收，因为每一次执行垃圾回收，jvm都会强制启动垃圾回收器运行，这会耗费更多的系统资源，会与正常的Java程序运行争抢资源，只有在执行大量的对象的释放，才调用垃圾回收最好。

Runtime类中也有gc方法。

## Scanner类

|  |  |
| --- | --- |
| **Scanner方法摘要** | |
| boolean | **hasNextXxx**()判断是否有下一个数据且为Xxx类型 |
| boolean | **hasNext**()判断是否有下一个数据且为字符串 |
| Xxx | **nextXxx**()获取下一个Xxx数据 |
| String | **next**()获取下一个字符串数据 |

## 包装类

因为Java是个面向对象的语言，其基本数据类型不是面向对象的，又有其优势（效率高和节省空间）。但是实际使用中经常需要将基本数据类型转换为对象，便于操作：

1. 集合的操作
2. 使用Object类型接受任意类型的数据
3. 泛型实参

针对八种基本数据类型定义相应的引用类型，有了类的特点就可以调用类中的方法，包装类就是基本数据类型的增强版，每一个包装类中都有很多关于对应的基本数据类型的方法。



其中除了Boolean和Character，其他类都是Number的子类。

**包装类的作用**

**数据类型的范围：**包装类提供了许多常量表示数据类型的最大值、最小值等

**数据类型的转换：**包装类可以和基本数据类型互相转换

**int 和Integer占用内存**

Integer 对象会占用更多的内存。Integer是一个对象，需要存储对象的元数据。但是 int 是一个基本数据类型，所以占用的空间更少。

|  |  |
| --- | --- |
| **包装类方法摘要** | |
|  | **Xxx**(int value)实例化Xxx对象，表示value的值 |
|  | **Xxx**(String s)实例化Xxx对象，表示String参数所指示的Xxx类型的值 |
| static Xxx | **parseXxx**(String s)将字符串参数作为基本数据类型进行解析 |
| Xxx | **XxxValue**()从基本数据类型的Xxx类型返回该包装类的值 |

### 包装类、基本数据类型和String之间的转换

#### 基本数据类型转包装类

通过包装类的构造器转换

byte a = 100;  
char c = 'a';  
boolean b = true;  
Byte byte1 = new Byte(a);  
Short short1 = new Short(a);  
Integer integer = new Integer(a);  
Long long1 = new Long(a);  
Float float1 = new Float(a);  
Double double1 = new Double(a);  
Character character = new Character(c);  
Boolean boolean1 = new Boolean(b);

#### 包装类转基本数据类型

通过包装类的对象.xxxValue方法就可以转换，xxx就是需要转换的基本数据类型

byte a = new Byte((byte) 10).byteValue();  
short b = new Short((short) 10).shortValue();  
int c = new Integer(10).intValue();  
double f = new Double(0.0).doubleValue();  
char g = new Character('a').charValue();  
boolean h = new Boolean("aaa").booleanValue();

#### 自动拆箱和自动装箱

只有包装类和对应的基本数据类型可以自动装箱和拆箱

装箱就是将基本数据类型转换为引用数据类型，编译器省略了调用valueOf静态方法。

拆箱就是将引用数据类型转换为基本数据类型，编译器省略了对象调用的xxxValue方法。

**自动拆箱**：将包装类的对象直接赋值给对应的基本数据类型

包装类的对象和基本数据类型进行运算时，就会发生自动拆箱。

|  |
| --- |
| **public** **class** Example{  **public** **static** **void** main(String[] args) {  **byte** a = **new** Byte((**byte**) 10);  **short** b = **new** Short((**short**) 10);  **int** c = **new** Integer(10);  **double** f = **new** Double(0.0);  **char** g = **new** Character('a');  **boolean** h = **new** Boolean("aaa");  }  } |

**自动装箱**：将基本数据类型直接赋值给对应的包装类

直接使用基本数据类型给包装类赋值时，就会发生自动装箱。

|  |
| --- |
| **public** **class** Example{  **public** **static** **void** main(String[] args) {  Integer integer = 10;  Long long1 = 10L;  Float float1 = 0.0f;  Double double1 = 0.0;  Character character = 'a';  Boolean boolean1 = **false**;  }  } |

**自动装箱的底层实现**

Byte/Short/Integer/ Long：如果赋值的数值在-128~127则从数组中直接获取对应的Integer对象，如果不在这个范围内，就创建一个新的Short/Integer/Long对象

Character底层缓存了0~127

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  Integer i = **new** Integer(1);  Integer j = **new** Integer(1);  System.***out***.println(i == j);  // 自动装箱底层调用了valueOf方法  // 如果是在 -128~127 的自动装箱，就是从cache(缓存)中返回即可(就是同一个对象)  /\*  \* 底层代码实现  public static Integer valueOf(int i) {  if (i >= IntegerCache.low && i <= IntegerCache.high)  return IntegerCache.cache[i + (-IntegerCache.low)];  return new Integer(i);  }  \*/  Integer m = 1;  Integer n = 1;  System.***out***.println(m == n);  Integer x = 128;  Integer y = 128;  System.***out***.println(x == y);  }  } |
| false  true  false |

#### String转包装类

通过包装类的构造器转换 Integer integer = new Integer(String str);

通过包装类的静态方法valueOf(str) Integer integer = Integer.valueOf(String str)

|  |
| --- |
| **public** **class** Example{  **public** **static** **void** main(String[] args) {  Integer integer = **new** Integer("10");  Double double1 = **new** Double("0.0");  Character character = "a".charAt(0);  //Boolean构造器的方法调用了parseBoolean方法，只要传递的字符串不是true，则返回false  Boolean boolean1 = **new** Boolean("true");  }  } |

#### String转基本数据类型

通过包装类中的Xxx.parseXxx方法，Xxx就是包装类

|  |
| --- |
| **public** **class** Example{  **public** **static** **void** main(String[] args) {  **int** integer = Integer.*parseInt*("10");  **double** double1 = Double.*parseDouble*("0.0");  **char** character = "a".charAt(0);  **boolean** boolean1 = Boolean.*parseBoolean*("true");  }  } |

通过包装类的构造器实现

|  |
| --- |
| **public** **class** Example{  **public** **static** **void** main(String[] args) {  **int** c = **new** Integer("10");  **double** f = **new** Double("0.0");  **char** g = **new** Character('a');  **boolean** h = **new** Boolean("aaa");  }  } |

#### 包装类转String

包装类对象.toString方法

|  |
| --- |
| **public** **class** Example{  @SuppressWarnings("unused")  **public** **static** **void** main(String[] args) {  String a = **new** Byte((**byte**) 10).toString();  String b = **new** Short((**short**) 10).toString();  String c = **new** Integer(10).toString();  String f = **new** Double(0.0).toString();  String g = **new** Character('a').toString();  String h = **new** Boolean("aaa").toString();  }  } |

#### 基本数据类型转String

1、基本数据类型后面连接空字符串

2、String.valueof(基本数据类型)方法

|  |
| --- |
| **public** **class** Example{  @SuppressWarnings("unused")  **public** **static** **void** main(String[] args) {  //基本数据类型后面连接空字符串  String string1 = 1 + "";  String string2 = 'a' + "";  //String.valueof(基本数据类型)方法  String string3 = String.*valueOf*(10);  String string4 = String.*valueOf*('a');  }  } |

### 包装类方法

#### Interger和Character常见方法和常用属性

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  System.***out***.println(Integer.***MAX\_VALUE***);  System.***out***.println(Integer.***MIN\_VALUE***);  // Character常用方法  System.***out***.println(Character.*isDigit*('a'));// false  // 判断是不是一个字母  System.***out***.println(Character.*isLetter*('a'));// true  // 判断是不是大写的  System.***out***.println(Character.*isUpperCase*('A'));// true  // 判断是不是小写的  System.***out***.println(Character.*isLowerCase*('A'));// false  // 将字符转成大写  System.***out***.println(Character.*toUpperCase*('a'));// 'A'  System.***out***.println(Character.*toLowerCase*('A'));// 'a'  }  } |

## 数组相关类

### Arrays类

Arrays类包含了一系列静态方法，用于管理和操作数组。

|  |  |
| --- | --- |
| **Arrays方法摘要** | |
| static void | **sort**(overload)排序（自然排序和定制排序） |
| static int | **binarySearch**(overload)通过二分搜索法进行查找，要求数组有序 |
| static Xxx[] | **copyOf**(overload)数组元素的复制，返回值类型和参数类型一致 |
| static Xxx[] | **copyOfRange**(overload)指定范围的数组元素的复制 |
| static void | **fill**(overload)数组元素的填充 |
| static boolean | **equals**(overload)比较两个数组元素内容是否完全一致 |
| static String | **toString**(overload)返回数组的字符串形式 |
| static<T> List<T> | **asList**(T...a) 将数组转换成final修饰的list，该list不允许添加新的元素 |

你可以使用 Arrays.toString() 和 Arrays.deepToString() 方法来打印数组。由于数组没有实现 toString() 方法，所以如果将数组传递给 System.out.println() 方法，将无法打印出数组的内容，但是 Arrays.toString() 可以打印每个元素。

## 字符串相关类

### String类

#### String类简介

**String字符串实质是字符数组**

1、String类被final修饰，该类不能被继承。

2、实现了Serializable接口，可以被序列化。序列化指将内容转换为二进制流，只有能序列化的内容才能写入到磁盘上，不同进程间的数据传输也要求数据必须能够序列化。

3、实现了Comparable接口，可以用来比较内容。

4、实现了CharSequence接口，可以获取内容的长度，可以截取字串等操作。

5、String底层是一个final修饰的char类型的数组。

6、String是一个不可变(immutable)的字符序列。

7、字符串拼接运算时，只要有变量参与运算，在编译时不能确定变量的值，那么在底层调用StringBuilder中的toString方法创建一个新的对象。

8、字符串存储在常量池中

**不可变对象**

不可变对象指对象一旦被创建，状态就不能再改变。任何修改都会创建一个新的对象，如String、Integer和其他包装类。

**创建一个包含可变对象的不可变对象**

创建一个包含可变对象的不可变对象，不要共享可变对象的引用，如果需要变化时，就返回原对象的一个拷贝。最常见的例子就是对象中包含一个日期对象的引用。

#### String实例化

直接赋值创建对象：

String s = "abc";

构造器创建对象：

String s = new String(String original);//将字符串数据封装成字符串对象

this.value = original.value;

String s = new String();//创建一个空值的字符串对象

this.value = new char[0];

String s = new String(char[] a);//将字符数据的数据封装成字符串对象

this.value = Arrays.copyOf(value, value.length);

String s = new String(char[] a,int startIndex,int count)//将字符数组中的一部分数据封装成字符串对象

常量池的特点：常量池中存放的字符串都是唯一的，不会有重复的字符串。

#### String内存原理

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  /\*  \* 1、JVM先在栈中写入一个str1的引用  \* 2、判断常量池中是否存在该字符串，如果有，将该字符串的地址值赋给引用str1，如果没有，新建一个字符串，再将地址值赋给str1  \*/  //因为之前常量池中没有该字符串，因此创建了一个对象  String str1 = "AA";  /\*  \* 1、JVM先在栈中写入一个str2的引用  \* 2、判断常量池中是否存在该字符串，因为该字符串已存在，将该字符串的地址值赋给引用str2  \*/  //因为之前常量池中有该字符串，因此没有创建对象  String str2 = "AA";  /\*  \* 1.JVM先在栈写入一个str3的引用  \* 2.在堆内存开辟了一个叫new String()内存  \* 3.把new String()地址值赋给了引用str3  \* 4.把字符串常量池里的AA的地址值存在new String()的内存里  \*/  //因为之前常量池中有该字符串，因此只创建了new String()一个对象  String str3 = **new** String("AA");  //因为之前常量池中没有该字符串，因此创建了BB和new String()两个对象  String str4 = **new** String("BB");    System.***out***.println(str1 == str2);  System.***out***.println(str1.equals(str2));  System.***out***.println(str1 == str3);  //intern方法先去常量池中查找是否有目标对象，如果有，就会直接使用常量池中的对象，而不是创建新的对象。如果没有就创建新的对象。  System.***out***.println(str1 == str3.intern());  System.***out***.println(str1.equals(str3));  }  } |
| true  true  false  true  true |

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String []args) {  String a = "abc";  String b = "ab";  **final** String c = "c";  //该运算会查找常量池中是否存在内容为"abc"字符串对象，如存在则直接让e引用该对象  String e = "ab"+"c";  System.***out***.println(a==e);  //涉及到变量的相加，会生成新的对象，其内部实现是先new一个StringBuilder，然后 append(b),append(c);然后让d引用toString()返回的对象，因此结构不相等  String d = b + "c";  System.***out***.println(a==d);  //如果参与运算的变量是被final修饰的，则为常量，最后的运算结果和非变量运算结果相同  String f = "ab"+c;  System.***out***.println(a.equals(f));  }  } |
| true  false  true |

#### **String常用方法**

|  |  |
| --- | --- |
| **String方法摘要** | |
| char | **charAt**(int index)返回指定位置字符 |

|  |
| --- |
| **public** **void** method() {  String string = "abcdefghijklmnopqrstuvwxyz";  System.***out***.println(string.charAt(4));  } |
| e |

|  |  |
| --- | --- |
| **String方法摘要** | |
| int | **length**()字符串长度 |
| int | **compareTo**(String anotherString)两个字符串的长度相减 |
| int | **indexOf**(String s)从前向后查找获取s在当前字符串的索引位置 |
| int | **indexOf**(String s ,int startpoint)从前向后查找,从指定的位置向后查找，获取s在当前字符串的索引位置 |
| int | **lastIndexOf**(String s)从后向前查找获取s在当前字符串的索引位置 |
| int | **lastIndexOf**(String s ,int startpoint)从后向前查找,从指定的位置向前查找，获取s在当前字符串的索引位置 |

|  |
| --- |
| **public** **void** method2() {  String string = "abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz";  System.***out***.println(string.length());  System.***out***.println(string.indexOf("cde"));  System.***out***.println(string.indexOf("lmn", 26));  System.***out***.println(string.lastIndexOf("cde"));  System.***out***.println(string.lastIndexOf("lmn",26));  } |
| 52  2  37  28  11 |

|  |  |
| --- | --- |
| **String方法摘要** | |
| boolean | **equals**(Object anObject)将此字符串与指定的对象比较，严格区分大小写 |
| boolean | **equalsIgnoreCase**(String anStr)比较时忽略大小写 |
| boolean | **startsWith**(String prefix)当前字符串是否是以prefix开头 |
| boolean | **endsWith**(String suffix)当前字符串是否是以suffix结尾 |
| boolean | **contains**(CharSequence s)当前字符串中是否包含s |
| boolean | **regionMatches**(int firstStart,String other,int otherStart ,int length)测试两个字符串的区域是否相等。（另一个字符串的区域是否是当前字符串的开头），第一个参数：从当前字符串的哪个位置开始比较，第二个参数：另一个字符串，第三个参数：从另一个字符串的哪个位置开始，第四个传参数：从另一个字符串的开始位置的长度 |
| boolean | **matches**(String regex)判断该字符串是否匹配给定的字符串，和Pattern类的静态方法**matches**功能相同 |

|  |
| --- |
| **public** **void** method3() {  String string = "abcdefghijklmnopqrstuvwxyz";  System.***out***.println(string.equals("abcdefghijklmnopqrstuvwxyz"));  System.***out***.println(string.equalsIgnoreCase("ABCDEfghijklmnopqrstuvwxyz"));  System.***out***.println(string.startsWith("abcde"));  System.***out***.println(string.endsWith("xyz"));  System.***out***.println(string.contains("efgh"));  System.***out***.println(string.regionMatches(3, "cde", 1, 2));  } |
| true  true  true  true  true  true |

|  |  |
| --- | --- |
| **String方法摘要** | |
| String | **substring**(int startpoint)从当前字符串的startPoint位置开始截取子串截取到最后 |
| String | **substring**(int start,int end)从当前字符串的startPoint位置开始截取子串截取到end。(包头不包尾（左闭右开）) |
| String | **toUpperCase**()转成大写 |
| String | **toLowerCase**()转成小写 |
| String | **replaceAll**(String old,String new)当前字符串中的new替换old |
| String | **replace**(char oldChar,char newChar)将字符串中的字符char1替换为char2 |
| String | **intern**()先去常量池中查找是否有该对象，如果有，就返回常量池中字符串的引用，如果没有就在常量池中创建新的字符串对象，并返回该对象的引用，因此最终都会返回常量池中字符串的引用 |
| String | **trim**()去除字符串两端的空格并返回新的字符串 |
| String | **concat**(String str)将两个字符串拼接并返回拼接后的字符串 |
| String[] | **split**(String regex)根据给定正则表达式的匹配拆分此字符串 |
| char[] | **toCharArray**()将此字符串转换为一个字符数组 |
| byte[] | **getBytes**(overload)将字符串转为字节数组，并可以传入编码集指定字节数的的编码集 |
| static String | **valueOf**(Xxx xxx)返回参数xxx的字符串表示形式 |

|  |
| --- |
| **public** **void** method4() {  String string = "ABCDefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz";  System.***out***.println(string.substring(26));  System.***out***.println(string.substring(26, 32));  System.***out***.println(string.toLowerCase());  System.***out***.println(string.toUpperCase());  System.***out***.println(string.replaceAll("xyz", "000"));  System.***out***.println(string.replace('z', '0'));  String string2 = " a a a a ";  System.***out***.println(string2.trim());  System.***out***.println(string2.concat(string2));  String string3 = "a-a-a-a-a";  String[] strings = string3.split("-");  **for**(String string4:strings) {  System.***out***.print(string4);  }  } |
| abcdefghijklmnopqrstuvwxyz  abcdef  abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz  ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ  ABCDefghijklmnopqrstuvw000abcdefghijklmnopqrstuvw000  ABCDefghijklmnopqrstuvwxy0abcdefghijklmnopqrstuvwxy0  a a a a  a a a a a a a a  aaaaa |

#### 字符、字节数组转换

将byte[]转换为String时，可以使用 String 接收 byte[] 参数的构造器来进行转换，需要注意的点是要使用的正确的编码，否则会使用平台默认编码，这个编码可能跟原来的编码相同，也可能不同。

|  |
| --- |
| @Test  **public** **void** method() {  String str = "abcdefgh";  // 将字符串转成字符数组  **char**[] charArr = str.toCharArray();  System.***out***.println(Arrays.*toString*(charArr));  // 从字符串起始位置开始，到  str.getChars(0, 1, charArr, charArr.length-1);  System.***out***.println(Arrays.*toString*(charArr));  // 将字符数组转成字符串  String str2 = **new** String(charArr);  System.***out***.println(str2);  // 从字符数组0的位置开始，获取长度为3的字符串  String str3 = **new** String(charArr,0,3);  System.***out***.println(str3);  // 将字符串转成Byte[]数组  **byte**[] bytes = str.getBytes();  System.***out***.println(Arrays.*toString*(bytes));  // 将Byte[]数组转成字符串  String str4 = **new** String(bytes);  System.***out***.println(str4);  } |
| [a, b, c, d, e, f, g, h]  [a, b, c, d, e, f, g, a]  abcdefga  abc  [97, 98, 99, 100, 101, 102, 103, 104]  abcdefgh |

### StringBuffer类和StringBuilder类

#### 简介

StringBuffer和StringBuilder是可变的字符序列，可以对字符串进行增删，很多方法与String相同，但两者是可变长度的。StringBuffer是一个容器，StringBuffer是一个初始容量为16的字符串缓冲区。StringBuilder 和 StringBuffer 非常类似，均代表可变的字符序列，而且方法也一样。两者均位于java.lang包下。

**String、StringBuffer和StringBuilder的区别是什么？**

**String**：一个不可变的字符序列，底层是一个被final修饰的char[]数组，效率低，但是复用率高。

**String效率低的原因**

|  |
| --- |
| String s = "a";  s = s + "b"; |

实际上"a"字符串对象已经丢弃了，现在又产生一个字符串s + "b"（也就是"ab")。如果多次执行改变内容的操作，会产生大量副本字符串对象存留在内存中，降低效率。这样的操作放到循环中，会极大影响程序的性能。

**StringBuffer**：一个可变长度的字符序列，底层是一个没有final修饰的char[]数组，继承了AbstractStringBuilder，**线程安全的**，但是效率相比StringBuilder略低。

**StringBuilder**：一个可变长度的字符序列，底层是一个没有final修饰的char[]数组，继承了AbstractStringBuilder，**线程不安全的**，但是效率相比StringBuffer略高。

**String、StringBuffer和StringBuilder的选择？**

如果对字符串的增删操作多，使用StringBuilder或StringBuffer，若要线程安全，使用 StringBuffer， 否则用 StringBuilder。

如果只是为了读操作，选择使用String。

String和StringBuffer主要区别是性能：String是不可变对象，每次对String类型进行操作都等同于产生了一个新的String对象，然后指向新的String对象。所以尽量不在对String进行大量的拼接操作，否则会产生很多临时对象，导致GC开始工作，影响系统性能。

StringBuffer是对对象本身操作，而不是产生新的对象，因此在有大量拼接的情况下，我们建议使用StringBuffer。

但是需要注意现在JVM会对String拼接做一定的**优化**：  
String s=“This is only ”+”simple”+”test”会被虚拟机直接优化成String s=“This is only simple test”，此时就不存在拼接过程。

#### StringBuffer方法

|  |  |
| --- | --- |
| **StringBuffer方法摘要** | |
| StringBuffer | **append**(overload)向StringBuffer的尾部添加新的内容 |
| StringBuffer | **insert**(overload)在指定位置插入字符串 |
| StringBuffer | **reverse**()将字符串内容反转并返回新的字符串 |
| StringBuffer | **delete**(int start,int end)删除指定范围内的字符串，包头不包尾 |
| char | **charAt**(int index)获取指定位置的字符 |
| void | **setCharAt**(int index,char ch)将指定位置的字符替换为新字符 |
| StringBuffer | **replace**(int start,int end,String str)将指定范围内的字符串替换为新字符串，包头不包尾 |
| int | **indexof**(overload)查询指定内容在字符串中的位置 |
| String | **substring**(overload)截取指定范围内的字符串，返回String型字符串，包头不包尾 |
| int | **length**()获取字符串长度（实际值） |
| int | **capacity**()返回当前容量（理论值） |

|  |
| --- |
| **public** **void** method() {  StringBuffer stringBuffer = **new** StringBuffer("abcdefgli");  //返回值：StringBuffer；添加新的内容  stringBuffer.append("00000");  System.***out***.println(stringBuffer);  //返回值：StringBuffer；在指定位置插入字符串  stringBuffer.insert(2,"2b");  System.***out***.println(stringBuffer);  //返回值：StringBuffer；字符串反转  stringBuffer.reverse();  System.***out***.println(stringBuffer);  //返回值：StringBuffer；将数组中索引值从3到5位置的所有内容全部删除，包头不包尾  stringBuffer.delete(3,5);  System.***out***.println(stringBuffer);  //返回值：char；获取指定位置上的字符  System.***out***.println(stringBuffer.charAt(3));  //返回值：void；将当前字符串指定位置的字符替换为新字符  stringBuffer.setCharAt(3,'\*');  System.***out***.println(stringBuffer);  //返回值：StringBuffer；将当前字符串7位置到10位置上的所有字符串替换为新字符串"$"，包头不包尾  stringBuffer.replace(7,10,"$");  System.***out***.println(stringBuffer);  //返回值：int；从前向后查找"\*"在当前字符串中的位置  System.***out***.println(stringBuffer.indexOf("\*"));  //返回值：String：将当前字符串中从5到10上的内容截取成新的字符串，包头不包尾  System.***out***.println(stringBuffer.substring(5,10));  //返回值：int；字符串内容长度  System.***out***.println(stringBuffer.length());  } |
| abcdefgli00000  ab2bcdefgli00000  00000ilgfedcb2ba  000ilgfedcb2ba  i  000\*lgfedcb2ba  000\*lgf$b2ba  3  gf$b2  12 |

#### 构造器

|  |
| --- |
| **public** **void** method() {  //参数为字符串：底层创建长度为"abcde".length()+16的char数组，再用append方法添加"abcde"  StringBuffer stringBuffer1 = **new** StringBuffer("abcde");  //参数为数字：底层创建一个指定长度为10的char数组  StringBuffer stringBuffer2 = **new** StringBuffer(10);  //参数为空：构造没有字符的字符串缓冲区，并构造了默认长度为16的char数组  StringBuffer stringBuffer3 = **new** StringBuffer();  } |

#### 扩容

通过空参的构造器创建对象时，底层默认会创建一个长度为16的char数组。当向数组中添加第17个元素时，底层会扩容，扩容（新建一个数组）为原来数组长度的2倍加2；同时将原来数组中的内容存放到新数组中。

如果需要添加的内容的长度+原内容的长度超过原来数组长度的2倍加2，创建的新数组的长度为需要添加的内容的长度+原内容的长度。

StringBuilder时间效率最高，其次StringBuffer，String时间效率最低。

|  |
| --- |
| **public** **void** method() {  **long** startTime,endTime;  StringBuffer stringBuffer = **new** StringBuffer();  StringBuilder stringBuilder = **new** StringBuilder();  String string = "";  startTime = System.*currentTimeMillis*();  **for** (**int** i = 0; i < 30000; i++) {  //只有超过长度超出时才会创建新的对象，而且是线程安全，故效率较高  stringBuffer.append(i);  }  endTime = System.*currentTimeMillis*();  System.***out***.println("StringBuffer使用时间为：" + (endTime - startTime));  startTime = System.*currentTimeMillis*();  **for** (**int** i = 0; i < 30000; i++) {  //StringBuilder更快是因为没有加锁，牺牲了线程安全  stringBuilder.append(i);  }  endTime = System.*currentTimeMillis*();  System.***out***.println("StringBuilder使用时间为：" + (endTime - startTime));  startTime = System.*currentTimeMillis*();  **for** (**int** i = 0; i < 30000; i++) {  //string每运算一次就创建一个新的对象  string += i;  }  endTime = System.*currentTimeMillis*();  System.***out***.println("String使用时间为：" + (endTime - startTime));  } |
| StringBuffer使用时间为：2  StringBuilder使用时间为：1  String使用时间为：1267 |

#### 和String互相转换

|  |
| --- |
| **public** **void** method4() {  //String转StringBuffer  String string = "String to StringBuffer";  StringBuffer stringBuffer = **new** StringBuffer(string);  StringBuffer stringBuffer2 = **new** StringBuffer();  stringBuffer2.append(string);  System.***out***.println(stringBuffer+"/"+stringBuffer2);  //StringBuffer转String  String string2 = stringBuffer.toString();  String string3 = **new** String(stringBuffer2);  System.***out***.println(string2+"/"+string3);  } |

### Pattern类

该类中定义了8中模式，例如多行模式、unix模式等。

|  |  |
| --- | --- |
| **Pattern类方法摘要** | |
| static Pattern | **compile**(overload)获取指定正则表达式（和模式）的Pattern对象 |
| static boolean | **matches**(String regex,CharSequence input)将input与指定正则表达式比较 |
| static String | **quote**(String s)返回指定字符串的字面值 |
| Matcher | **matcher**(CharSequence input)获取该模式的匹配器 |
| int | **flags**()获取该模式的标志 |
| String | **pattern**()获取此模式的正则表达式 |
| String[] | **split**(overload)获取分割的字符串个数，如果limit为0则全部获取，如果大于实际个数则返回实际个数 |

## 日期相关类

### Date类

#### java.util.Date

Date类的对象表示特定的瞬间，精确到毫秒。

**构造器**

Date()： 源代码this(System.currentTimeMillis())获取当前时间

Date(long date) 根据传入的毫秒数创建date对象

|  |  |
| --- | --- |
| **Date类方法摘要** | |
| void | **getTime**():返回自 1970 年 1 月 1 日 00:00:00 GMT 以来此 Date 对象表示的毫秒数。 |
| long | **toString**():把此 Date 对象转换为以下形式的 String： dow mon dd hh:mm:ss zzz yyyy 其中dow 是一周中的某一天 (Sun, Mon, Tue, Wed, Thu, Fri, Sat)， zzz是时间标准。 |
| void | **setTime**(long time)根据传入的毫秒数设置时间 |

|  |
| --- |
| **public** **void** method() {  //基准时间：1970年1月1日0时0分0秒  //空参构造器：创建date对象，并获取当前时间  Date date = **new** Date();  //输出当前日期时间  System.***out***.println(date.toString());  //获取距基准时间的时间长度，以毫秒数表示  System.***out***.println(date.getTime());  //获取距基准时间的时间长度，以毫秒数表示  System.***out***.println(System.*currentTimeMillis*());  //有参构造器：根据参数获取基于参数的时间对象  Date date2 = **new** Date(1556273354124L);  System.***out***.println(date2);  } |
| Fri Apr 26 18:48:28 CST 2019  1556275708811  1556275708818  Fri Apr 26 18:09:14 CST 2019 |

#### java.sql.Date

和数据库中的日期时间格式对应，如果要将数据库中的时间获取出来保存到对象中，那么该对象所对应的属性的类型就是java.sql.Date

|  |
| --- |
| **public** **void** method() {  //有参构造器：根据参数获取基于参数的时间对象  Date date = **new** Date(1556273354124L);  //输出当前日期时间  System.***out***.println(date.toString());  //获取距基准时间的时间长度，从结果上看等于构造器中传入的参数  System.***out***.println(date.getTime());  } |
| 2019-04-26  1556273354124 |

**将当前时间插入到数据库中**

|  |
| --- |
| **public** **void** method() {  //先利用util包中的Date获取当前时间  java.util.Date date = **new** java.util.Date();  //再利用util包中的Date获取当前时间对应的毫秒数，将其作为参数传递给sql包中的Date的构造器  java.sql.Date date2 = **new** java.sql.Date(date.getTime());  System.***out***.println(date);  System.***out***.println(date2);  } |
| Fri Apr 26 18:48:00 CST 2019  2019-04-26 |

### SimpleDateFormat类

由于Date类的格式不利于国际化，因此有了SimpleDateFormat，将Date对象的时间格式转成我们想要的时间格式，该类位于java.text包内。java.text包下的类都是用于国际化。

**SimpleDateFormat是线程不安全的**

非常不幸，DateFormat 的所有实现，包括 SimpleDateFormat 都不是线程安全的，因此你不应该在多线程序中使用，除非是在对外线程安全的环境中使用，如 将 SimpleDateFormat 限制在 ThreadLocal 中。如果你不这么做，在解析或者格式化日期的时候，可能会获取到一个不正确的结果。因此，从日期、时间处理的所有实践来说，因此要使用 joda-time 库。

|  |  |
| --- | --- |
| **SimpleDateFormat方法摘要** | |
| String | **format**(Date date)将一个Date格式化为日期/时间字符串 |
| Date | **parse**(String source)从给定的字符串的起始位置解析文本，生成一个日期 |

|  |  |
| --- | --- |
| Date and Time Pattern | Result |
| "yyyy.MM.dd G 'at' HH:mm:ss z" | 2001.07.04 AD at 12:08:56 PDT |
| "EEE, MMM d, ''yy" | Wed, Jul 4, '01 |
| "h:mm a" | 12:08 PM |
| "hh 'o''clock' a, zzzz" | 12 o'clock PM, Pacific Daylight Time |
| "K:mm a, z" | 0:08 PM, PDT |
| "yyyyy.MMMMM.dd GGG hh:mm aaa" | 02001.July.04 AD 12:08 PM |
| "EEE, d MMM yyyy HH:mm:ss Z" | Wed, 4 Jul 2001 12:08:56 -0700 |
| "yyMMddHHmmssZ" | 010704120856-0700 |
| "yyyy-MM-dd'T'HH:mm:ss.SSSZ" | 2001-07-04T12:08:56.235-0700 |
| "yyyy-MM-dd'T'HH:mm:ss.SSSXXX" | 2001-07-04T12:08:56.235-07:00 |
| "YYYY-'W'ww-u" | 2001-W27-3 |

|  |
| --- |
| **public** **void** method() **throws** ParseException {  Date date = **new** Date();  System.***out***.println(date);  SimpleDateFormat simpleDateFormat = **new** SimpleDateFormat();  //使用format(Date date)方法将date对象转换为字符串  String string = simpleDateFormat.format(date);  System.***out***.println(string);  //有参构造器，按照指定的模板创建simpledateformat对象  SimpleDateFormat simpleDateFormat2 = **new** SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSZ");  string = simpleDateFormat2.format(date);  System.***out***.println(string);  Date date2 = **new** Date();  //将字符串解析成时间一定要注意时间格式和字符串格式是否一样，格式不一样就会抛出异常  date2 = simpleDateFormat2.parse("2019-04-26T18:42:48.099+0800");  System.***out***.println(date2);  //构造器中传入的模板字符串可以自定义  SimpleDateFormat simpleDateFormat3 = **new** SimpleDateFormat("日期为yyyy年MM月dd日，时间为HH时mm分ss秒");  string = simpleDateFormat3.format(date);  System.***out***.println(string);  } |
| Fri Apr 26 18:46:46 CST 2019  19-4-26 下午6:46  2019-04-26T18:46:46.869+0800  Fri Apr 26 18:42:48 CST 2019  日期为2019年04月26日，时间为18时46分46秒 |

**将前端发来的字符串的时间插入到数据库中**

将字符串转为java.util.Date的对象，要借助于SimpleDateFormat中的parse方法。

将java.util.Date对象转成java.sql.Date对象。

|  |
| --- |
| **public** **void** method() **throws** ParseException {  java.util.Date date = **new** java.util.Date();  //将simpleDateFormat对象设置一定的模板，是时间字符串和模板格式保持一致  SimpleDateFormat simpleDateFormat2 = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  //模拟前端发来的表示时间的字符串  String string = simpleDateFormat2.format(date);  System.***out***.println(string);  SimpleDateFormat simpleDateFormat = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  //解析时要保证时间字符串和模板格式一致，否则会报java.text.ParseException: Unparseable date异常  java.util.Date date2 = simpleDateFormat.parse(string);  java.sql.Date date3 = **new** java.sql.Date(date2.getTime());  System.***out***.println(date3);  } |
| 2019-04-26 18:58:22  2019-04-26 |

### Calendar类

Calendar（日历类）是一个抽象基类，用于完成日期字段之间的相互操作，该类不能创建实例，因此要用getInstance()方法获取其子类对象

|  |  |
| --- | --- |
| **Calendar方法摘要** | |
| abstract void | **add**(int field,int amount)根据日历规则为给给定的日历字段添加或减去指定的时间量 |
| int | **get**(field)返回给定日历字段的值 |
| static Calendar | **getInstance**()使用默认时区和语言获取一个Calendar对象 |
| Date | **getTime**()返回此Date对象 |
| void | **set**(int field,int value)将给定的日历字段设置为给定值 |

|  |
| --- |
| **public** **void** method() {  //Calendar是个抽象类，无法实例化，因此通过getInstance方法获取子类的对象  Calendar calendar = Calendar.*getInstance*();  //通过getName方法可以知道获取的对象对应的类名，创建的对象是其子类GregorianCalendar的实例  System.***out***.println(calendar.getClass().getName());  //获取当前时间对应的那个月的第几天，Calendar.DAFY\_OF\_MONTH是个常量，值为5  **int** day = calendar.get(Calendar.***DAY\_OF\_MONTH***);  System.***out***.println(day);  //将当前时间对应的那个月的第几天改为5，即那个月的5号  calendar.set(Calendar.***DAY\_OF\_MONTH***, 5);  day = calendar.get(Calendar.***DATE***);  System.***out***.println(day);  //因为之间将calendar的时间改为了当月5号，此时在其基础上减一天，为4号  //如果要在其基础上加num天，就将-1改为正数的num  calendar.add(Calendar.***DAY\_OF\_MONTH***, -1);  day = calendar.get(Calendar.***DAY\_OF\_MONTH***);  System.***out***.println(day);  //获取当前时间对应的当年的第几天，DAY\_OF\_YEAR的值为6  day = calendar.get(Calendar.***DAY\_OF\_YEAR***);  System.***out***.println(day);  //通过calendar获取当前时间的date对象  Date date = calendar.getTime();  System.***out***.println(date);  } |
| java.util.GregorianCalendar  26  5  4  94  Thu Apr 04 20:42:40 CST 2019 |

### JDK8后时间API

由于Date类在引入Calendar后大部分方法被弃用，而Calendar类存在格式化（日期和时间这样的类应该是不可变的）、偏移性（Date中的年份是从1900年开始，月份是从0开始）、不能处理闰秒和线程不安全等问题，因此有了新的时间日期类。

Java8中引入了Joda-Time的精华，包含了本地日期（**LocalDate**）、本地时间（**LocalTime**）、本地日期时间（**LocalDateTime**）、时区（**ZonedDateTime**）和持续时间（**Duration**）的类。 Date 类新增了 toInstant() 方法，用于把 Date 转换成新的表示形式。

**LocalDate**、**LocalTime**、**LocalDateTime** 类是其中较重要的几个类，它们的实例是不可变的对象，分别表示使用 ISO-8601日历系统的日期、时间、日期和时间。它们提供了简单的本地日期或时间，并不包含当前的时间信息，也不包含与时区相关的信息。

**LocalDate**只包含日期，可以获取日期字段

**LocalTime**只包含时间，可以获取时间字段

**LocalDateTime**包含日期+时间，可以获取日期和时间字段

|  |  |
| --- | --- |
| **Arrays方法摘要** | |
| static void | **sort**(overload)排序（自然排序和定制排序） |

|  |  |
| --- | --- |
| **方法** | **描述** |
| now()/\*now(ZoneId zone) | 静态方法，根据当前时间创建对象/指定时区的对象 |
| of() | 静态方法，根据指定日期/时间创建对象 |
| getDayOfMonth()/getDayOfYear() | 获得月份天数(1-31) /获得年份天数(1-366) |
| getDayOfWeek() | 获得星期几(返回一个 DayOfWeek 枚举值) |
| getMonth() | 获得月份, 返回一个 Month 枚举值 |
| getMonthValue()/getYear() | 获得月份(1-12) /获得年份 |
| getHours()/getMinute()/getSecond() | 获得当前对象对应的小时、分钟、秒 |
| withDayOfMonth()/withDayOfYear()/withMonth()/withYear() | 将月份天数、年份天数、月份、年份修改为指定的值并返回新的对象 |
| with(TemporalAdjuster t) | 将当前日期时间设置为校对器指定的日期时间 |
| plusDays(),plusWeeks(）,plusMonths(),plusYears(),plusHours() | 向当前对象添加几天、几周、几个月、几年、几小时 |
| minusMonths()/minusWeeks()/minusDays()/minusYears()/minusHours() | 从当前对象减去几月、几周、几天、几年、几小时 |
| plus(TemporalAmount t)minus(TemporalAmount t) | 添加或减少一个 Duration 或 Period |
| isBefore()/isAfter() | 比较两个 LocalDate |
| isLeapYear() | 判断是否是闰年（在LocalDate类中声明） |
| format(DateTimeFormatter t) | 格式化本地日期、时间，返回一个字符串 |
| parse(Charsequence text,DateTimeFormatter t) | 将指定格式的字符串解析为日期、时间 |

|  |
| --- |
| **public** **void** method(){  //获取当前日期  LocalDate localDate = LocalDate.*now*();  //获取当前的时间  LocalTime localTime = LocalTime.*now*();  //获取当前日期和时间  LocalDateTime localDateTime = LocalDateTime.*now*();  System.***out***.println(localDate);  System.***out***.println(localTime);  System.***out***.println(localDateTime);  //of()根据给定的数据创建对应的对象  LocalDate of = LocalDate.*of*(2019, 3, 26);  System.***out***.println(of);  //获取当天是当月的第几天  **int** dayOfMonth = localDate.getDayOfMonth();  System.***out***.println(dayOfMonth);  //获取当天是当年的第几天  **int** dayOfYear = localDate.getDayOfYear();  System.***out***.println(dayOfYear);  //修改当天为当年的第30天，返回一乐新的LocalDate对象，原对象不发生改变  LocalDate withDayOfYear = localDate.withDayOfYear(30);  System.***out***.println("新对象=" +withDayOfYear);  System.***out***.println("原对象=" +localDate);  //在当前的日期的基础上加1天  LocalDate plusDays = localDate.plusDays(1);  System.***out***.println(plusDays);  //在当前的日期的基础上减1天  LocalDate minusDays = localDate.minusDays(1);  System.***out***.println(minusDays);  //判断当前的年份是否是闰年  localDate = LocalDate.*of*(2016, 1, 1);  **boolean** leapYear = localDate.isLeapYear();  System.***out***.println(leapYear);  LocalDate of2 = LocalDate.*of*(2019, 3, 26);  LocalDate of3 = LocalDate.*of*(2018, 3, 26);  //of2的日期是否是在of3之间  System.***out***.println(of2.isBefore(of3));  //of2的日期是否是在of3之后  System.***out***.println(of2.isAfter(of3));  } |
| 2019-04-26  20:47:37.133  2019-04-26T20:47:37.133  2019-03-26  26  116  新对象=2019-01-30  原对象=2019-04-26  2019-04-27  2019-04-25  true  false  true |

### Instant类

java.time包通过值类型Instant（瞬时）提供机器视图，不提供处理人类意义上的时间单位。Instant表示时间线上的一点，而不需要任何上下文信息。例如，时区。概念上讲，它只是简单的表示自1970年1月1日0时0分0秒（UTC）开始的秒数。因为java.time包是基于纳秒计算的，所以Instant的精度可以达到纳秒级。

1秒 = 1000毫秒 =10^6微秒=10^9纳秒。

|  |  |
| --- | --- |
| **方法** | **描述** |
| **now()** | 静态方法，返回默认UTC时区的Instant类的对象 |
| **ofEpochMilli(long epochMilli)** | 静态方法，返回在1970-01-01 00:00:00基础上加上指定毫秒数之后的Instant类的对象 |
| **atOffset(ZoneOffset offset)** | 结合即时的偏移来创建一个 OffsetDateTime |
| **toEpochMilli()** | 返回1970-01-01 00:00:00到当前时间的毫秒数，即为时间戳 |

时间戳（基准时间）是指格林威治时间1970年01月01日00时00分00秒(北京时间1970年01月01日08时00分00秒)起至现在的总秒数。

|  |
| --- |
| **public** **void** test(){  //创建对象 - 获取的是中时区的时间（UTC）  Instant instant = Instant.*now*();  System.***out***.println(instant);  //根据毫秒数获取对应的时间  Instant instant2 = Instant.*ofEpochMilli*(123123123132L);  System.***out***.println(instant2);  //结合即时的偏移来创建一个 OffsetDateTime(可以理解成在中时区的基础上加减时间)  OffsetDateTime atOffset = instant.atOffset(ZoneOffset.*ofHours*(8));  System.***out***.println(atOffset);  //1556264534121 - 根据当前的时间（中时区的时间）获取对应的毫秒数  **long** epochMilli = instant.toEpochMilli();  System.***out***.println(epochMilli);  } |
| 2019-04-26T12:51:09.833Z  1973-11-26T00:52:03.132Z  2019-04-26T20:51:09.833+08:00  1556283069833 |

## Optional类

为了解决空指针异常，引入了Optional类。Optional实际上是个容器，可以存储类型T的值，或者仅仅存储null，Optional提供很多方法避免进行空指针检测

Optional<T>类(java.util.Optional)是一个容器类，代表一个值存在或不存在，原来用null表示一个值不存在，现在 Optional 可以更好的表达这个概念。并且可以避免空指针异常。

|  |  |
| --- | --- |
| **方法摘要** | |
| static<T> Optional<T> | **empty**()创建一个空的 Optional 实例 |
| T | **get**()如果调用对象包含值，返回该值，否则抛异常 |
| void | **isPresent**()判断是否包含值 |
| static<T> Optional<T> | **of**(T value)创建一个 Optional 实例 |
| static<T> Optional<T> | **ofNullable**(T t)若 t 不为 null,创建 Optional 实例,否则创建空实例 |
| <U> Optional<U> | **map**(Function<? super T,? extends U> mapper)如果有值对其处理，并返回处理后的Optional，否则返回 Optional.empty() |
| <U> Optional<U> | **flatMap**(Function mapper)与 map 类似，要求返回值必须是Optional |
| T | **orElse**(T t)如果调用对象包含值，返回该值，否则返回t |
| T | **orElseGet**(Supplier s)如果调用对象包含值，返回该值，否则返回 s 获取的值 |

|  |
| --- |
| **public** **void** test(){  String str = "wangge";  //创建对象  Optional ofNullable = Optional.*ofNullable*(str);  //isPresent() 用来判断容器中的数据是否为null  **if**(ofNullable.isPresent()){  //获取容器中的数据  System.***out***.println(ofNullable.get());  }**else**{  System.***out***.println("容器中的数据为null");  }  //如果容器中的数据为null则输出"aaa".如果容器中的数据不为null再输出原数据，该方法底层是三元运算符  Object obj = ofNullable.orElse("aaa");  System.***out***.println(obj);  } |
| wangge  wangge |

|  |
| --- |
| **public** **void** test2(){  //创建对象的另外另种方式  Optional empty = Optional.*empty*(); //创建一个空的对象(几乎不用)  String str = "aaa";  Optional of = Optional.*of*(str); //不能进行数据为null的检查（几乎也不用）  System.***out***.println(of);  } |
| Optional[aaa] |

## 数学相关类

### Math类

Math类位于java.lang包中，提供了一系列静态方法用于科学计算，其方法的参数和返回值类型一般为double型。

|  |  |
| --- | --- |
| **Math方法摘要** | |
| static xxx | **abs**(overload)绝对值，返回值类型和参数类型一致 |
| static double | **pow**(double a,double b)求幂，返回第一个参数的第二个参数次幂的值 |
| static double | **ceil**(double a)向上取整/天花板 |
| static double | **floor**(double a)向下取整/地板 |
| static xxx | **round**(overload)四舍五入，参数为double返回long，float返回int；原理时将传入的数值加上0.5后向下取整 |
| static double | **sqrt**(double a)求开方 |
| static double | **random**()求随机数 |
| static xxx | **max**(overload)求两个数的最大值，返回值类型和参数类型一致 |
| static xxx | **min**(overload)求两个数的最小值，返回值类型和参数类型一致 |

除了Math类中的random方法可以获取随机数外，java.util.Random类也可以通过实例化创建一个随机数生成器。

|  |
| --- |
| **public** **void** test(){  //该方法返回传递参数的绝对值  **int** abs = Math.*abs*(-5);  System.***out***.println(abs);  //该方法比较传入两个参数的大小，并将较大的数作为返回值返回  **double** max = Math.*max*(12.3, 15.6);  System.***out***.println(max);  } |
| 5  15.6 |

### BigInteger类

BigInteger可以理解为Integer类的升级版，Integer作为int的包装类，能存储的最大整型值为2^31-1,有时不能满足需求。BigInteger类的数值范围较Integer类和Long类的数值范围要大得多，可以支持任意精度的整数。

|  |  |
| --- | --- |
| **BigInteger方法摘要** | |
|  | **BigInteger**(overload)根据传入的参数将其转换为BigInteger对象 |
| BigInteger | **abs**()返回当前BigInteger的绝对值 |
| BigInteger | **add**(BigInteger val) 返回当前BigInteger与val相加的和 |
| BigInteger | **subtract**(BigInteger val) 返回当前BigInteger与val相减的值 |
| BigInteger | **multiply**(BigInteger val) 返回当前BigInteger与val相乘的值 |
| BigInteger | **divide**(BigInteger val) 返回当前BigInteger与val相除的值 |
| BigInteger | **remainder**(BigInteger val) 返回当前BigInteger与val相除的余数 |
| BigInteger[] | **divideAndRemainder**(BigInteger val) 返回两个元素的BigInteger数组，包含相除的结果和余数 |

|  |
| --- |
| **public** **void** test2(){  **int** number = Integer.***MAX\_VALUE*** + 1;  System.***out***.println(number);  BigInteger a = **new** BigInteger(String.*valueOf*(Integer.***MAX\_VALUE***));  BigInteger b = **new** BigInteger(String.*valueOf*(1));  BigInteger sum = a.add(b);  System.***out***.println(sum);  System.***out***.println(Integer.***MAX\_VALUE***);  } |
| -2147483648  2147483648  2147483647 |

### BigDecimal类

一般的Float类和Double类可以用来做科学计算或工程计算，但是在商业计算价格中，要求数字精度比较高，此时Float类和Double类已经不能满足需求，故用到BigDecimal类，该类在java.math包中，支持任何精度的定点数。

|  |  |
| --- | --- |
| **BigDecimal方法摘要** | |
|  | **BigDecimal**(int|double|long|String)根据传入的参数将其转换为BigInteger对象 |
| BigDecimal | **add**(overload)求和 |
| BigDecimal | **subtract**(overload)求差 |
| BigDecimal | **multiply**(overload)求积 |
| BigDecimal | **divide**(overload)求除数 |

BigDecimal初始化时使用基本数据类型进行初始化仍然会产生精度丢失的问题，只有使用**String类型的构造器参数**进行初始化才能避免精度丢失。

|  |
| --- |
| // 1 数值的初始化 BigDecimal b1 = new BigDecimal(0.01f); BigDecimal b2 = new BigDecimal(0.01d); BigDecimal b3 = new BigDecimal("0.01"); |

BigDecimal之间进行比较大小时，需要使用compareTo()方法，而不能使用算术运算符进行比较大小。

|  |
| --- |
| // 2 数值的比较 int i = b1.compareTo(b2);// -1 0 1 |

BigInteger和BigDecimal是用对象表示数据的，底层是用字符串存储数据的，因此无法使用“算术运算符”进行算术运算，只能调用add等方法完成计算，且每次进行运算的结果会被封装为新的BigDecimal对象返回。而float,double,int,long等是基本数据类型，可以直接用算术运算符运算，但是有存储范围有限以及精度不足的问题。

|  |
| --- |
| // 3 数值的运算 BigDecimal b4 = new BigDecimal("6"); BigDecimal b5 = new BigDecimal("7"); BigDecimal add = b4.add(b5);// + BigDecimal subtract = b4.subtract(b5);// - BigDecimal multiply = b4.multiply(b5);// \*  //第一个参数表示被除数，第二个参数表示运算结果保留的小数个数，第三个参数表示求近似值的具体规则 BigDecimal divide = b4.divide(b5,2,BigDecimal.*ROUND\_HALF\_DOWN*);// / |

BigDecimal进行求近似数的运算时，可以调用setScale()方法来实现该功能，在setScale()方法的参数中传递具体的求近似数的规则。

**8种舍入模式如下**

**1、ROUND\_UP**

舍入远离零的舍入模式。在丢弃非零部分之前始终增加数字(始终对非零舍弃部分前面的数字加1)。注意，此舍入模式始终不会减少计算值的大小。

**2、ROUND\_DOWN**

接近零的舍入模式。在丢弃某部分之前始终不增加数字(从不对舍弃部分前面的数字加1，即截短)。注意，此舍入模式始终不会增加计算值的大小。

**3、ROUND\_CEILING**

接近正无穷大的舍入模式。如果 BigDecimal 为正，则舍入行为与 ROUND\_UP 相同;如果为负，则舍入行为与 ROUND\_DOWN 相同。注意，此舍入模式始终不会减少计算值。

**4、ROUND\_FLOOR**

接近负无穷大的舍入模式。如果 BigDecimal 为正，则舍入行为与 ROUND\_DOWN 相同;如果为负，则舍入行为与 ROUND\_UP 相同。注意，此舍入模式始终不会增加计算值。

**5、ROUND\_HALF\_UP**

向“最接近的”数字舍入，如果与两个相邻数字的距离相等，则为向上舍入的舍入模式。如果舍弃部分 >= 0.5，则舍入行为与 ROUND\_UP 相同;否则舍入行为与 ROUND\_DOWN 相同。注意，这是我们大多数人在小学时就学过的舍入模式(四舍五入)。

**6、ROUND\_HALF\_DOWN**

向“最接近的”数字舍入，如果与两个相邻数字的距离相等，则为上舍入的舍入模式。如果舍弃部分 > 0.5，则舍入行为与 ROUND\_UP 相同;否则舍入行为与 ROUND\_DOWN 相同(五舍六入)。

**7、ROUND\_HALF\_EVEN**

向“最接近的”数字舍入，如果与两个相邻数字的距离相等，则向相邻的偶数舍入。如果舍弃部分左边的数字为奇数，则舍入行为与 ROUND\_HALF\_UP 相同;如果为偶数，则舍入行为与 ROUND\_HALF\_DOWN 相同。

注意，在重复进行一系列计算时，此舍入模式可以将累加错误减到最小。此舍入模式也称为“银行家舍入法”，主要在美国使用。四舍六入，五分两种情况。如果前一位为奇数，则入位，否则舍去。以下例子为保留小数点1位，那么这种舍入方式下的结果。

1.15>1.2 1.25>1.2

**8、ROUND\_UNNECESSARY**

断言请求的操作具有精确的结果，因此不需要舍入。如果对获得精确结果的操作指定此舍入模式，则抛出ArithmeticException。

|  |
| --- |
| // 4 约等于 BigDecimal add1 = b1.add(b2);  // 第一个参数表示运算结果保留的小数个数，第二个参数表示求近似值的具体规则 BigDecimal bigDecimal = add1.setScale(4, BigDecimal.*ROUND\_HALF\_DOWN*); |

### DecimalFormat类

该类位于java.text包下，用于对数字进行格式化。

## 比较器相关接口

### Comparable接口

自然排序：自定义类通过实现Comparable来实现自身对象的有序性，通过重写compareTo方法来实现比较两个对象的大小。

compareTo方法如果返回0，则表示两个对象的大小相等；如果方法返回值为0，则表示两个对象的大小相等；如果返回值大于0，则表示当前对象大于被比较对象；如果返回值小于0，则表示当前对象小于被比较对象。

实现Comparable接口的对象列表（和数组）可以通过 Collections.sort（和 Arrays.sort）进行自动排序。实现此接口的对象可以用作有序映射中的键或有序集合中的元素，无需指定比较器。

**Comparable的典型实现**

**BigDecimal**、**BigInteger** 以及所有的数值型对应的包装类：按它们对应的数值大小进行比较

**Character**：按字符的 unicode值来进行比较

**Boolean**：true 对应的包装类实例大于 false 对应的包装类实例

**String**：按字符串中字符的 unicode 值进行比较

**Date**、**Calendar**：后边的时间、日期比前面的时间、日期大

|  |  |
| --- | --- |
| **方法摘要** | |
| int | [**compareTo**](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/lang/Comparable.html#compareTo(T))([T](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/lang/Comparable.html) o)比较此对象与指定对象的顺序。 |

### Comparator接口

定制排序：一般通过匿名内部类的方式实现Comparator接口，并重写compare方法，其他内容和Comparable相同。Comparable 总是只有一个，但是可以有多个 comparator 来定义对象的顺序。

当元素的类型没有实现java.lang.Comparable接口而又不方便修改代码，或者实现了java.lang.Comparable接口的排序规则不适合当前的操作，那么可以考虑使用 Comparator 的对象来排序。

|  |  |
| --- | --- |
| **方法摘要** | |
| int | [**compare**](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/lang/Comparable.html#compareTo(T))([T](mk:@MSITStore:D:\Eclipse\JDK_API_1_6_zh_CN.CHM::/java/lang/Comparable.html) o1,T o2) 比较用来排序的两个参数。 |
| boolean | **equals**(Object obj) 指示某个其他对象是否“等于”此 Comparator。 |

在使用Arrays.sort(数组)或Collections.sort(集合)方法时，TreeSet和TreeMap元素默认按照Comparable比较规则排序；也可单独为使用Arrays.sort(数组)或Collections.sort(Collection集合)方法的TreeSet和TreeMap指定Comparator定制比较器对象。

## 序列化相关接口

### 序列化机制

序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化，即把对象的内容转成二进制流。可以对流化后的对象进行读写操作（本地读写），也可将流化后的对象传输于网络之间（网络传输）。序列化是为了解决对象流读写操作时可能引发的问题（如果不进行序列化可能会存在数据乱序的问题）。

序列化：用ObjectOutputStream类将一个具体的对象进行持久化，写入到硬盘上。

反序列化：用ObjectInputStream类读取基本类型数据或对象。

序列化是RMI（Remote Method Invoke–远程方法调用）过程的参数和返回值都必须实现的机制，而RMI 是JavaEE的基础，因此序列化机制是JavaEE平台的基础。

如果需要让某个对象支持序列化机制，则必须让其类是可序列化的，为了让某个类是可序列化的，该类必须实现Serializable或Externalizable接口。

**实现序列化的步骤**

（1）实现Serializable接口或Externalizable接口，并且视情况而定指定一个序列化版本ID（serialVersionUID）值；而且要保留公共的无参构造。

（2）如果某个对象的属性也是引用数据类型，那么该数据类型也要实现Serializable接口或Externalizable接口；

（3）如果要序列化，则使用一个输出流来构造一个对象输出流ObjectOutputStream并通过writeObject(Object obj)方法就可以将实现对象写出(即保存其状态)；如果需要反序列化则可以用一个输入流建立对象输入流ObjectInputStream，然后通过readObject方法从流中读取对象。

（4）如果某些属性不参与序列化，如果是实现Serializable接口的，直接在属性前面加transient修饰，注意：static修饰的属性也不会被序列化，如果是实现Externalizable接口，那么只要在重写writeExternal()和readExternal()方法时，不处理该属性即可。

### Serializable接口

该接口是一个没有方法的标记接口，用于给类指定一个serialVersionUID，该serialVersionUID是通过实现接口类中的可序列化成员的数字签名运算并自动生成的一个long类型的值，只要这些成员没有变化，那么该serialVersionUID值每次运算都一样。

显式提供一个serialVersionUID，用于在反序列化时和序列化时的serialVersionUID进行比较是否一致，如果一致才可以反序列化，则表示能够兼容。否则会报异常。

如果在序列化对象所在的类中没有显式的提供一个serialVersionUID，那么在序列化时将会使用一个系统提供的serialVersionUID。那么在序列化后，磁盘上的对象也会记录下这个serialVersionUID。当我们反序列化时如果对类进行了修改，那么原来的serialVersionUID将会发生改变，导致无法反序列化。

如果显式提供了serialVersionUID，那么我们在反序列化时对类进行修改也不会改变serialVersionUID的值，可以继续反序列化。

对象所在的类中的**属性除基本数据类型外**都需要实现Serializable接口。

**注意**

1. 类变量不会被序列化，因为类变量不在堆中，而是存储在方法区中
2. 对于实例变量且不想被序列化的属性而言，可以使用transient关键字修饰

### Externalizable接口

实现该接口时还必须实现readExternal()和writeExternal()这两个方法，不同于实现Serializable接口无须实现任何方法。

### Serializable与Externalizable的区别

Serializable 接口是一个序列化 Java 类的接口，以便于它们可以在网络上传输或者可以将它们的状态保存在磁盘上，是 JVM 内嵌的默认序列化方式，成本高、脆弱而且不安全。Externalizable 允许你控制整个序列化过程，指定特定的二进制格式，增加安全机制。

# JDBC

JDBC（java database connectivity）是sun公司指定的java数据库连接技术的简称，其作用是作为一个API编程接口，制定了统一的Java程序访问所有种类数据库的规范。

在JDBC出现之前，Java程序访问MySQL、DB2和Oracle等不同的数据库需要编写不同的访问代码，较为繁琐。为了解决这个问题，就有了JDBC，为针对数据库的访问提供统一规范。

**JDBC的优化**

优先使用批量操作来插入和更新数据

使用PreparedStatement来避免SQL漏洞

使用数据连接池

通过列名来获取结果集

## JDBC常用类

### DriverManager类

该类位于java.sql包下，作为驱动管理器，其作用是注册JDBC驱动和获取数据库连接

|  |  |
| --- | --- |
| **DriverManager方法摘要** | |
| static void | **registerDriver**()注册JDBC驱动 |
| static Connection | **getConnection**(overload)获取数据库连接 |

### Connection接口

该接口位于java.sql包下，用于获取数据库连接

|  |  |
| --- | --- |
| **Connection方法摘要** | |
| Statement | **createStatement**(overload)创建Statement（sql执行器）对象 |
| PrepareStatement | **prepareStatement**(overload)预编译sql语句，创建PreparedStatement对象 |

### Statement接口

SQL执行器，可以通过Connection的createStatement()方法获取Statement的实现类对象，获取时不需要传入sql语句。

|  |  |
| --- | --- |
| **Statement方法摘要** | |
| int | **executeUpdate**(String sql)执行insert、delete和update语句，并返回int型返回值 |
| ResultSet | **executeQuery**(String sql)执行查询语句，并返回ResultSet结果集 |

因为该接口存在SQL注入的风险，为了解决这个问题就有了PreparedStatement接口。

SQL注入指的是在输入的字符串中添加admin' or 1 = 1 or 2 = '使拼接成的SQL语句执行结果始终为true，从而避免了密码、用户名等核查，导致系统被破解。

#### PreparedStatement接口

SQL执行器（预编译后），是Statement接口的子类，可以通过Connection的prepareStatement()方法获取PreparedStatement实现类的对象，获取时需要传入sql语句，此时可以对sql语句进行预编译。

|  |  |
| --- | --- |
| **Statement方法摘要** | |
| int | **executeUpdate**()执行insert、delete和update语句，并返回int型返回值 |
| ResultSet | **executeQuery**()执行查询语句，并返回ResultSet结果集 |

**PreparedStatement和statement的区别**

1、Statement在创建的时候不需要sql语句，而 PreparedStatement需要。

2、Statement在调用executeUpdate和executeQuery方法时都需要sql语句做为参数； PreparedStatement在调用executeUpdate和executeQuery方法时都不需要sql语句做为参数。

3、PreparedStatement它可以执行带有占位符?的sql语句；statement不行。

4、PreparedStatement安全；statement不安全（有sql注入安全问题）。

5、PreparedStatement在创建的时候已经预编译了sql语句，性能上会比statement高一点点。

### ResultSet接口

ResultSet是结果集，可以理解为查询得到的数据库表。

|  |  |
| --- | --- |
| **ResultSet方法摘要** | |
| Xxx | **getXxx**(overload)获取指定类型的数据 |
| boolean | **next**()结果集光标向下移动一行 |
| boolean | **previous**()结果集光标向上移动一行 |
| boolean | **absolute**(int row)将光标移动到指定行 |
| boolean | **first**()将光标移动到第一行 |
| boolean | **last**()将光标移动到最后一行 |

ResultSet接口的getXxx方法可以获取指定类型的数据，并且有两种获取方式：

按**列的索引**获取：第一列索引值是1，第二列索引值是2，以此类推，第n列索引值是n

按**结果集的列名**获取：如果sql语句中给列名设置了别名，则需要按照该列的别名获取

## JDBC简单实例

**测试用数据库表**

**表一**

|  |
| --- |
| **drop** database if **exists** jdbcDB;  **create** database jdbcDB;  use jdbcDB;  **create** **table** t\_book(  `id` **int**(11) **primary** **key** auto\_increment, ## 主键  `name` **varchar**(50) , ## 书名  `author` **varchar**(50) , ## 作者  `price` **decimal**(11,2) , ## 价格  `sales` **int**(11) , ## 销量  `stock` **int**(11) , ## 库存  `publishing` **varchar**(200) ## 出版社  );  **insert** **into** t\_book(`name` , `author` , `price` , `sales` , `stock` , `publishing` )  **values**('java从入门到放弃' , '王振国' , 80 , 100 , 0 , '北京大学出版社' );  **insert** **into** t\_book(`name` , `author` , `price` , `sales` , `stock` , `publishing` )  **values**('数据结构与算法' , '国哥' , 78 , 87 , 13 , '上海交大出版社' );  **insert** **into** t\_book(`name` , `author` , `price` , `sales` , `stock` , `publishing` )  **values**('西游记' , '吴承恩' , 68, 0 , 100 , '清华大学出版社' );  **insert** **into** t\_book(`name` , `author` , `price` , `sales` , `stock` , `publishing` )  **values**('权力的游戏' , '马丁' , 16, 50 , 50 , '上海交大出版社' );  **insert** **into** t\_book(`name` , `author` , `price` , `sales` , `stock` , `publishing` )  **values**('地狱' , '丹布朗' , 16, 50 , 50 , '北京大学出版社' );  **insert** **into** t\_book(`name` , `author` , `price` , `sales` , `stock` , `publishing` )  **values**('水浒传' , '施耐庵' , 45 , 5 , 95 , '清华大学出版社' ); |

**表二**

|  |
| --- |
| **create** **table** t\_user(  `id` **int** **primary** **key** auto\_increment,  `username` **varchar**(20) **not** **null**,  `password` **varchar**(30) **not** **null**,  `money` **int** **default** 1000  );  **insert** **into** t\_user(`username`,`password`) **values**('admin','admin');  **insert** **into** t\_user(`username`,`password`) **values**('wzg168','666666'); |

**JDBC编程步骤：**

1. 注册JDBC驱动程序
2. 通过DriverManager获取数据库连接
3. 通过连接对象创建SQL语句执行器
4. 使用SQL语句执行器执行SQL语句
5. 获取执行结果
6. 关闭资源

|  |
| --- |
| **void** jdbctest() **throws** Exception {  //注册JDBC驱动程序，如果这里没有注册驱动，也不会影响程序执行，因为导入的JDBC包中默认注册了MySQL数据库的驱动  //但是如果连接的不是MySQL数据库，就无法使用，因此实际开发中都需要手动注册JDBC驱动程序。  // DriverManager.registerDriver(new Driver());  //因为在Driver类的静态初始化块中就已经注册了JDBC驱动，使用DriverManager调用registerDriver方法就重复注册了  //因此一般会使用反射来通过全类名直接获取Driver的对象并注册JDBC驱动  Class.*forName*("com.mysql.jdbc.Driver");  /\*  \* getConnection方法共有3个参数，url、user、password  \* url 地址  \* user 表示连接的数据库的用户名  \* password 表示连接的数据库的用户密码  \* \*/  Connection connection = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/JDBCDB", "root", "root");  //使用createStatement方法获取SQL语句执行器  Statement statement = connection.createStatement();  String sql = "select `id`,`name`,`author`,`price`,`sales`,`stock`,`publishing` from t\_book;";  //查询操作使用executeQuery方法，并将结果作为结果集ResultSet返回  ResultSet rs = statement.executeQuery(sql);  **while**(rs.next()) {  Book book = **new** Book(  rs.getInt(1),  rs.getString(2),  rs.getString("author"),  rs.getBigDecimal("price"),  rs.getInt(5),  rs.getInt(6),  rs.getString(7)  );  System.***out***.println(book);  }  //关闭资源，关闭顺序按照先打开后关闭的原则来进行关闭  rs.close();  statement.close();  connection.close();  } |

url地址：jdbc:mysql://localhost:3306/jdbcDB(?parameter = value & parameter = value...) ，位于括号内的内容是可选项，问好后面的都是参数，不同的参数对连接有不同的操作。

如果访问的服务器是localhost并且端口号为3306，则IP地址和端口号可以省略不写。

jdbc: jdbc协议

mysql:// 子协议

localhost 服务器的IP地址

:3306 服务器的端口号

/jdbcDB 访问的数据库名称

### Class.forName

按参数中指定的字符串形式的类名去搜索并加载相应的类，如果该类字节码已经被加载过，则返回代表该字节码的Class实例对象，否则，按类加载器的委托机制去搜索和加载该类，如果所有的类加载器都无法加载到该类，则抛出ClassNotFoundException。加载完这个Class字节码后，接着就可以使用Class字节码的newInstance方法去创建该类的实例对象了。

有时候程序中所有使用的具体类名在设计时（即开发时）无法确定，只有程序运行时才能确定，这时候就需要使用Class.forName去动态加载该类，这个类名通常是在配置文件中配置的，例如，spring的ioc中每次依赖注入的具体类就是这样配置的，jdbc的驱动类名通常也是通过配置文件来配置的，以便在产品交付使用后不用修改源程序就可以更换驱动类名。

## CRUD

CRUD指的是针对数据库表中数据的增删改查操作。

C create 增，添加

R read 读，查询

U update 更新，修改

D delete 删除

一般情况下。CRUD，有五个语句：

插入

insert into t\_book(`name`,`author`,`price`,`sales`,`stock`,`publishing`)

values('三国演义' , '罗贯中' , 88.88 , 99,100 , '北京大学出版社')

删除

delete from t\_book where id = 编号

更改

update t\_book set `name`='xxxx',`author`='xxxx' where id = 编号

查询全部

select `id`,`name`,`author`,`price`,`sales`,`stock`,`publishing` from t\_book

按主键查询一个

select `id`,`name`,`author`,`price`,`sales`,`stock`,`publishing` from t\_book where id = 编号

### statement

**增**

|  |
| --- |
| Statement statement = connection.createStatement();  String sql = "insert into t\_book(`name`,`author`,`price`,`sales`,`stock`,`publishing`) values('三国演义','罗贯中',88.88,99,100,'北京大学')";  **int** executeUpdate = statement.executeUpdate(sql);  System.***out***.println(executeUpdate + executeUpdate > 0?"success":"fail"); |

**删**

|  |
| --- |
| Statement statement = connection.createStatement();  String string = "delete from t\_book where `name` = '三国演义'";  **int** executeUpdate = statement.executeUpdate(string);  System.***out***.println(executeUpdate + executeUpdate > 0?"success":"fail"); |

**改**

|  |
| --- |
| Statement statement = connection.createStatement();  String string = "update t\_book set `name`='三国志',`author`='陈寿' where `name` = '三国演义'";  **int** executeUpdate = statement.executeUpdate(string);  System.***out***.println(executeUpdate + executeUpdate > 0?"success":"fail"); |

**查**

**查所有**

|  |
| --- |
| Statement statement = connection.createStatement();  String string = "select \* from t\_book";  ResultSet rs = statement.executeQuery(string);  **while**(rs.next()) {  Book book = **new** Book(  rs.getInt(1),  rs.getString(2),  rs.getString("author"),  rs.getBigDecimal("price"),  rs.getInt(5),  rs.getInt(6),  rs.getString(7)  );  System.***out***.println(book);  } |

**根据ID查询**

|  |
| --- |
| Statement statement = connection.createStatement();  String string = "select \* from t\_book where id = 2";  ResultSet rs = statement.executeQuery(string);  **while**(rs.next()) {  Book book = **new** Book(  rs.getInt(1),  rs.getString(2),  rs.getString("author"),  rs.getBigDecimal("price"),  rs.getInt(5),  rs.getInt(6),  rs.getString(7)  );  System.***out***.println(book);  } |

### PrepareStatement

**增**

|  |
| --- |
| String sql = "insert into t\_book(`name`,`author`,`price`,`sales`,`stock`,`publishing`) values(?,?,?,?,?,?)";  PreparedStatement prepareStatement = connection.prepareStatement(sql);  prepareStatement.setString(1, "三国演义");  prepareStatement.setString(2, "罗贯中");  prepareStatement.setBigDecimal(3, **new** BigDecimal(88.88));  prepareStatement.setInt(4, 400);  prepareStatement.setInt(5, 40);  prepareStatement.setString(6, "北京大学");  **int** executeUpdate = prepareStatement.executeUpdate();  System.***out***.println(executeUpdate); |

**删**

|  |
| --- |
| String sql = "delete from t\_book where `name` = ?";  PreparedStatement prepareStatement = connection.prepareStatement(sql);  prepareStatement.setString(1, "三国演义");  **int** executeUpdate = prepareStatement.executeUpdate();  System.***out***.println(executeUpdate); |

**改**

|  |
| --- |
| String sql = "update t\_book set `name`=?,`author`=? where `name` = ?";  PreparedStatement prepareStatement = connection.prepareStatement(sql);  prepareStatement.setString(1, "三国演义");  prepareStatement.setString(2, "陈寿");  prepareStatement.setString(3, "三国志");  **int** executeUpdate = prepareStatement.executeUpdate();  System.***out***.println(executeUpdate); |

**查**

**查所有**

|  |
| --- |
| String sql = "select \* from t\_book";  PreparedStatement preparedStatement = connection.prepareStatement(sql);  ResultSet rs = preparedStatement.executeQuery();  **while**(rs.next()) {  Book book = **new** Book(  rs.getInt(1),  rs.getString(2),  rs.getString("author"),  rs.getBigDecimal("price"),  rs.getInt(5),  rs.getInt(6),  rs.getString(7)  );  System.***out***.println(book);  } |

**根据ID查询**

|  |
| --- |
| String sql = "select \* from t\_book where id in (?,?,?)";  PreparedStatement preparedStatement = connection.prepareStatement(sql);  preparedStatement.setInt(1, 7);  preparedStatement.setInt(2, 6);  preparedStatement.setInt(3, 5);  ResultSet rs = preparedStatement.executeQuery();  **while**(rs.next()) {  Book book = **new** Book(  rs.getInt(1),  rs.getString(2),  rs.getString("author"),  rs.getBigDecimal("price"),  rs.getInt(5),  rs.getInt(6),  rs.getString(7)  );  System.***out***.println(book);  } |

## 批处理

批处理指的是一次执行多条SQL语句，使用批处理可以大大加快JDBC的执行速度。需要在创建数据库连接的url地址后面添加参数：**rewriteBatchedStatements = true**。

|  |
| --- |
| **void** batchTest() **throws** Exception {  Class.*forName*("com.mysql.jdbc.Driver");  //在url地址后面添加参数rewriteBatchedStatements=true  Connection connection = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/jdbcdb?rewriteBatchedStatements=true","root","root");  PreparedStatement prepareStatement = **null**;  String sql = "insert into t\_book(`name` , `author` , `price` , `sales` , `stock` , `publishing` ) "  + "values('水浒传' , '施耐庵' , 45 , ? , 95 , '清华大学出版社' )";  prepareStatement = connection.prepareStatement(sql);  **for** (**int** i = 0; i < 5000; i++) {  prepareStatement.setInt(1, 100);  //将当前SQL和数据添加到批处理中  prepareStatement.addBatch();  }  //执行批处理中的所有的SQL语句  prepareStatement.executeBatch();  prepareStatement.close();  connection.close();  } |

## 事务

例如在人员管理系统中，删除一个人员，既需要删除人员的基本资料，也要删除和该人员相关的信息，如信箱，文章等等，这样，这些数据库操作语句就构成一个事务！（所有操作要么都成功。要么都失败）。

### 事务的特性ACID

**原子性、隔离性和持久性都是为了保证一致性的手段，只有实现了这三大特性才有可能保证一致性。**

**原子性（Atomicity）：**一组事务，要么成功；要么撤回。

**利用Innodb的回滚日志（undo log）来实现原子性，当事务回滚时能够撤销所有已经成功执行的sql语句，他需要记录你要回滚的相应日志信息。**

**例如**

**(1)当你delete一条数据的时候，就需要记录这条数据的信息，回滚的时候，insert这条旧数据**

**(2)当你update一条数据的时候，就需要记录之前的旧值，回滚的时候，根据旧值执行update操作**

**(3)当年insert一条数据的时候，就需要这条记录的主键，回滚的时候，根据主键执行delete操作**

**undo log 记录了这些回滚需要的信息，当事务执行失败或调用了rollback，导致事务需要回滚，便可以利用undo log中的信息将数据回滚到修改之前的样子**

**一致性（Consistency）：**事务执行后，数据库状态与其他业务规则保持一致。

**隔离性（Isolation）：**事务独立运行。一个事务处理后的结果，影响了其他事务，那么其他事务会撤回。事务的100%隔离，需要牺牲速度。

利用锁和MVCC机制来实现隔离性。MVCC即多版本并发控制(Multi Version Concurrency Control),一个行记录数据有多个版本对快照数据，这些快照数据在undo log中。如果一个事务读取的行正在做 DELELE或者UPDATE操作，读取操 作不会等行上的锁释放，而是读取该行的快照版本。由于MVCC机制在可重复读(Repeateable Read)和读已提交(Read Commited)的MVCC表现形式不同，就不赘述了。

在事务隔离级别为读已提交(Read Commited)时，一个事务能够读到另一个事务已经提交的数据 ，是不满足隔离性的。但是当事务隔离级别为可重复读(Repeateable Read)时，是满足隔离性的。

**持久性（Durability）：事务完成后，所有操作就会写入到数据库中。**

利用Innodb的redo log来实现持久性。Mysql是先把磁盘上的数据加载到内存中，在内存中对数据进行修改，再刷回磁盘上。如果此时突然宕机，内存中的数据就会丢失。

为了解决这个问题，事务提交前需要直接把数据写入磁盘。但是只修改一个页面里的一个字节，就要将整个页面刷入磁盘，太浪费资源。毕竟一个页面16kb大小，只改其中一点点东西，就要将16kb的内容刷入磁盘，听着也不合理。毕竟一个事务里的SQL可能牵涉到多个数据页的修改，而这些数据页可能不是相邻的，也就是属于随机IO。显然操作随机IO，速度会比较慢。于是，决定采用 redo log 解决上面的问题。当做数据修改的时候，不仅在内存中操作，还会在redo log中记录这次操作。当事务提交的时候，会将redo log日志进行刷盘。( redo log一部分在内存中，一部分在磁盘上)。当数据库宕机重启的时候，会将redolog中的内容恢复到数据库中，再根据 undo log 和 binlog 内容决定回滚数据还是提交数据。

采用redo log的好处在于将 redo log 进行刷盘比对数据页刷盘效率高，具体表现如下：redo log 体积小，毕竟只记录了哪一页修改了啥，因此体积小，刷盘快；redo log 是一直往末尾进行追加，属于顺序IO。效率显然比随机IO来的快。

### 事务的安全问题

#### 读问题

脏读：指一个事务读到了另一个事务还未提交的数据

不可重复读：一个事务读到了另一个事务提交的数据，导致多次查询结果不一致

幻读：一个事务读到了另一个事务已提交的插入的数据，导致多次查询结果不一致

#### 写问题

丢失更新：指一个事务去修改数据库， 另一个事务也修改数据库，最后的那个事务，不管是提交还是回滚都会造成前面一个事务的数据更新丢失。解决丢失更新通常有两种方法: 悲观锁和乐观锁。

**悲观锁**

悲观锁指事务在一开始就认为丢失更新一定会发生，这是一件很悲观的事情。 具体操作步骤如下：

所有事务在执行操作前，先查询一次数据，查询语句如下：select \* from student for update; 后面的for update其实是数据库锁机制，一种排他锁。

哪个事务先执行这个语句，哪个事务就持有了这把锁，可以查询出来数据，后面的事务再执行这条语句，不会有任何数据显示，就只能等着。

一直等到前面的那个事务提交数据后，后面的事务数据才会出来，那么才可以往下接着操作，相当于Java中同步的概念。

**乐观锁**

乐观锁是指从来不会觉得丢失更新会发生。

要求程序员在数据库中添加字段，然后在后续更新的时候，对该字段进行判定比对，如果一致才允许更新。例子如下：

数据库表中，额外添加了一个version字段，用于记录版本，默认从0 开始，只要有针对表中数据进行修改的，那么version就+1。

开启A事务，然后开启B事务 。

A先执行数据库表操作，因为以前都没有人修改过，所以是允许A事务修改数据库的，但是修改完毕，就把version的值变成 1了 。

B事务这时候如果想执行修改，那么是不允许修改的，因为B事务以前是没有查询过数据库内容的，所以它认为数据库版本还是0，但是数据库的版本经过A修改，已经是1了，所以这时候不允许修改， 要求其重新查询。

B重新查询后，将会得到version 为1的数据，这份数据就是之前A 事务修改的数据了， B在进行修改，也是在A的基础上修改的。 所以就不会有丢失更新的情况出现了。

乐观锁的机制 ，其实是通过比对版本或者比对字段的方式来实现的，这与版本控制软件SVN , Git机制是一样的。

### ****事务的隔离级别****

**事务四种隔离级别**

**1、读未提交read uncommitted**

读未提交指的是一个事务可以读取到另一个事务还未提交的数据。这就会引发“脏读”读取到的是数据库内存中的数据，而并非真正磁盘上的数据。

1. 开启一个命令行窗口A，开始事务，然后查询表中记录。设置当前窗口的事务隔离级别为读未提交命令如下：

set session transaction isolation level read uncommitted;

2. 另外在打开一个窗口B， 也开启事务， 然后执行 sql 语句， 但是不提交

3. 在A窗口重新执行查询， 会看到B窗口没有提交的数据。

**2、读已提交read committed**

读已提交与前面的读未提交刚好相反，这个隔离级别是 ，只能读取到其他事务已经提交的数据，那些没有提交的数据是读不出来的。但是这会造成一个问题是： 前后读取到的结果不一样。 发生了不可重复!!!, 所谓的不可重复读，就是不能执行多次读取，否则出现结果不一。

1. 开启一个命令行窗口A，开始事务，然后查询表中记录。设置当前窗口的事务隔离级别为读未提交命令如下：

set session transaction isolation level read committed;

2. 另外在打开一个窗口B， 也开启事务， 然后执行 sql 语句， 但是不提交

3. 在A窗口重新执行查询， 是不会看到B窗口刚才执行sql 语句的结果，因为它还没有提交。

4. 在B窗口执行提交。

5. 在A窗口中执行查看， 这时候才会看到B窗口已经修改的结果。

6. 但是这会造成一个问题是： 在A窗口中， 第一次查看数据和第二次查看数据，结果不一样。

**3、可重复读repeatable read**

可重复读（MySql 默认的隔离级别）可以让事务在自己的会话中重复读取数据，并且不会出现结果不一样的状况，即使其他事务已经提交了，也依然还是显示以前的数据。

1. 开启一个命令行窗口A，开始事务，然后查询表中记录。设置当前窗口的事务隔离级别为读未提交命令如下：

set session transaction isolation level read committed;

2. 另外在打开一个窗口B，也开启事务，然后执行 sql 语句，但是不提交

3. 在A窗口重新执行查询，是不会看到B窗口刚才执行sql 语句的结果，因为它还没有提交。

4. 在B窗口执行提交。

5. 在A窗口中执行查看，这时候查询结果，和以前的查询结果一致，不会发生改变。

**4、串行事务serializable**

该事务级别是最高级的事务级别，比前面几种都要强大一点，也就是前面脏读、不可重复读、幻读都能够解决。但是有一些缺点。

1. 开启一个命令行窗口A，开始事务，然后查询表中记录。设置当前窗口的事务隔离级别为读未提交命令如下：

set session transaction isolation level read serializable;

2. 另外在打开一个窗口B， 也开启事务， 然后执行 sql 语句， 但是不提交

3. 在A窗口重新执行查询， 会卡主，没有任何信息显示。

4. 在B窗口执行提交。

5. 在A窗口中执行查看， 这时候才会显示结果。

Serializable可以防止上面的所有问题，但是都使用该隔离级别也会有些问题。比如造成并发的性能问题。其他的事务必须得等当前正在操作表的事务先提交，才能接着往下，否则只能一直在等着。

**由事务隔离级别产生的几个常见问题**

读未提交，可导致脏读，所谓脏读，指的是，在某一个事务中，读取到了其他事务中未提交的数据。称之为脏读。

读已提交，可导致不可重复读，指的是一个事务中，多次读取，由于其他事务进行修改操作。导致两次得到的内容不同。称之为不可重复读。

重复读，可导致幻读，指的是在一个事务中，每次读取的数据都跟第一次的数据相同，不管数据本身是否已经被其他事务所修改。称之为幻读。

### 事务的Java实现

|  |
| --- |
| **void** testACID() **throws** Exception {  PreparedStatement prepareStatement1 = **null**;  PreparedStatement prepareStatement2 = **null**;  Class.*forName*("com.mysql.jdbc.Driver");  //对于数据库而言，每次执行完一个SQL语句就会马上自动生效  Connection connection = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/jdbcdb", "root", "root");  **try** {  //设置手动管理事务  connection.setAutoCommit(**false**);  prepareStatement1 = connection.prepareStatement("update t\_user set `money` = `money` - 500 where `username` = 'admin';");  prepareStatement1.executeUpdate();  //手动设置异常使程序结束执行，进入catch中执行rollback方法  **int** i = 12/0;  prepareStatement2 = connection.prepareStatement("update t\_user set `money` = `money` + 500 where `username` = 'wzg168';");  prepareStatement2.executeUpdate();  //提交事务，如果之前没有发生异常，那么执行到commit方法才会使之前的SQL语句生效  connection.commit();  }**catch** (Exception e) {  System.***out***.println("warning!!!");  //回滚所有之前的操作，就是恢复到操作之前的状态和数值  connection.rollback();  }**finally** {  prepareStatement2.close();  prepareStatement1.close();  connection.close();  }  } |

## 数据库连接池

每次访问数据库时，都需要使用到数据库连接。在整个JDBC编程中，创建数据库连接比较耗时，那些大型网站在部署后会有大量的高并发访问，从而会有大量的访问数据库的操作。这时就需要优化数据库的访问，避免大量创建连接导致的响应速度过慢。

因此在启动网站时就创建好大量的数据库连接存放在一个集合（池pool）中，每次访问数据库需要数据库连接时就可以直接从池中获取，使用完毕后再放回池中，让用户可以继续复用，从而达到数据库访问性能优化的效果。

J2EE服务器启动时会建立一定数量的池连接，并一直维持不少于此数目的池连接。客户端程序需要连接时，池驱动程序会返回一个未使用的池连接并将其表记为忙。如果当前没有空闲连接，池驱动程序就新建一定数量的连接，新建连接的数量有配置参数决定。当使用的池连接调用完成后，池驱动程序将此连接表记为空闲，其他调用就可以使用这个连接。

实现方式，返回的Connection是原始Connection的代理，代理Connection的close方法不是真正关连接，而是把它代理的Connection对象还回到连接池中。

**连接池的优点**

1. 更快响应速度，连接池里的连接在一开始就已经创建好了，后面如果需要直接拿就可以了，无需创建。

2. 资源的重复利用、避免重复创建对象，连接对象使用完毕后，再归还到池子中进行统一管理即可。

### 开源数据库连接池

**DBCP**（**DataBase Connection Pool**）由Apache开发，通过数据库连接池，可以让程序自动管理数据库连接的释放和断开。

1. 导入jar包 commons-dbcp.jar,commons-pool.jar

2. 不使用配置文件方式

BasicDataSource dataSource = new BasicDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql://localhost/users");

dataSource.setUsername("root");

dataSource.setPassword("root");

conn = dataSource.getConnection();

String sql = "select \* from user";

pstmt = conn.prepareStatement(sql);

3. 使用配置文件方式。

1. 在 src 下定义配置文件 dbcp.properties. 内容如下：

#连接设置

driverClassName=com.mysql.jdbc.Driver

url=jdbc:mysql://localhost:3306/jdbc

username=root

password=root

2. 使用代码读取配置文件，即可获取连接池

Properties properties = new Properties();

properties.load(new FileInputStream("src/dbcp.properties"));

DataSource dataSource = BasicDataSourceFactory.createDataSource(properties);

conn = dataSource.getConnection();

String sql = "select \* from user";

pstmt = conn.prepareStatement(sql);

**C3P0** 开源的JDBC[连接池](https://baike.baidu.com/item/%E8%BF%9E%E6%8E%A5%E6%B1%A0/1699713)，它实现了数据源和JNDI绑定，支持JDBC3规范和JDBC2的标准扩展。目前使用它的[开源项目](https://baike.baidu.com/item/%E5%BC%80%E6%BA%90%E9%A1%B9%E7%9B%AE/3406069)有Hibernate，Spring等。

1. 拷贝jar 文件 c3p0-0.9.1.2.jar

2. 不使用配置文件方式

ComboPooledDataSource dataSource = new ComboPooledDataSource();

dataSource.setDriverClass("com.mysql.jdbc.Driver");

dataSource.setJdbcUrl("jdbc:mysql://localhost/users");

dataSource.setUser("root");

dataSource.setPassword("root");

conn = dataSource.getConnection();

String sql = "select \* from user";

pstmt = conn.prepareStatement(sql);

...

3. 使用配置文件方式。

> c3p0的配置文件 支持 properties , 也支持 xml 方式。 不过开发中，一般使用xml方式来配置

1. src下， 创建xml文件，名为：c3p0-config.xml 　注意此处，名字需要固定了。

<c3p0-config>

<default-config>

<property name="driverClass">com.mysql.jdbc.Driver</property>

<property name="jdbcUrl">jdbc:mysql://localhost/user</property>

<property name="user">root</property>

<property name="password">root</property>

<!-- 可选配置 -->

<property name="initialPoolSize">5</property>

</default-config>

</c3p0-config>

2. 代码中获取连接child对象

ComboPooledDataSource dataSource = new ComboPooledDataSource();

**Druid**（阿里巴巴开源代码）一个拥有大数据实时查询和分析的高容错、高性能开源分布式系统，旨在快速处理大规模的数据，并能够实现快速查询和分析。尤其是当发生代码部署、机器故障以及其他产品系统遇到宕机等情况时，Druid仍然能够保持100%正常运行。创建Druid的最初意图主要是为了解决查询延时问题，当时试图使用hadoop 来实现交互式查询分析，但是很难满足实时分析的需要。而Druid提供了以交互方式访问数据的能力，并权衡了查询的灵活性和性能二采取了特殊的存储格式。

**jdk自带数据库连接池缺点**

1. 对象没有做成单例，在哪里使用，都需要new MyDataSource(). 这就会造成有多个对象的情况出现， 那就不只一个池子了。

2. 需要额外记住 addBack方法，sun公司定义的数据库连接池里面并没有这个addBack方法，所以谁需要用我们这个连接池，需要记住这个方法是用来回收连接对象的。

3. 无法面向接口编程，由于我们的连接池直接定义成了一个类，并且里面还额外添加了一个addBack方法，这就造成了我们无法面向接口编程

### Druid的使用

1、导入Druid和JDBC包

2、在src源码目录下创建jdbc.properties属性配置文件，存放数据库的连接属性

3、使用druid连接池

#### 通过配置文件创建数据库连接池

|  |
| --- |
| **void** druidPropertiesTest() **throws** Exception {  /\*  \* 配置文件创建数据库连接池  \*/  //创建配置文件对象  Properties properties = **new** Properties();  //通过类加载器的getResourceAsStream方法获取输入流  InputStream is = DruidTest.**class**.getClassLoader().getResourceAsStream("jdbc.properties");  System.***out***.println(is);  //向配置文件对象中加载配置文件jdbc.properties中的配置信息  properties.load(is);  //通过DruidDataSourceFactory的静态createDataSource方法传入配置文件对象来创建数据库连接池  DruidDataSource dataSource = (DruidDataSource) DruidDataSourceFactory.*createDataSource*(properties);  //在数据库连接池中获取数据库连接  Connection connection = dataSource.getConnection();  PreparedStatement prepareStatement = connection.prepareStatement("select \* from t\_book");  ResultSet resultSet = prepareStatement.executeQuery();  **while**(resultSet.next()) {  System.***out***.print(resultSet.getInt("id"));  System.***out***.print(resultSet.getString("name"));  System.***out***.print("\t"+resultSet.getString("author"));  System.***out***.print("\t"+resultSet.getBigDecimal("price"));  System.***out***.print(resultSet.getString("publishing"));  System.***out***.println();  }  resultSet.close();  prepareStatement.close();  connection.close();  } |
| **配置文件（文件名： jdbc.properties；文件路径：源码目录（src）中；加载路径：类路径（bin）中）**  username=root  password=root  url=jdbc:mysql://localhost:3306/jdbcdb  driverClassName=com.mysql.jdbc.Driver  initialSize=5  maxActive=10 |

#### 通过非配置文件创建数据库连接池

|  |
| --- |
| **void** druidNonPropertiesTest() **throws** Exception {  /\*  \* 非配置文件创建数据库连接池  \*/  DruidDataSource druidDataSource = **new** DruidDataSource();  druidDataSource.setDriverClassName("com.mysql.jdbc.Driver");  druidDataSource.setUsername("root");  druidDataSource.setPassword("root");  druidDataSource.setUrl("jdbc:mysql://localhost:3306/jdbcdb");  druidDataSource.setInitialSize(5);  druidDataSource.setMaxActive(8);  System.***out***.println(druidDataSource.getConnection());  druidDataSource.close();  } |

#### Druid的检测

|  |
| --- |
| **void** druidCheckTest() **throws** Exception {  Properties properties = **new** Properties();  InputStream is = DruidTest.**class**.getClassLoader().getResourceAsStream("jdbc.properties");  properties.load(is);  DruidDataSource dataSource = (DruidDataSource) DruidDataSourceFactory.*createDataSource*(properties);  Connection connection = **null**;  //可以通过修改for循环中的数值来观察从数据库池中取出数据库连接的个数变化  **for**(**int** i = 0;i<4;i++) {  connection = dataSource.getConnection();  System.***out***.println(connection);  }  connection.close();  System.***out***.println(dataSource);  } |

## JdbcUtils工具类

### 类变量和静态初始化块

|  |
| --- |
| **static** DruidDataSource *dataSource*;  **static** {  **try** {  //通过类名创建运行是对象，从而获取类加载器，最后获取含有配置文件的输入流  InputStream is = JdbcUtils.**class**.getClassLoader().getResourceAsStream("jdbc.properties");  Properties properties = **new** Properties();  properties.load(is);  *dataSource* = (DruidDataSource) DruidDataSourceFactory.*createDataSource*(properties);  } **catch** (IOException e) {  e.printStackTrace();  } **catch** (Exception e) {  e.printStackTrace();  }  } |

### 获取连接

getConnection()从连接池中获取连接

|  |
| --- |
| **public** **static** Connection getConnection() {  Connection connection = **null**;  **try** {  connection = *dataSource*.getConnection();  } **catch** (SQLException e) {  e.printStackTrace();  }  **return** connection;  } |

### 关闭连接

closeConnection()

|  |
| --- |
| **public** **static** **void** closeConnection(Connection connection) {  **try** {  **if**(connection != **null**) {  connection.close();  }  } **catch** (SQLException e) {  e.printStackTrace();  }  } |

close(结果集，sql执行器，connection)

|  |
| --- |
| **public** **static** **void** close(ResultSet resultSet,Statement statement,Connection connection) {  **try** {  **if**(resultSet != **null**) {  resultSet.close();  }  } **catch** (SQLException e) {  e.printStackTrace();  }  **try** {  **if**(statement != **null**) {  statement.close();  }  } **catch** (SQLException e) {  e.printStackTrace();  }  *closeConnection*(connection);  } |

### 测试

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** SQLException {  Connection connection = JdbcUtils.*getConnection*();  String sql = "select `id`,`name`,`author`,`price`,`sales`,`stock`,`publishing` from t\_book";  PreparedStatement prepareStatement = connection.prepareStatement(sql);  ResultSet rs = prepareStatement.executeQuery();  **while** (rs.next()) {  System.***out***.println( rs.getInt("id") );  System.***out***.println( rs.getString("name") );  System.***out***.println( rs.getString("author") );  System.***out***.println( rs.getBigDecimal("price") );  System.***out***.println( rs.getInt("sales") );  System.***out***.println( rs.getInt("stock") );  System.***out***.println( rs.getString("publishing") );  System.***out***.println();  }  JdbcUtils.*close*(rs, prepareStatement, connection);  } |

## DBUtils

Commons DbUtils是Apache组织提供的一个对JDBC进行简单封装的开源工具类库，使用它能够简化JDBC应用程序的开发，同时也不会影响程序的性能。

1. 导入jar包 commons-dbutils-1.4.jar

2. 基本代码如下：

//创建 QueryRunner 对象

QueryRunner queryRunner = new QueryRunner(JDBCUtil.getDataSource());

//执行查询， 参数一为 sql 语句， 参数二为： 对应的封装对象

List<User> list = queryRunner.query("select \* from stu", new ResultSetHandler<List<User>>(){

@Override

public List<User> handle(ResultSet rs) throws SQLException {

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

while(rs.next()){

User user = new User();

user.setId(rs.getInt("id"));

user.setName(rs.getString("name"));

user.setAge(rs.getInt("age"));

list.add(user);

}

return list;

}

});

### QueryRunner类

用来执行SQL语句

update()方法执行insert、update、delete语句

query()方法执行select语句

#### 添加操作

|  |
| --- |
| **void** testInsert() **throws** Exception {  QueryRunner queryRunner = QueryRunner.**class**.newInstance();  String sql = "insert into t\_book(`name`,`author`,`price`,`sales`,`stock`,`publishing`) values(?,?,?,?,?,?)";  Connection connection = JdbcUtils.*getConnection*();  **int** result = queryRunner.update(connection, sql, "海贼王","尾田荣一郎",**new** BigDecimal(100),10,20,"日本出版社");  System.***out***.println("sql语句影响的地数：" + result);  JdbcUtils.*closeConnection*(connection);  } |

#### 删除操作

|  |
| --- |
| **void** testDelete() **throws** Exception {  QueryRunner queryRunner = QueryRunner.**class**.newInstance();  String sql = "delete from t\_book where `author` = ?";  Connection connection = JdbcUtils.*getConnection*();  **int** result = queryRunner.update(connection, sql, "尾田荣一郎");  System.***out***.println("sql语句影响的地数：" + result);  JdbcUtils.*closeConnection*(connection);  } |

#### 更新操作

|  |
| --- |
| **void** testUpdate() **throws** Exception {  QueryRunner queryRunner = **new** QueryRunner();  String sql = "update t\_book set `name`=?, `author`=?, `publishing`=? where id = ?";  Connection connection = JdbcUtils.*getConnection*();  **int** result = queryRunner.update(connection, sql,"龙珠","鸟山明","日本出版社",2);  System.***out***.println("sql语句影响的地数：" + result);  JdbcUtils.*closeConnection*(connection);  } |

### ResultSetHandler接口

对查询之后返回的结果进行处理的接口。

接口的实现类

BeanHandler将查询回来的结果封装称为JavaBean，查询只有一条记录

BeanListHandler将查询回来的多条结果封装成为JavaBean对象

ScalarHandler查询个别列的情况，获取列的值（统计函数，且只有一行一列）

**MapHandler**查询结果有多个列且是一行记录（与JavaBean对应不上）

**MapListHandler**查询结果有多个列，且有多行记录（与JavaBean对应不上）

查询的结果的列名（别名优先）要和JavaBean中的属性名一致。

#### 查询一个对象

**BeanHandler**的使用实例，可以和JavaBean对应

|  |
| --- |
| **void** testQuerySingleInstance() **throws** Exception {  QueryRunner queryRunner = **new** QueryRunner();  String sql = "select `id`, `name`, `author`, `price`, `sales`, `stock`,`publishing` from t\_book where id = ?";  Connection connection = JdbcUtils.*getConnection*();  Book book = queryRunner.query(connection, sql,**new** BeanHandler<Book>(Book.**class**),2);  System.***out***.println(book);  JdbcUtils.*closeConnection*(connection);  } |

**MapHandler**的使用实例，和JavaBean无法对应

|  |
| --- |
| **public** **void** mapHandler() **throws** SQLException {  QueryRunner queryRunner = **new** QueryRunner();  String sql = "select `name`,`author` from t\_book where id = ?";  Connection connection = JdbcUtils.*getConnection*();  Map<String, Object> query = queryRunner.query(connection, sql, **new** MapHandler(),2);  System.***out***.println(query);  JdbcUtils.*closeConnection*(connection);  } |

#### 查询多个对象

**BeanListHandler**的使用实例

|  |
| --- |
| **void** testQueryMulti() **throws** Exception {  QueryRunner queryRunner = **new** QueryRunner();  String sql = "select `id`, `name`, `author`, `publishing` from t\_book";  Connection connection = JdbcUtils.*getConnection*();  List<Book> books = queryRunner.query(connection, sql,**new** BeanListHandler<Book>(Book.**class**));  **for**(Book book : books) {  System.***out***.println(book.toShortString());  }  JdbcUtils.*closeConnection*(connection);  } |

**MapListHandler**的使用实例，和JavaBean无法对应

|  |
| --- |
| **public** **void** mapListHandler() **throws** SQLException {  QueryRunner queryRunner = **new** QueryRunner();  String sql = "select `name`,`author` from t\_book where id in (?,?)";  Connection connection = JdbcUtils.*getConnection*();  List<Map<String,Object>> query = queryRunner.query(connection, sql, **new** MapListHandler(), 2,3);  System.***out***.println(query);  JdbcUtils.*closeConnection*(connection);  } |

#### 查询统计情况

**ScalarHandler**的使用实例

|  |
| --- |
| **void** testQueryStatis() **throws** Exception {  QueryRunner queryRunner = **new** QueryRunner();  String sql = "select sum(price) from t\_book";  Connection connection = JdbcUtils.*getConnection*();  Object object = queryRunner.query(connection, sql,**new** ScalarHandler());  System.***out***.println(""+object);  JdbcUtils.*closeConnection*(connection);  } |

## Dao

Dao（data access object）数据访问对象，专门用来执行这个类对应的表的数据库操作。

在实际开发中，一张数据库表对应一个JavaBean，数据库表中的列对应JavaBean的属性，一个JavaBean对应一个Dao对象。

Dao就是一种封装的体现，将对JavaBean的数据库操作封装到一个Dao类中，和工具类的作用类似，专门用来执行数据库操作。

### Dao的使用

|  |
| --- |
| **public** **class** BookDao {  **private** QueryRunner queryRunner = **new** QueryRunner();  **public** **int** saveBook(Book book) {  Connection connection = JdbcUtils.*getConnection*();  String sql = "insert into t\_book(`name`,`author`,`price`,`sales`,`stock`,`publishing`) values(?,?,?,?,?,?)";  **try** {  **return** queryRunner.update(connection, sql, book.getName(),book.getAuthor(),book.getPrice(),book.getSales(),book.getStock(),book.getPublishing());  } **catch** (SQLException e) {  e.printStackTrace();  }**finally** {  JdbcUtils.*closeConnection*(connection);  }  **return** -1;  }  **public** **int** updateBook(Book book) {  Connection connection = JdbcUtils.*getConnection*();  String sql = "update t\_book set `name`=?,`author`=? where `name` = '红楼梦'";  **try** {  **return** queryRunner.update(connection, sql,book.getName(),book.getAuthor());  } **catch** (SQLException e) {  e.printStackTrace();  }**finally** {  JdbcUtils.*closeConnection*(connection);  }  **return** -1;  }  **public** **int** deleteBookById(Integer id) {  Connection connection = JdbcUtils.*getConnection*();  String sql = "delete from t\_book where `id` = ?";  **try** {  **return** queryRunner.update(connection, sql, id);  } **catch** (SQLException e) {  e.printStackTrace();  }**finally** {  JdbcUtils.*closeConnection*(connection);  }  **return** -1;  }  **public** Book queryBookById(Integer id) {  Connection connection = JdbcUtils.*getConnection*();  String sql = "select \* from t\_book where id = ?";  **try** {  **return** queryRunner.query(connection, sql, **new** BeanHandler<Book>(Book.**class**),id);  } **catch** (SQLException e) {  e.printStackTrace();  }**finally** {  JdbcUtils.*closeConnection*(connection);  }  **return** **null**;  }  **public** List<Book> queryBooks(){  Connection connection = JdbcUtils.*getConnection*();  String sql = "select \* from t\_book";  **try** {  **return** queryRunner.query(connection, sql, **new** BeanListHandler<>(Book.**class**));  } **catch** (SQLException e) {  e.printStackTrace();  }**finally** {  JdbcUtils.*closeConnection*(connection);  }  **return** **null**;  }  } |

### BaseDao抽取

BaseDao类就是对多个具有类似功能的Dao类进一步抽取封装，并让被抽取的类继承BaseDao类，从而提高代码的复用性和扩展性。

#### BaseDao类的实例

|  |
| --- |
| **public** **abstract** **class** BaseDao {  QueryRunner queryRunner = **new** QueryRunner();  /\*\*  \* update方法执行insert、update、delete语句  \* **@return**  \*/  **public** **int** update(String sql, Object... args) {  Connection connection = JdbcUtils.*getConnection*();  **try** {  **return** queryRunner.update(connection, sql, args);  } **catch** (SQLException e) {  e.printStackTrace();  } **finally** {  JdbcUtils.*closeConnection*(connection);  }  **return** -1;  }  /\*\*  \* 执行select查询语句，查询返回一个javaBean的sql  \* **@param** type  \* **@param** sql  \* **@param** args  \* **@return**  \*/  **public** <T> T queryForOne(Class<T> type, String sql, Object... args) {  Connection connection = JdbcUtils.*getConnection*();  **try** {  **return** queryRunner.query(connection, sql, **new** BeanHandler<T>(type),  args);  } **catch** (SQLException e) {  e.printStackTrace();  } **finally** {  JdbcUtils.*closeConnection*(connection);  }  **return** **null**;  }  **public** <T> List<T> queryForList(Class<T> type, String sql, Object... args) {  Connection connection = JdbcUtils.*getConnection*();  **try** {  **return** queryRunner.query(connection, sql, **new** BeanListHandler<T>(  type), args);  } **catch** (SQLException e) {  e.printStackTrace();  } **finally** {  JdbcUtils.*closeConnection*(connection);  }  **return** **null**;  }  } |

#### 继承BaseDao类后的BookDao类

|  |
| --- |
| **public** **class** BookDao2 **extends** BaseDao {  **public** **int** saveBook(Book book) {  String sql = "insert into t\_book(`name`,`author`,`price`,`sales`,`stock`,`publishing`) values(?,?,?,?,?,?)";  **return** update(sql, book.getName(), book.getAuthor(), book.getPrice(), book.getSales(), book.getStock(),  book.getPublishing());  }  **public** **int** updateBook(Book book) {  String sql = "update t\_book set `name`=?,`author`=?,`price`=?,`sales`=?,`stock`=?,`publishing`=? where id = ?";  **return** update(sql, book.getName(), book.getAuthor(), book.getPrice(), book.getSales(), book.getStock(),  book.getPublishing(), book.getId());  }  **public** **int** deleteBookById(Integer id) { // 根据主键删除图书  String sql = "delete from t\_book where id = ?";  **return** update(sql, id);  }  **public** Book queryBookById(Integer id) {  String sql = "select `id`,`name`,`author`,`price`,`sales`,`stock`,`publishing` from t\_book where id = ?";  **return** queryForOne(Book.**class**, sql, id);  }  **public** List<Book> queryBooks() {// 查询全部图书  String sql = "select `id`,`name`,`author`,`price`,`sales`,`stock`,`publishing` from t\_book";  **return** queryForList(Book.**class**, sql);  }  } |

# Dom4j

dom4j用于解析xml文件，该即使不是sun公司的技术，而是第三方公司的技术，所以使用dom4j 就要到其官网下载dom4j的jar包，然后导入到项目中。

## XML解析方式

DOM:消耗内存：先把xml文档都读到内存中，然后再用DOM API来访问树形结构，并获取数据。这个写起来很简单，但是很消耗内存。

SAX:解析效率高，占用内存少，基于事件驱动的。更加简单地说就是对文档进行顺序扫描，当扫描到文档(document)开始与结束、元素(element)开始与结束等地方时通知事件处理函数，由事件处理函数做相应动作，然后继续同样的扫描，直至文档结束。

PULL:与SAX类似，也是基于事件驱动，可以调用其next方法，来获取下一个解析事件（开始文档，结束文档，开始标签，结束标签），当处于某个元素时可以调用XmlPullParser的getAttributte方法来获取属性的值，也可调用它的nextText方法获取本节点的值。

## dom4j编程步骤

### 简单入门

第一步：先加载xml文件创建Document对象

第二步：通过解析Document对象拿到根元素对象

第三步：通过根元素.elelemts(标签名); 可以返回一个集合，这个集合里放着。所有你指定的标签名的元素对象

第四步：找到你想要修改、删除的子元素，进行相应在的操作

第五步：保存到硬盘上

|  |
| --- |
| //dom4j 获取Documet对象 public void getDocument() throws DocumentException {  // 要创建一个Document对象，需要我们先创建一个SAXReader对象  SAXReader reader = new SAXReader();  // 这个对象用于读取xml文件，然后返回一个Document。  Document document = reader.read("src/books.xml");  // 打印到控制台，看看是否创建成功  System.*out*.println(document); } |

### 遍历标签

需要分四步操作:

第一步，通过创建SAXReader对象。来读取xml文件，获取Document对象

第二步，通过Document对象。拿到XML的根元素对象

第三步，通过根元素对象。获取所有的book 标签对象

第四步，遍历每个book标签对象。然后获取到book标签对象内的每一个元素，再通过getText() 方法拿到起始标签和结束标签之间的文本内容

|  |
| --- |
| //读取xml文件中的内容 public void readXML() throws DocumentException {  //第一步，通过创建SAXReader对象。来读取xml文件，获取Document对象  SAXReader reader = new SAXReader();  Document document = reader.read("src/books.xml");  //第二步，通过Document对象。拿到XML的根元素对象  Element root = document.getRootElement();  // 打印测试  // Element.asXML() 它将当前元素转换成为String对象  // System.out.println( root.asXML() );  //第三步，通过根元素对象。获取所有的book 标签对象  // Element.elements(标签名)它可以拿到当前元素下的指定的子元素的集合  List<Element> books = root.elements("book");  //第四步，遍历每个book标签对象。然后获取到book标签对象内的每一个元素，  for (Element book : books) {  // 测试  // System.out.println(book.asXML());  // 拿到book下面的name元素对象  Element nameElement = book.element("name");  // 拿到book下面的price元素对象  Element priceElement = book.element("price");  // 拿到book下面的author元素对象  Element authorElement = book.element("author");  // 再通过getText() 方法拿到起始标签和结束标签之间的文本内容  System.*out*.println("书名" + nameElement.getText() + " , 价格:"  + priceElement.getText() + ", 作者：" + authorElement.getText());  } } |

# NIO

新IO和传统的IO有相同的目的，都是用于进行输入输出的，但新IO使用了不同的方式来处理输入输出，采用内存映射文件的方式，将文件或者文件的一段区域映射到内存中，就可以像访问内存一样的来访问文件了，这种方式效率比旧IO要高很多，但是目前好多地方我们看到的还是旧IO的引用，所以我们仍以旧IO为主，知道NIO即可。

BIO的特点：单向，无缓冲，适用于文件读写。

NIO的特点：双向，有缓冲，基于管道的，适用于网络传输。

## Buffer

缓冲

## Channer

通道

## Path

与平台无关的路径

## Paths

包含了返回Path的静态方法

|  |  |
| --- | --- |
| **Paths方法摘要** | |
| static Path | **get**(overload)根据传入的参数将其转换为Path对象 |

## Files

操作文件的工具类，提供了大量的方法

|  |  |
| --- | --- |
| **Files方法摘要** | |
| static long/Path | **copy**(overload)将文件中的字节输出到指定位置 |

# JUC

在 Java 5.0 提供了 java.util.concurrent （简称 JUC ）包，在此包中增加了在并发编程中很常用的实用工具类，用于定义类似于线程的自定义子系统，包括线程池、异步 IO 和轻量级任务框架。提供可调的、灵活的线程池。还提供了设计用于多线程上下文中的 Collection 实现等。

## volatile关键字

### 内存可见性

内存可见性（**Memory Visibility**）是指当某个线程正在使用对象状态而另一个线程在同时修改该状态，需要确保当一个线程修改了对象状态后，其他线程能够看到发生的状态变化。

可见性错误是指当读操作与写操作在不同的线程中执行时，无法确保执行读操作的线程能适时地看到其他线程写入的值，有时甚至是根本不可能的事情。

主存

i=0

线程A

缓存

i++

线程B

缓存

i

如上图所示，线程A先从主存中获取变量i并执行i++操作，而在线程A将i++的结果返回到主存中前，线程B从主存中获取到变量i，此时线程B获取到的变量i的值仍然为0。

### volatile

可以通过synchronized同步来保证对象被安全地发布，除此之外也可以使用volatile关键字定义变量，用来确保变量的更新操作通知到其他线程。可以将 volatile 看做一个轻量级的锁，但是又与锁有些不同：

1、volatile不具备互斥性

2、volatile不能保证变量的原子性

|  |
| --- |
| public class TestVolatile {  public static void main(String[] args) {  VolatileDemo volatileDemo = new VolatileDemo();  new Thread(volatileDemo).start();  /\* 如果flag变量没有使用volatile关键字定义，main主线程的while循环从主存中读取到的  flag值就为false，从而一直处在死循环中 \*/  while (true){  // 因为Java中循环操作速度很快，快到无法从主存中第二次获取到变量flag的值 // 因此flag的值一直为false  if(volatileDemo.isFlag()){ System.*err*.println("----------------");break; }  }  //如果在循环中添加打印语句，循环会变慢，此时就可以再一次从主存中获取flag的值  //即修改后的flag值，此时值为true，则下一次的循环时会进入if判断然后跳出循环  //使用debug模式开启时也会时循环执行速度变慢，从而获取到修改会的flag的值  System.*out*.println(Thread.*currentThread*().getName());  } } class VolatileDemo implements Runnable{  /\* volatile关键字定义的变量在被线程修改时不再进入该线程的缓存中进行处理，而是直接在主存中  进行修改，从而避免了内存可见性的问题 \*/  private volatile boolean flag = false;  @Override  public void run() {  try { Thread.*sleep*(200); }   catch (InterruptedException e) { e.printStackTrace(); }  flag = true;  System.*err*.println("flag = " + flag);  }  public boolean isFlag() { return flag; }  public void setFlag(boolean flag) { this.flag = flag; } } |

## 原子变量封装类

### 原子变量

因为volatile定义的变量不具备原子性，如i++的操作在底层就分为了“读-改-写”三个步骤，因此在juc的保重提供了原子变量来保证原子性，通过调用原子变量的方法来保证原子性。

类的小工具包，支持在单个变量上解除锁的线程安全编程。事实上，此包中的类可将 volatile 值、字段和数组元素的概念扩展到那些也提供原子条件更新操作的类。

类 AtomicBoolean、AtomicInteger、AtomicLong 和 AtomicReference 的实例各自提供对相应类型单个变量的访问和更新。每个类也为该类型提供适当的实用工具方法。

AtomicIntegerArray、AtomicLongArray 和 AtomicReferenceArray 类进一步扩展了原子操作，对这些类型的数组提供了支持。这些类在为其数组元素提供 volatile 访问语义方面也引人注目，这对于普通数组来说是不受支持的。

核心方法：

boolean compareAndSet(expectedValue, updateValue)当前值如果等于expectedValue预期值，则将updateValue更新值赋值给当前值。

java.util.concurrent.atomic 包下提供了一些原子操作的常用类:

AtomicBoolean 、AtomicInteger 、AtomicLong 、 AtomicReference

AtomicIntegerArray 、AtomicLongArray

AtomicMarkableReference

AtomicReferenceArray

AtomicStampedReference

|  |
| --- |
| public class TestAtomic {  public static void main(String[] args) {  AtomicDemo atomicDemo = new AtomicDemo();  for (int i = 0; i < 10; i++) {  new Thread(atomicDemo).start();  }  } } class AtomicDemo implements Runnable{  //volatile只能避免内存不可见问题，而无法保证变量的原子性  private volatile int serialNumber = 0;  //原子变量可以保证变量的原子性  private AtomicInteger atomicNumber = new AtomicInteger();  @Override  public void run() {  try {  Thread.*sleep*(200);  } catch (InterruptedException e) {  e.printStackTrace();  }  //volatile修饰的变量在多线程中的运算可能会产生异常  System.*out*.print(serialNumber++);  //原子变量通过方法的形式进行运算来避免发生异常  System.*err*.println(atomicNumber.getAndIncrement());  } } |

### CAS算法

CAS (**Compare-And-Swap**) 是一种硬件对并发的支持，针对多处理器操作而设计的处理器中的一种特殊指令，用于管理对共享数据的并发访问。CAS 是一种无锁的非阻塞算法的实现，包含了 3 个操作数：需要读写的内存值V、进行比较的值A和拟写入的新值B，当且仅当 V 的值等于 A 时，CAS 通过原子方式用新值 B 来更新 V 的值，否则不会执行任何操作。

原子变量保证原子性的原理就在于使用了CAS算法，如在AtomicInteger的getAndIncrement()方法中就通过compareAndSwapInt本地方法使用了CAS算法，如下所示：

|  |
| --- |
| public final int getAndIncrement() {  return *unsafe*.getAndAddInt(this, *valueOffset*, 1); }  public final int getAndAddInt(Object var1, long var2, int var4) {  int var5;  do {  var5 = this.getIntVolatile(var1, var2);  } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));   return var5; }  //compareAndSwapInt本地方法的C++代码 compareAndSwap (volatile jint \*addr, jint old, jint new\_val) {  jboolean result = false;  spinlock lock;  if ((result = (\*addr == old)))  \*addr = new\_val;  return result; } |

#### 模拟CAS算法

|  |
| --- |
| public class TestCompareAndSwap {  public static void main(String[] args) {  CompareAndSwap cas = new CompareAndSwap();  for (int i = 0; i < 100; i++) {  //多个线程并发执行时，只有获取到预期值的线程才能成功将内存值修改为目标值  //而获取的预期值和当前内存值不同的线程则修改失败  new Thread(() -> {  int expectedValue = cas.get();  boolean b = cas.compareAndSet(expectedValue,(int)(Math.*random*()\*101));  System.*out*.println(b);  },"线程" + i ).start();  }  } } //通过Java代码模拟CAS算法的原理 class CompareAndSwap {  private int value;  public synchronized int get() {  return value;  }  /\* value是需要读写的内存值，expectedValue是旧的预期值，newValue是将要更新的目标值  只有当需要读写的内存值和旧的预期值相同时才会将内存值修改为目标值 \*/  public synchronized int compareAndSwap(int expectedValue, int newValue) {  int oldValue = value;  if (oldValue == expectedValue) {  value = newValue;  }  return oldValue;  }  public synchronized boolean compareAndSet(int expectedValue, int newValue) {  System.*out*.println(Thread.*currentThread*().getName()  + " : " + get() + " : " + expectedValue + " : " + newValue);  return expectedValue == compareAndSwap(expectedValue, newValue);  } } |

#### CAS算法缺陷

CAS虽然解决了原子操作的问题，但是其仍然有三个缺点：

1、循环时间长开销很大

AtomicInteger的getAndAddInt方法执行时，如果CAS失败，会一直进行尝试。如果CAS长时间一直不成功，可能会给CPU带来很大的开销。

2、只能保证一个共享变量的原子操作

当对一个共享变量执行操作时，可以使用循环CAS的方式来保证原子操作，但是对多个共享变量操作时，循环CAS就无法保证操作的原子性，此时就可以用锁来保证原子性。

3、ABA问题

如果内存地址V初次读取的值是A，并且在准备赋值的时候检查到它的值仍然为A，无法保证它的值没有被其他线程改变过。

如果在这段期间它的值曾经被改成了B，后来又被改回为A，那CAS操作就会误认为它从来没有被改变过。这个漏洞称为CAS操作的“ABA”问题。Java并发包为了解决这个问题，提供了一个带有标记的原子引用类“AtomicStampedReference”，它可以通过控制变量值的版本来保证CAS的正确性。因此，在使用CAS前要考虑清楚“ABA”问题是否会影响程序并发的正确性，如果需要解决ABA问题，改用传统的互斥同步可能会比原子类更高效。

## 并发集合容器

Java 5.0 在 java.util.concurrent 包中提供了多种并发容器类来改进同步容器的性能。

ConcurrentHashMap 同步容器类是Java 5 增加的一个线程安全的哈希表。对与多线程的操作，介于 HashMap 与 Hashtable 之间。内部采用“锁分段”机制替代 Hashtable 的独占锁。进而提高性能。

此包还提供了设计用于多线程上下文中的 Collection 实现：ConcurrentHashMap、ConcurrentSkipListMap、ConcurrentSkipListSet、CopyOnWriteArrayList 和 CopyOnWriteArraySet。当期望许多线程访问一个给定 collection 时，ConcurrentHashMap 通常优于同步的 HashMap，ConcurrentSkipListMap 通常优于同步的 TreeMap。当期望的读数和遍历远远大于列表的更新数时，CopyOnWriteArrayList 优于同步的 ArrayList。

|  |
| --- |
| //CopyOnWriteArrayList/CopyOnWriteArraySet写入并复制集合 //若写入操作较多时，不推荐使用，因为每次都会进行复制，效率较低 //若并发迭代较多时，推荐使用 public class TestCopyOnWriteArrayList {  public static void main(String[] args) {  ThreadDemo td = new ThreadDemo();  for (int i = 0; i < 10; i++) {  new Thread(td).start();  System.*out*.println(i);  }  } } class ThreadDemo implements Runnable{  //这种方式给集合添加锁会使程序效率较低  private static List<String> *list* = Collections.*synchronizedList*(new ArrayList<>());  private static CopyOnWriteArrayList<String> *copyOnWriteArrayList* =  new CopyOnWriteArrayList<>();  static{  *copyOnWriteArrayList*.add("AA");  *copyOnWriteArrayList*.add("BB");  *copyOnWriteArrayList*.add("CC");  }  @Override  public void run() {  Iterator<String> iterator = *copyOnWriteArrayList*.iterator();  while(iterator.hasNext()){  System.*out*.println(iterator.next());  *copyOnWriteArrayList*.add("AA");  }  } } |

## CountDownLatch闭锁

CountDownLatch 一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

闭锁可以延迟线程的进度直到其到达终止状态，闭锁可以用来确保某些活动直到其他活动都完成才继续执行：确保某个计算在其需要的所有资源都被初始化之后才继续执行;确保某个服务在其依赖的所有其他服务都已经启动之后才启动;等待直到某个操作所有参与者都准备就绪再继续执行。

|  |
| --- |
| // CountDownLatch闭锁，在其他多个线程进行运算时，当前线程处于等待状态， // 若其他线程运行结束后，当前线程再进行运算 public class TestCountDownLatch {  public static void main(String[] args) {  //CountDownLatch底层维护了一个计数器，并通过构造器参数的形式给该计数器传值  //该值表示需要等待执行完毕的线程的个数，每执行完毕一个线程，该值减一  //该值为0时表示等待的线程全部执行完毕，然后解锁  CountDownLatch countDownLatch = new CountDownLatch(10);  LatchDemo latchDemo = new LatchDemo(countDownLatch);  long start = System.*currentTimeMillis*();  for (int i = 0; i < 10; i++) {  //并发执行的线程数小于计数器的值会导致闭锁提前解锁  //并发执行的线程数大于计数器的值会导致闭锁一直等待而无法解锁  new Thread(latchDemo).start();  }  try {  //主线程等待，直到计数器的值为0  countDownLatch.await();  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println("耗费时间为：" + (System.*currentTimeMillis*() - start));  } } class LatchDemo implements Runnable{  private CountDownLatch latch;  public LatchDemo(CountDownLatch latch){  this.latch = latch;  }  @Override  public void run(){  try {  for (int i = 0; i < 50000; i++) {  if(i % 2 == 0){  System.*out*.print(i);  }  }  System.*out*.println();  } finally {  System.*out*.println(latch.getCount());  //将计数器的值减一，直到其值为0  latch.countDown();  }  } } |

## 实现Callable接口

Java 5.0 在 java.util.concurrent 提供了一个新的创建执行线程的方式：Callable 接口

Callable 接口类似于 Runnable，两者都是为那些其实例可能被另一个线程执行的类设计的。但是 Runnable 不会返回结果，并且无法抛出经过检查的异常，而实现Callable接口可以重写具有返回值的call()方法。

Callable 需要依赖FutureTask ，FutureTask 也可以用作闭锁。

|  |
| --- |
| // Callable相较于Runnable接口，多了一个泛型和返回值，其他方面没有区别 public class TestCallable {  public static void main(String[] args) {  ThreadTwo thread = new ThreadTwo();  //Callable接口需要FutureTask的支持，用于接收运算结果  FutureTask<String> task = new FutureTask<>(thread);  new Thread(task).start();  try {  //因为FutureTask获取结果必须等待其他线程执行完毕，因此其内部封装了闭锁  String hello = task.get();  System.*out*.println(hello);  } catch (InterruptedException | ExecutionException e) {  e.printStackTrace();  }  } } class ThreadTwo implements Callable<String>{  @Override  public String call() throws Exception {  return "hello";  } } class ThreadOne implements Runnable{  @Override  public void run() { } } |

## Lock同步锁

在 Java 5.0 之前，协调共享对象的访问时可以使用的机制只有 synchronized 和 volatile 。Java 5.0 后增加了一些新的机制，但并不是一种替代内置锁的方法，而是当内置锁不适用时，作为一种可选择的高级功能。

ReentrantLock 实现了 Lock 接口，并提供了与 synchronized 相同的互斥性和内存可见性，但相较于 synchronized 提供了更高的处理锁的灵活性。

|  |
| --- |
| /\* 解决多线程安全问题的方式  隐式锁  1、同步代码块  2、同步方法  显式锁  3、同步锁Lock，通过lock()方法上锁，然后必须通过unlock()方法释放锁 \*/ public class TestLock {  public static void main(String[] args) {  Ticket ticket = new Ticket();  new Thread(ticket, "1号窗口").start();  new Thread(ticket, "2号窗口").start();  new Thread(ticket, "3号窗口").start();  } } class Ticket implements Runnable {  int ticket = 100;  private Lock lock = new ReentrantLock();  @Override  public void run() {  while (true) {  lock.lock();  try {  if (ticket > 0) {  try {  Thread.*sleep*(20);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(Thread.*currentThread*().getName()  + "完成售票，余票为：" + --ticket);  }  } finally {  lock.unlock();  }  }  } } |

## Condition控制线程通信

Condition 接口描述了可能会与锁有关联的条件变量。这些变量在用法上与使用 Object.wait 访问的隐式监视器类似，但提供了更强大的功能。需要特别指出的是，单个 Lock 可能与多个 Condition 对象关联。为了避免兼容性问题，Condition 方法的名称与对应的 Object 版本中的不同。

在 Condition 对象中，与 wait、notify 和 notifyAll 方法对应的分别是 await、signal 和 signalAll。

Condition 实例实质上被绑定到一个锁上。要为特定 Lock 实例获得 Condition 实例，请使用其 newCondition() 方法。

### Object类的线程通信方法

|  |
| --- |
| public class TestProducerAndConsumer {  public static void main(String[] args) {  Clerk clerk = new Clerk();  Producer pro1 = new Producer(clerk);  Consumer con1 = new Consumer(clerk);  new Thread(pro1, "生产者1").start();  new Thread(con1, "消费者1").start();  new Thread(pro1, "生产者2").start();  new Thread(con1, "消费者2").start();  } }  class Clerk {  private int product = 0;  public synchronized void sale() {  // 在对变量进行判断时要使用while进行判断，而不能使用if，  // 使用if可能会发生虚假唤醒的问题  if (product <= 0) {  System.*out*.println("缺货");  try {  this.wait();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  //唤醒其他线程以及对变量的操作不要在else代码块中执行  System.*out*.println(Thread.*currentThread*().getName() + " 消费了 " + --product);  this.notifyAll();  }  public synchronized void get() {//循环次数：0  if (product >= 1) {  System.*out*.println("产品已满！");  try {  this.wait();  } catch (InterruptedException e) {  }  }  System.*out*.println(Thread.*currentThread*().getName() + " 生产了 " + ++product);  this.notifyAll();  } }  class Producer implements Runnable {  private Clerk clerk;  Producer(Clerk clerk) {  this.clerk = clerk;  }  @Override  public void run() {  for (int i = 0; i < 20; i++) {  try {  Thread.*sleep*(200);  } catch (InterruptedException e) {  }  clerk.get();  }  } } class Consumer implements Runnable {  private Clerk clerk;  Consumer(Clerk clerk) {  this.clerk = clerk;  }  @Override  public void run() {  for (int i = 0; i < 20; i++) {  clerk.sale();  }  } } |

### Condition的线程通信方法

|  |
| --- |
| public class TestLockProducerAndConsumer {  public static void main(String[] args) {  LockClerk lockClerk = new LockClerk();  LockConsumer lockConsumer = new LockConsumer(lockClerk);  LockProducer lockProducer = new LockProducer(lockClerk);  new Thread(lockConsumer,"消费者").start();  new Thread(lockProducer,"生产者").start();  } } class LockClerk{  private int product = 0;  private Lock lock = new ReentrantLock();  private Condition condition = lock.newCondition();  public void sale(){  lock.lock();  try {  while(product <= 0){  System.*out*.println("缺货");  try {  condition.await();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  System.*out*.println(Thread.*currentThread*().getName() + "消费了" + --product);  condition.signalAll();  }finally {  lock.unlock();  }  }  public void get(){  lock.lock();  try {  while(product >= 1){  System.*out*.println("产品已满");  try {  condition.await();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  System.*out*.println(Thread.*currentThread*().getName() + "生产了" + ++product);  condition.signalAll();  }finally {  lock.unlock();  }  } } class LockProducer implements Runnable{  private LockClerk lockClerk;  public LockProducer(LockClerk lockClerk){  this.lockClerk = lockClerk;  }  @Override  public void run() {  for (int i = 0; i < 20; i++) {  try {  Thread.*sleep*(200);  } catch (InterruptedException e) {  e.printStackTrace();  }  lockClerk.get();  }  } } class LockConsumer implements Runnable{  private LockClerk lockClerk;  public LockConsumer(LockClerk lockClerk) {  this.lockClerk = lockClerk;  }  @Override  public void run() {  for (int i = 0; i < 20; i++) {  lockClerk.sale();  }  } } |

### 虚假唤醒

在多线程环境下，执行唤醒所有线程的方法时可能会唤醒多个阻塞在条件变量上的线程，导致数据发生异常，这种效应就称为虚假唤醒。

#### 虚假唤醒现象

1） 1号线程从队列中获取了一个元素，此时队列变为空。

2） 2号线程也想从队列中获取一个元素，但此时队列为空，2号线程便只能进入阻塞(cond.wait())，等待队列非空。

3） 这时，3号线程将一个元素入队，并调用cond.notify()唤醒条件变量。

4） 处于等待状态的2号线程接收到3号线程的唤醒信号，便准备解除阻塞状态，执行接下来的任务(获取队列中的元素)。

5） 然而可能出现这样的情况：当2号线程准备获得队列的锁，去获取队列中的元素时，此时1号线程刚好执行完之前的元素操作，返回再去请求队列中的元素，1号线程便获得队列的锁，检查到队列非空，就获取到了3号线程刚刚入队的元素，然后释放队列锁。

6） 等到2号线程获得队列锁，判断发现队列仍为空，1号线程“偷走了”这个元素，所以对于2号线程而言，这次唤醒就是“虚假”的，它需要再次等待队列非空。

#### 解决方式

在多个线程中对变量的判断和wait()方法放在while循环中进行判断，而不是if语句判断。

如果使用if判断，多个等待线程设置为不满足if判断则等待，唤醒后不再进行条件判断而直接执行后续操作。此时如果这多个等待线程被唤醒，但实际上判断条件并不满足，仍然会执行必须要满足判断条件后才能执行的操作，就会发生数据异常。

使用while判断就避免了这种多个线程被唤醒却不是因为条件满足而被唤醒的情况，因为每一次唤醒都要不断检查判断条件是否满足，不满足则继续等待，直到满足判断条件才执行下一步操作。

### 线程按序交替

通过Condition控制线程通信的方法来实现线程按指定顺序交替执行。

编写一个程序，开启 3 个线程，这三个线程的 ID 分别为 A、B、C，每个线程将自己的 ID 在屏幕上打印 10 遍，要求输出的结果必须按顺序显示。

如：ABCABCABC…… 依次递归

|  |
| --- |
| public class TestAlternate {  public static void main(String[] args) {  AlternateDemo alternateDemo = new AlternateDemo();  new Thread(()->{ for (int i = 0; i <= 10; i++) { alternateDemo.loopA(i); }   },"A").start();  new Thread(()->{ for (int i = 0; i <= 10; i++) { alternateDemo.loopB(i); }  },"B").start();  new Thread(()->{ for (int i = 0; i <= 10; i++) { alternateDemo.loopC(i); }  },"C").start();  } } class AlternateDemo{  private int number = 1;  private Lock lock = new ReentrantLock();  private Condition condition1 = lock.newCondition();  private Condition condition2 = lock.newCondition();  private Condition condition3 = lock.newCondition();  public void loopA(int totalLoop){  lock.lock();  try {  if(number != 1){  try { condition1.await(); }   catch (InterruptedException e) { e.printStackTrace(); }  }  System.*out*.println(Thread.*currentThread*().getName() + " : " + totalLoop);  number = 2;  condition2.signal();  }finally { lock.unlock(); }  }  public void loopB(int totalLoop){  lock.lock();  try {  if(number != 2){  try { condition2.await(); }   catch (InterruptedException e) { e.printStackTrace(); }  }  System.*out*.println(Thread.*currentThread*().getName() + " : " + totalLoop);  number = 3;  condition3.signal();  }finally { lock.unlock(); }  }  public void loopC(int totalLoop){  lock.lock();  try {  if(number != 3){  try { condition3.await(); }   catch (InterruptedException e) { e.printStackTrace(); }  }  System.*out*.println(Thread.*currentThread*().getName() + " : " + totalLoop);  number = 1;  condition1.signal();  }finally { lock.unlock(); }  } } |

## ReadWriteLock锁

ReadWriteLock 维护了一对相关的锁，一个用于只读操作，另一个用于写入操作。只要没有 writer，读取锁可以由多个 reader 线程同时保持。写入锁是独占的。

ReadWriteLock 读取操作通常不会改变共享资源，但执行写入操作时，必须独占方式来获取锁。对于读取操作占多数的数据结构。 ReadWriteLock 能提供比独占锁更高的并发性。而对于只读的数据结构，其中包含的不变性可以完全不需要考虑加锁操作。

|  |
| --- |
| public class TestReadWriteLock {  public static void main(String[] args) {  ReadWriteLockDemo readWriteLockDemo = new ReadWriteLockDemo();  for (int i = 0; i < 10; i++) {  new Thread(()->readWriteLockDemo.set(((int) (Math.*random*() \* 101)))).start();  }  for (int i = 0; i < 100; i++) {  new Thread(()->readWriteLockDemo.get()).start();  }  } } class ReadWriteLockDemo{  private int number;  private ReadWriteLock lock = new ReentrantReadWriteLock();  public void get(){  lock.readLock().lock();  System.*out*.println(Thread.*currentThread*().getName()+":"+number);  lock.readLock().unlock();  }  public void set(int number){  lock.writeLock().lock();  System.*out*.println(Thread.*currentThread*().getName());  this.number = number;  lock.writeLock().unlock();  } } |

## 线程八锁

一个对象里面如果有多个synchronized方法，某一个时刻内，只要一个线程去调用其中的一个synchronized方法了，其它的线程都只能等待，换句话说，某一个时刻内，只能有唯一一个线程去访问这些synchronized方法

锁的是当前对象this，被锁定后，其它的线程都不能进入到当前对象的其它的synchronized方法

加个普通方法后发现和同步锁无关

换成两个对象后，不是同一把锁了，情况立刻变化。

都换成静态同步方法后，情况又变化

所有的非静态同步方法用的都是同一把锁——实例对象本身，也就是说如果一个实例对象的非静态同步方法获取锁后，该实例对象的其他非静态同步方法必须等待获取锁的方法释放锁后才能获取锁，可是别的实例对象的非静态同步方法因为跟该实例对象的非静态同步方法用的是不同的锁，所以毋须等待该实例对象已获取锁的非静态同步方法释放锁就可以获取他们自己的锁。

所有的静态同步方法用的也是同一把锁——类对象本身，这两把锁是两个不同的对象，所以静态同步方法与非静态同步方法之间是不会有竞态条件的。但是一旦一个静态同步方法获取锁后，其他的静态同步方法都必须等待该方法释放锁后才能获取锁，而不管是同一个实例对象的静态同步方法之间，还是不同的实例对象的静态同步方法之间，只要它们同一个类的实例对象！

|  |
| --- |
| /\* 线程锁的八种情况：  \* 1. 两个同步方法，一个对象，两个线程，分别调用 getOne()和 getTwo(), 打印？ //1 2  \* 2. 在 getOne() 方法中添加睡眠3秒，打印？ //1 2  \* 3. 添加非同步方法，启动三个线程，打印？ // 3 1 2  \* 4. 创建两个对象，一个调用 getOne() 一个调用getTwo(), 打印？ //2 1  \* 5. 将 getOne() 修改为静态同步方法，一个对象，打印？ //2 1  \* 6. 将 getTwo() 修改为静态同步方法，一个对象，打印？ //1 2  \* 7. 将 getTwo() 修改为非静态同步方法，两个对象， 打印？ //2 1  \* 8. 两个静态同步方法，两个对象，打印？// 1 2  \* 结论：  \* ①非静态同步方法的锁为 this，静态同步方法的锁为当前类对象的 Class 实例  \* ②当多个线程共享一把锁，一个特定时间点只能有一个线程执行流执行，无论几个方法。\*/ public class TestThread8Moniter {  public static void main(String[] args) {  Thread8Demo td = new Thread8Demo();  Thread8Demo td2 = new Thread8Demo();  new Thread(() -> td.*getOne*()).start();  new Thread(() -> td2.getTwo()).start();  new Thread(() -> td2.getThree()).start();  } } class Thread8Demo{  public static synchronized void getOne(){//ThreadDemo.class  try { Thread.*sleep*(3000); } catch (InterruptedException e) { }  System.*out*.println("1");  }  public synchronized void getTwo(){  System.*out*.println("2");  }  public void getThree(){  System.*out*.println("3");  } } |

## 线程池

第四种获取线程的方法：线程池，一个 ExecutorService，它使用可能的几个池线程之一执行每个提交的任务，通常使用 Executors 工厂方法配置。

线程池可以解决两个不同问题：由于减少了每个任务调用的开销，它们通常可以在执行大量异步任务时提供增强的性能，并且还可以提供绑定和管理资源（包括执行任务集时使用的线程）的方法。每个 ThreadPoolExecutor 还维护着一些基本的统计数据，如完成的任务数。

为了便于跨大量上下文使用，此类提供了很多可调整的参数和扩展钩子 (hook)。但是，强烈建议程序员使用较为方便的 Executors 工厂方法 ：

Executors.newCachedThreadPool()（无界线程池，可以进行自动线程回收）

Executors.newFixedThreadPool(int)（固定大小线程池）

Executors.newSingleThreadExecutor()（单个后台线程）

它们均为大多数使用场景预定义了设置。

|  |
| --- |
| /\* 一、线程池：维护了一个线程队列，队列中有多个处于等待状态的线程，  避免了每次新建与销毁线程的额外开销，提高了相应速度。  \* 二、线程池的体系结构  \* java.util.concurrent.Executor : 线程池与线程调度的根接口  \* |-- \*\* java.util.concurrent.ExecutorService : 线程池的根接口  \* |-- java.util.concurrent.ThreadPoolExecutor : 线程池的实现类  \* |-- java.util.concurrent.ScheduledExecutorService : 线程调度的子接口  \* |--java.util.concurrent.ScheduledThreadPoolExecutor :   \* 继承了 ThreadPoolExecutor, 实现了 ScheduledExecutorService  \* 三、java.util.concurrent.Executors 工具类  \* ExecutorService newCachedThreadPool() : 缓存线程池，可根据实际应用，自动的分配线程个数  \* ExecutorService newFixedThreadPool() : 固定大小的线程池  \* ExecutorService newSingleThreadExecutor() : 只有一个线程的线程池  \*  \* ScheduledExecutorService newScheduledThreadPool(int corePoolSize) : 线程调度的线程池，  可以延迟或定时执行任务  \*/ public class TestThreadPool {  public static void main(String[] args) throws Exception {  ExecutorService pool = Executors.*newFixedThreadPool*(5);  List<Future<Integer>> list = new ArrayList<>();  for (int i = 0; i < 10; i++) {  Future<Integer> future = pool.submit(new Callable<Integer>() {  @Override  public Integer call() throws Exception {  int num = (int)(Math.*random*() \* 101);  System.*out*.println(Thread.*currentThread*().getName() + ":" + num);  return num;  }  });  list.add(future);  }  pool.shutdown();  for (Future<Integer> future : list) {  System.*out*.println(future.get());  }  /\*ThreadPoolDemo tp = new ThreadPoolDemo();  //创建数量为 5 的线程池  ExecutorService pool = Executors.newFixedThreadPool(10);  //分配任务  for (int i = 0; i < 10; i++) {  pool.submit(tp);  }  //释放资源  pool.shutdown();\*/  }  /\* new Thread(tp).start();  \* new Thread(tp).start(); \*/ }  class ThreadPoolDemo implements Runnable{  private int i = 0;  @Override  public void run() {  while(i <= 100){  System.*out*.println(Thread.*currentThread*().getName() + ":" + i++);  }  } } |

## 线程调度

一个 ExecutorService，可安排在给定的延迟后运行或定期执行的命令。

|  |
| --- |
| public class TestScheduledThreadPool {  public static void main(String[] args) throws Exception {  //线程数量为5的线程池  ScheduledExecutorService pool = Executors.*newScheduledThreadPool*(5);  //分配任务  for (int i = 0; i < 5; i++) {  Future<Integer> future = pool.schedule(new Callable<Integer>() {  @Override  public Integer call() throws Exception {  int num = (int)(Math.*random*() \* 101);  System.*out*.println(Thread.*currentThread*().getName() + ": " + num);  return num;  }  }, 3, TimeUnit.*SECONDS*);  System.*out*.println(future.get());  }  //释放资源  pool.shutdown();  } } |

## Fork/Join

Fork/Join框架是指将一个大任务拆分(fork)成若干个小任务（拆到指定临界值不可再拆时），再将一个个的小任务运算的结果进行Join汇总。

### 工作窃取模式

工作窃取模式（**work-stealing**）是指当执行新的任务时它可以将其拆分分成更小的任务执行，并将小任务加到线程队列中，然后再从一个随机线程的队列中偷一个并把它放在自己的队列中。

相对于一般的线程池实现，fork/join框架的优势体现在对其中包含的任务的处理方式上.在一般的线程池中，如果一个线程正在执行的任务由于某些原因无法继续运行，那么该线程会处于等待状态。而在fork/join框架实现中，如果某个子问题由于等待另外一个子问题的完成而无法继续运行。那么处理该子问题的线程会主动寻找其他尚未运行的子问题来执行.这种方式减少了线程的等待时间，提高了性能。

|  |
| --- |
| public class TestForkJoinPool {  //Fork/Join框架能将CPU利用率提高到100%，在电脑CPU核数较多时效果更明显  public static void main(String[] args) {  long start = System.*currentTimeMillis*();  ForkJoinPool pool = new ForkJoinPool();  ForkJoinTask<Long> task = new ForkJoinSum(0L, 300000000000L);  long sum = pool.invoke(task);  System.*out*.println(sum);  long end = System.*currentTimeMillis*();  System.*out*.println("耗费时间为：" + (end - start)); //7650 62742  }  //使用普通for循环，在电脑配置较低，任务量较小时和Fork/Join框架的效率差不多  @Test  public void test1(){  long start = System.*currentTimeMillis*();  long sum = 0L;  for (long i = 0; i <= 300000000000L; i++) {  sum += i;  }  System.*out*.println(sum);  long end = System.*currentTimeMillis*();  System.*out*.println("耗费时间为：" + (end - start));//9536 92259  } } class ForkJoinSum extends RecursiveTask<Long>{  private long start;  private long end;  private static final long *THRESHOLD* = 10000L;//临界值  public ForkJoinSum(long start,long end){  this.start = start;  this.end = end;  }  @Override  protected Long compute() {  //Fork/Join将大任务不断拆分成小任务执行，最后将小任务执行的结果汇总后返回  long length = end - start;  if (length <= *THRESHOLD*){  long sum = 0L;  for (long i = start; i <= end; i++) {  sum += i;  }  return sum;  }else{  long middle = (start + end) / 2;  ForkJoinSum left = new ForkJoinSum(start, middle);  left.fork();  ForkJoinSum right = new ForkJoinSum(middle + 1, end);  right.fork();  return left.join() + right.join();  }  } } |

# Java 8

## Lambda表达式

Lambda表达式（Lambda Expressions） 是一个匿名函数，我们可以把 Lambda 表达式理解为是一段可以传递的代码（将代码像数据一样进行传递）。使用它可以写出更简洁、更灵活的代码。作为一种更紧凑的代码风格，使Java的语言表达能力得到了提升。

Lambda 表达式：在Java 8 语言中引入的一种新的语法元素和操作符。这个操作符为 “**->**” ， 该操作符被称为Lambda操作符或箭头操作符。它将 Lambda 分为两个部分：

**左侧：**指定了 Lambda 表达式需要的参数列表

**右侧：**指定了 Lambda 体，是抽象方法的实现逻辑，也即 Lambda 表达式要执行的功能。

**类型推断**

上述 Lambda 表达式中的参数类型都是由编译器推断得出的。Lambda 表达式中无需指定类型，程序依然可以编译，这是因为 javac 根据程序的上下文，在后台推断出了参数的类型。Lambda 表达式的类型依赖于上下文环境，是由编译器推断出来的。这就是所谓的“**类型推断**”。 虽然这里的类型推断省略了，但是类型检查在编译的时候仍然是有的。

|  |
| --- |
| **public** **void** method() {  Runnable r1 = **new** Runnable() {  **public** **void** run() {  System.***out***.println("run");  }  };  Runnable r2 = () -> System.***out***.println("run");  **new** Thread(r1).start();  **new** Thread(r2).start();  } |
| **public** **void** method2() **throws** Exception {  TreeSet<String> ts1 = **new** TreeSet<>(**new** Comparator<String>() {  **public** **int** compare(String s1,String s2) {  **return** compare(s1, s2);  }  });  TreeSet<String> ts2 = **new** TreeSet<>(  (s1, s2) -> Integer.*compare*(s1.length(),s2.length())  );  } |

### Lambda表达式语法

如果Lamdba只有1个参数，则传递参数的小括号可以省略；如果Lambda只有一条语句，return和大括号{}都可以省略，如果省略则一起省略，或一起不能省略。

**无参无返回值**

|  |
| --- |
| **interface** Animal{  **void** start();  } |
| Animal animal = () -> {System.***out***.println("start");};  animal.start(); |

**有参数，无返回值**

|  |
| --- |
| **interface** Animal<T1,T2>{  **void** start(String string,Integer integer);  } |
| //如果在声明时定义了泛型，则括号内的数据类型可以省略，即类型推断  Animal<String,Integer> animal = (string,Integer)  -> {  System.***out***.println(string);  System.***out***.println(Integer);  };  animal.start("start",1); |

**无参有返回值**

|  |
| --- |
| **interface** Animal{  String start();  } |
| **public** **void** method3() **throws** Exception {  Animal animal = () -> "start";  System.***out***.println(animal.start());  } |

**有参有返回值**

|  |
| --- |
| **interface** Animal<T1,T2>{  String start(String string,Integer integer);  } |
| **public** **void** method3() **throws** Exception {  Animal<String,Integer> animal = (String string,Integer Integer)  -> {  String temp = string+Integer;  **return** temp;  };  System.***out***.println(animal.start("start", 1));  } |

### 函数式接口

函数式接口指的是只包含一个抽象方法的接口，可以使用Lambda表达式创建该接口的对象。（若Lambda表达式抛出一个受检异常，那么该异常需要在目标接口的抽象方法上进行声明）。可以在一个接口上使用 @FunctionalInterface 注解，这样做可以检查它是否是一个函数式接口。同时 javadoc 也会包含一条声明，说明这个接口是一个函数式接口。在java.util.function包下定义了java 8 的丰富的函数式接口。

Java从诞生日起就是一直倡导“一切皆对象”，在java里面面向对象(OOP)编程是一切。但是随着python、scala等语言的兴起和新技术的挑战，java不得不做出调整以便支持更加广泛的技术要求，也即java不但可以支持OOP还可以支持OOF（面向函数编程）

在函数式编程语言当中，函数被当做一等公民对待。在将函数作为一等公民的编程语言中，Lambda表达式的类型是函数。但是在Java8中，有所不同。在Java8中，Lambda表达式是对象，而不是函数，它们必须依附于一类特别的对象类型——函数式接口。

简单的说，在Java8中，Lambda表达式就是一个函数式接口的实例。这就是Lambda表达式和函数式接口的关系。也就是说，只要一个对象是函数式接口的实例，那么该对象就可以用Lambda表达式来表示。所以以前用匿名内部类表示的现在都可以用Lambda表达式来写。

作为参数传递 Lambda 表达式：为了将 Lambda 表达式作为参数传递，接收Lambda 表达式的参数类型必须是与该 Lambda 表达式兼容的函数式接口的类型。

**Java 内置四大核心函数式接口**

|  |  |  |  |
| --- | --- | --- | --- |
| **函数式接口** | **参数类型** | **返回类型** | **用途** |
| **Consumer<T>**  **消费型接口** | T | void | 对类型为T的对象应用操作，包含方法：void accept(T t) |
| **Supplier<T>**  **供给型接口** | 无 | T | 返回类型为T的对象，包含方法T get() |
| **Function<T, R>**  **函数型接口** | T | R | 对类型为T的对象应用操作，并返回结果。结果是R类型的对象。包含方法R apply(T t) |
| **Predicate<T>**  **断定型接口** | T | boolean | 确定类型为T的对象是否满足某约束，并返回 boolean 值。包含方法boolean test(T t) |
| **其他接口** | | | |
| **BiFunction<T, U, R>** | T, U | R | 对类型为 T, U 参数应用操作，返回 R 类型的结果。包含方法为R apply(T t, U u); |
| **UnaryOperator<T>**  **(Function子接口)** | T | T | 对类型为T的对象进行一元运算，并返回T类型的结果。包含方法为T apply(T t); |
| **BinaryOperator<T>**  **(BiFunction 子接口)** | T, T | T | 对类型为T的对象进行二元运算，并返回T类型的结果。包含方法为T apply(T t1, T t2); |
| **BiConsumer<T, U>** | T, U | void | 对类型为T, U 参数应用操作。包含方法为void accept(T t, U u) |
| **BiPredicate<T,U>** | T,U | boolean | 包含方法为boolean test(T t,U u) |
| **ToIntFunction<T>**  **ToLongFunction<T>**  **ToDoubleFunction<T>** | T | int  long  double | 分别计算int、long、double、值的函数 |
| **IntFunction<R>**  **LongFunction<R>**  **DoubleFunction<R>** | int  long  double | R | 参数分别为int、long、double 类型的函数 |

### 方法、构造器和数组引用

#### 方法的引用

当要传递参数给Lambda体的操作，可以使用方法引用。方法引用就是Lambda表达式，就是函数式接口的一个实例，通过方法的名字来指向一个方法，可以认为是Lambda表达式的一个语法糖。实现抽象方法的参数列表和返回值类型，必须与方法引用的方法的参数列表和返回值类型保持一致。

方法引用：使用操作符 “::” 将类(或对象) 与 方法名分隔开来。

如下三种主要使用情况：

对象：：实例方法名

类：：静态方法名

类：：实例方法名

|  |
| --- |
| **interface** Animal<T>{  **public** **void** start(String string);  } |
| **public** **void** method4() **throws** Exception {  Animal<String> animal = System.***out***::println;  animal.start("hello");  } |

**当函数式接口方法的第一个参数是需要引用方法的调用者，并且第二个参数是需要引用方法的参数(或无参数)时：ClassName::methodName**

#### 构造器的引用

格式：ClassName::new

与函数式接口相结合，自动与函数式接口中方法兼容。可以把构造器引用赋值给定义的方法，要求构造器参数列表要与接口中抽象方法的参数列表一致，且方法的返回值即为构造器对应类的对象。

#### 数组引用

格式：type[]::new

## Stream流

Stream是数据渠道，用于操作数据源（集合、数组等）所生成的元素序列。集合讲的是数据，Stream讲的是计算。

Stream (java.util.stream) 把真正的函数式编程风格引入到Java中。这是目前为止对Java类库最好的补充，因为Stream API可以极大提供Java程序员的生产力，让程序员写出高效率、干净、简洁的代码。

Stream 是Java8 中处理集合的关键抽象概念，它可以指定你希望对集合进行的操作，可以执行非常复杂的查找、过滤和映射数据等操作。 使用Stream 对集合数据进行操作，就类似于使用 SQL 执行的数据库查询。也可以使用Stream 来并行执行操作。简言之，Stream 提供了一种高效且易于使用的处理数据的方式。

**使用Stream的原因**

实际开发中，项目中多数数据源都来自于Mysql，Oracle等。但现在数据源可以更多了，有MongoDB，Redis等，而这些NoSQL的数据就需要java层面去处理。

**并行流和串行流**

并行流：将一个内容分成多个数据块，并用不同的线程分别处理每个数据块的流

Java 8中将并行进行了优化，我们可以很容易的对数据进行并行操作。Stream可以声明性地通过 parallel与sequential方法在并行流与顺序流之间进行切换。

### Stream特点

Stream不是集合，无法存储元素；

Stream不会改变源对象，而是返回一个持有结果的新Stream；

Stream操作是延迟执行的，必须知道有哪些数据才会继续往下执行，即会等到需要结果时才执行；

Stream只执行一次，如果要继续操作则需要重新获取Stream对象；

类似高级的iterator，单向不可重复，数据只能遍历一次，但是可以并行化数据。

### 创建Stream

#### 通过Collection

Java 8中的Collection接口被扩展，提供了两个获取流的方法。

|  |  |
| --- | --- |
| **Collection接口方法摘要** | |
| default Stream<E> | **stream**()返回一个顺序流 |
| default Stream<E> | **parallelStream**()返回一个并行流 |

|  |  |
| --- | --- |
| **BaseStream接口方法摘要** | |
| S extends BaseStream<T, S> s | **parallel**()返回一个并行流 |

|  |
| --- |
| public void build() {  // 最常用集合的stream()来创建流  Stream<Integer> stream = numList.stream(); }  public void parallel(){  Stream<Integer> parallelStream = numList.parallelStream();  Stream<Integer> parallelStream2 = numList.stream().parallel();  parallelStream.forEach(x -> System.*out*.print(x + "-"));  numList.forEach(x -> System.*out*.print(x + "-")); } |

#### 通过Arrays

Java 8中的Arrays的静态方法Stream可以获取Stream流。

|  |  |
| --- | --- |
| **Arrays接口方法摘要** | |
| static<T> Stream<T> | **stream**(overload)返回一个顺序流 |
| static XxxStream | **stream**(overload)返回一个顺序流，Xxx包括Int、Long、Double |

|  |
| --- |
| public void build() {  Stream<Integer> stream = Arrays.*stream*(numArray); } |

#### 通过Stream

可以调用Stream的静态方法

|  |  |
| --- | --- |
| **Stream接口方法摘要** | |
| static<T> Stream<T> | **of**(T...values)返回一个顺序流 |
| static<T> Stream<T> | **iterate**(T seed, UnaryOperator<T> f)返回无限有序连续的流 |
| static<T> Stream<T> | **generate**(Supplier<T> s)返回无限顺序无序流 |

|  |
| --- |
| public void build() {  Stream<Integer> stream = Stream.*of*(this.numArray);  Stream<String> stream2 = Stream.*generate*(() -> "test");  Stream<Integer> stream3 = Stream.*iterate*(0, n -> n + 10);  Stream<Integer> stream4 = StreamSupport.*stream*(numList.spliterator(), false); } |

### 中间操作

一个中间操作链，对数据源的数据进行处理。多个**中间操作**可以连接起来形成一个**流水线**，除非流水线上触发终止操作，否则**中间操作不会执行任何的处理**！而在**终止操作时一次性全部处理，称为“惰性求值”**。

#### 筛选与切片

|  |  |
| --- | --- |
| **Stream接口方法摘要** | |
| Stream<T> | **fileter**(Predicate<? super T> predicate)返回与指定Lambda匹配的元素组成的流 |
| Stream<T> | **distinct**()返回由该流的元素的hashCode和equals方法去除重复元素组的流 |
| Stream<T> | **limit**(long maxSize)返回长度不超过maxSize的流 |
| Stream<T> | **skip**(long n)跳过元素，返回不包含前n个元素的流 |

|  |
| --- |
| public void filter() {  Stream<Integer> stream = numList.stream().filter(x -> x < 16);  stream.forEach(System.*out*::print); }  public void distinct(){  Stream<Integer> stream = numList.stream().distinct(); }  public void limit(){  Stream<Integer> stream = numList.stream().limit(3); } public void skip(){  // skip()方法丢弃掉前n个元素的新流，和limit()一起使用可以完成分页的功能  // stream.skip(pageNumber\*pageSize).limit(pageSize)  Stream<Integer> stream = numList.stream().skip(3); } |

#### 映射与转换

|  |  |
| --- | --- |
| **Stream接口方法摘要** | |
| <R> Stream<R> | **map**<Function<? supper T,? extends R> mapper) 接收一个函数作为参数，该函数会被应用到每个元素上，并将其映射成一个新的元素 |
| XxxStream | **mapToXxx**(<ToXxxFunction<? super T> mapper) 接收一个函数作为参数，该函数会被应用到每个元素上，产生一个新的 XxxStream，Xxx包括Int、Long、Double |
| <R> Stream<R> | **flatMap**(Function<? supper T,? extends Stream<? extends R>> mapper) 接收一个函数作为参数，将流中的每个值都换成另一个流，然后把所有流连接成一个流 |
| XxxStream | **flatMapToXxx**(<Function<? supper T,? extends XxxStream> mapper)返回XxxStream，Xxx包括Int、Long、Double |

|  |
| --- |
| public void map() {  Stream<String> stream = numList.stream().map(x -> x + " ");  stream.forEach(System.*out*::print); }  public void mapToXxx(){  IntStream intStream = numList.stream().mapToInt(Integer::intValue);  intStream.forEach(System.*out*::println); }  public void flatMap() {  ArrayList<List<Integer>> arrayLists = new ArrayList<>();  arrayLists.add(numList);  arrayLists.add(numList);  Stream<Integer> stream = arrayLists.stream().flatMap(List::stream); }  public void flatMapToXxx(){  ArrayList<List<Integer>> arrayLists = new ArrayList<>();  arrayLists.add(numList);  arrayLists.add(numList);  IntStream stream = arrayLists.stream().flatMapToInt(  list -> list.stream().mapToInt(Integer::intValue)); } |

#### 排序与范围

|  |  |
| --- | --- |
| **Stream接口方法摘要** | |
| Stream<T> | **sorted**(overload)返回一个新流，按自然排序或定制排序进行排序 |

|  |  |
| --- | --- |
| **XxxStream接口方法摘要** | |
| XxxStream | **range**(int startInclusive, int endExclusive) 获取集合中指定范围的元素进行操作 |

|  |
| --- |
| public void sorted() {  Stream<Integer> stream = numList.stream().sorted((x, y) -> y - x); }  public void range() {  int index = 5;  IntStream.*range*(2, numList.size()).forEach(i -> {  if (i < index) {  System.*out*.println(numList.get(i));  }  }); } |

### 终止操作

执行中间操作链，并产生任何不是流的值的结果，之后就不再被使用。

#### 匹配与查找

|  |  |
| --- | --- |
| **Stream接口方法摘要** | |
| boolean | **allMatch**()检查是否匹配所有元素 |
| boolean | **anyMatch**()检查是否至少匹配一个元素 |
| boolean | **noneMatch**()检查是否没有匹配的元素 |
| Optional<T> | **findFirst**()返回第一个元素 |
| Optional<T> | **findAny**()返回当前流中的任意元素 |
| long | **count**()返回流中元素的总个数 |
| Optional<T> | **max**()返回流中最大值 |
| Optional<T> | **min**()返回流中最小值 |
| void | **forEach**()内部迭代 |

|  |
| --- |
| public void match() {  // anyMatch()可以判定集合中是否还有匹配的元素  boolean isHas = numList.parallelStream().anyMatch(i -> i > 8);  // allMatch()在所有元素匹配时返回true  boolean allHas = numList.parallelStream().allMatch(i -> i > 8);  // noMatch()在没有元素匹配时返回true  boolean noHas = numList.parallelStream().noneMatch(i -> i > 8);  System.*out*.println("存在该元素：" + isHas + "\n"  + "所有元素符合该条件：" + allHas + "\n"  + "所有元素都不符合该条件：" + noHas); } |

|  |
| --- |
| public void find() {  // findFirst()返回非空集合中的第一个值，通常与filter方法结合起来使用  Integer firstNumber = numList.stream().findFirst().orElse(0);  // findAny()只要找到任何一个匹配的元素就返回，在并行流时十分有效  Integer randomNumber = numList.parallelStream().findAny().orElse(0); } |

|  |
| --- |
| public void statistics() {  int count = numList.size();  int sum = numList.stream().mapToInt(x -> x).sum();  Double avg = numList.stream()  .collect(Collectors.*averagingInt*(Integer::intValue));  Integer max = numList.stream().max(Integer::compareTo).orElse(0);  Integer min = numList.stream().min(Integer::compareTo).orElse(0);  System.*out*.println(count + ":" + sum + ":" + avg + ":" + max + ":" + min); } |

#### 归约

|  |  |
| --- | --- |
| **Stream接口方法摘要** | |
| Optional<T>/T | **reduce**(overload)将流中元素反复结合起来，得到一个值 |

|  |
| --- |
| public void reduce() {  //求和  Integer sum = numList.stream().reduce(Integer::*sum*).orElse(0);  //含有初始标识的，求和  sum = numList.stream().reduce(10, Integer::*sum*);  //对元素的长度进行求和，类似于一个累加器，会被重复调用)  sum = numList.stream()  .reduce(0, (total, y) -> total + y.toString().length(), Integer::*sum*);  //简化一下，对元素长度进行求和。  sum = numList.stream().map(Objects::*toString*).mapToInt(String::length).sum(); } |

#### 收集

|  |  |
| --- | --- |
| **Stream接口方法摘要** | |
| <R,A> R/<R> R | **collect**(Collector c)将流转换为其他形式 |

Collector 接口中方法的实现决定了如何对流执行收集的操作(如收集到 List、Set、Map)。另外， Collectors 实用类提供了很多静态方法，可以方便地创建常见收集器实例。

collect方法将流中的元素聚合，可以转换为list、set或string。

|  |
| --- |
| public void collect() {  List<Integer> list = numList.stream()  .filter(x -> x > 8)  .collect(Collectors.*toList*());  Set<Integer> set = numList.stream()  .filter(x -> x > 8)  .collect(Collectors.*toSet*());  TreeSet<Integer> treeSet = numList.stream()  .filter(x -> x > 8)  .collect(Collectors.*toCollection*(TreeSet::new));  String string = numList.stream()  .map(x -> x + " ").collect(Collectors.*joining*());  String stringJoin = numList.stream()  .map(String::*valueOf*).collect(Collectors.*joining*("-","[","]")); } |

使用Collectors.toMap方法可以实现将结果集收集到指定类型的map中。

|  |
| --- |
| public void toMap() {  Map<Integer, Integer> map = numList.stream()  .collect(Collectors.*toMap*(Integer::intValue, Integer::intValue));  System.*out*.println(map);  // 已具体的元素来作为生成的map的value  Map<Integer, Integer> map2 = numList.stream()  .collect(Collectors.*toMap*(Integer::intValue, Function.*identity*()));  System.*out*.println(map2);  // 生成的map的key冲突时，可以通过第三个参数来指定冲突时保留的值  Map<Integer, Integer> map3 = numList.stream()  .collect(Collectors.*toMap*(  x -> x.toString().length(),  Function.*identity*(),  (nowValue, newValue) -> nowValue));  System.*out*.println(map3);  // 可以通过第四个参数来生成指定的map  TreeMap<Integer, Integer> map4 = numList.stream()  .collect(Collectors.*toMap*(  x -> x.toString().length(),  Function.*identity*(),  (nowValue, newValue) -> newValue,  TreeMap::new));  System.*out*.println(map4); } |

使用Collectors.groupingBy()将集合按照指定的函数来分组，分类函数返回布尔值的函数时，使用partitioningBy()更有效。

|  |
| --- |
| public void groupingBy() {  Map<Integer, List<Integer>> groupMap = numList.stream()  .collect(Collectors.*groupingBy*(x -> x.toString().length()));  System.*out*.println("groupMap：" + groupMap); }  public void partitioningBy() {  Map<Boolean, List<Integer>> partitionMap = numList.stream()  .collect(Collectors.*partitioningBy*(x -> x == 16));  System.*out*.println(partitionMap); } |

使用Collectors.counting()可以统计不同组的元素个数，适用于groupingBy()和partitioningBy()。

|  |
| --- |
| public void counting() {  Map<Integer, Long> countMap = numList.stream()  .collect(Collectors.*groupingBy*(  x -> x.toString().length(),  Collectors.*counting*()));  System.*out*.println(countMap);  Map<Boolean, Long> countMap2 = numList.stream()  .collect(Collectors.*partitioningBy*(  x -> x >= 128,  Collectors.*counting*()));  System.*out*.println(countMap2); } |

使用Collectors.summingInt方法可以在分组后对每组元素进行求和

|  |
| --- |
| public void summingInt() {  Map<Integer, Integer> sumNumMap = numList.stream()  .collect(Collectors.*groupingBy*(x -> x.toString().length(),  Collectors.*summingInt*(x -> x)));  System.*out*.println(sumNumMap);  Map<Boolean, Integer> sumNumMap2 = numList.stream()  .collect(Collectors.*partitioningBy*(  x -> x >= 128,  Collectors.*summingInt*(x -> x)));  System.*out*.println(sumNumMap2); } |

使用Collectors.maxBy方法可以在分组后获取每组的最大值，Collectors.minBy方法可以在分组后获取每组的最小值。

|  |
| --- |
| public void maxBy() {  Map<Integer, Optional<Integer>> maxMap = numList.stream()  .collect(Collectors.*groupingBy*(  x -> x.toString().length(),  Collectors.*maxBy*(Comparator.*comparing*(x -> x))));  System.*out*.println(JSON.*toJSONString*(maxMap));  Map<Boolean, Optional<Integer>> maxMap2 = numList.stream()  .collect(Collectors.*partitioningBy*(  x -> x >= 128,  Collectors.*maxBy*(Comparator.*comparing*(x -> x))));  System.*out*.println(JSON.*toJSONString*(maxMap2)); }  public void minBy() {  Map<Integer, Optional<Integer>> minMap = numList.stream()  .collect(Collectors.*groupingBy*(  x -> x.toString().length(),  Collectors.*minBy*(Comparator.*comparing*(x -> x))));  System.*out*.println(JSON.*toJSONString*(minMap));  Map<Boolean, Optional<Integer>> minMap2 = numList.stream()  .collect(Collectors.*partitioningBy*(  x -> x >= 128,  Collectors.*minBy*(Comparator.*comparing*(x -> x))));  System.*out*.println(JSON.*toJSONString*(minMap2)); } |

使用Collectors.mapping方法会将结果应用到另一个收集器上，获取分组中最大值的值

|  |
| --- |
| public void mapping(){  Map<Integer, Optional<Integer>> map = numList.stream()  .collect(Collectors.*groupingBy*(  x -> x.toString().length(),  Collectors.*mapping*(x -> x,  Collectors.*maxBy*(Comparator.*comparing*(Integer::*valueOf*)))));  System.*out*.println(JSON.*toJSONString*(map));  Map<Boolean, Optional<Integer>> map2 = numList.stream()  .collect(Collectors.*partitioningBy*(  x -> x >= 128,  Collectors.*mapping*(x -> x,  Collectors.*maxBy*(Comparator.*comparing*(Integer::*valueOf*)))));  System.*out*.println(JsonUtils.*toJSONString*(map2)); } |

使用Collectors.summarizingInt方法可以将返回类型是int、long、double的元素收集，获取其函数值的总和、平均值、总数、最大值和最小值。

|  |
| --- |
| public void summarizingInt(){  IntSummaryStatistics summaryStatistics = numList  .stream()  .collect(Collectors.*summarizingInt*(Integer::intValue));  System.*out*.println(summaryStatistics);  Map<Integer, IntSummaryStatistics> map = numList.stream()  .collect(Collectors.*groupingBy*(  x -> x.toString().length(),  Collectors.*summarizingInt*(x -> x)  ));  System.*out*.println(JsonUtils.*toJSONString*(map)); } |

## Optional类

|  |  |
| --- | --- |
| **Optional类方法摘要** | |
| boolean | **allMatch**()检查是否匹配所有元素 |

empty方法创建一个空的Optional对象。

|  |
| --- |
| public void empty() {  Optional<Room> empty = Optional.*empty*();  System.*out*.println(empty); } |

Optional类的of和ofNullable方法的区别在于of方法中传入null会抛出空指针异常，而ofNullable则不会，所以不确定对象是否为null时尽量使用ofNullable方法。

|  |
| --- |
| public void of() {  Optional<Room> room = Optional.*of*(this.room);  Optional<Room> emptyRoom = Optional.*of*(this.emptyRoom); }  public void ofNullable() {  Optional<Room> room = Optional.*ofNullable*(this.room);  Optional<Room> emptyRoom = Optional.*ofNullable*(this.emptyRoom); } |

ifPresent方法可以判断传入Optional中的对象是否不为空，为空则返回false，反之返回true。

|  |
| --- |
| public void ifPresent() {  Optional<Room> room = Optional.*ofNullable*(this.room);  boolean notNull = room.isPresent(); } |

get方法可以获取传入Optional实例中的对象，但是如果该对象为null时会抛出空指针异常。

|  |
| --- |
| public void get() {  Optional<Room> room = Optional.*ofNullable*(this.room);  Room newRoom = room.get(); } |

orElse方法也可以获取该对象，和get不同的是该对象如果为空时，则返回该方法的参数对象(默认值)。

|  |
| --- |
| public void orElse() {  Optional<Room> room = Optional.*ofNullable*(this.room);  Room newRoom = room.orElse(new Room(4, 5, 6));  System.*out*.println(newRoom); } |

orElseGet方法和orElse方法的区别在于当该对象不为空时，orElse方法仍然会创建参数中的对象，而orElseGet方法则不会再创建参数中的对象，因此会有微小的性能差异。

|  |
| --- |
| public void orElseGet(){  Optional<Room> room = Optional.*ofNullable*(this.room);  Room newRoom = room.orElseGet(() -> new Room(4,5,6));  System.*out*.println(newRoom); } |

orElseThrow方法在该对象为null时会抛出指定的异常。

|  |
| --- |
| public void orElseThrow(){  Optional<Room> room = Optional.*ofNullable*(this.emptyRoom);  Room newRoom = room.orElseThrow(ArrayIndexOutOfBoundsException::new);  System.*out*.println(newRoom); } |

map方法对对象进行转换并将转换后的对象包装在Optional中，flatMap可以将Function函数式接口作为参数来转换为指定的Optional封装的对象。

|  |
| --- |
| public void map(){  Integer integer = Optional.*ofNullable*(this.room)  .map(Room::getLength).orElse(0);  System.*out*.println(integer); }  public void flatMap(){  Integer integer = Optional.*ofNullable*(this.room)  .flatMap(x -> Optional.*ofNullable*(x.getLength()))  .orElse(0);  System.*out*.println(integer); } |

filter方法接收一个Predicate函数式接口作为参数来实现过滤功能，该函数返回的是true则正常返回，如果该函数返回的是false则返回一个空的Optional。

|  |
| --- |
| public void filter(){  Room room = Optional.*ofNullable*(this.room)  .filter(x -> x.getLength() > 10)  .orElse(new Room());  System.*out*.println(room); } |

# Question

## 基础难题

### 判断奇偶性

|  |
| --- |
| **public** **class** Example {  //求余操作的本质是(a / b) \* b + (a % b) == a，因此最后余数的符号与分子保持一致  **public** **static** **boolean** isOdd1(**int** i){  //此时无法判断负数的奇偶性  **return** i % 2 == 1;  }  **public** **static** **boolean** isOdd2(**int** i){  //此时解决了负数的奇偶性  **return** i % 2 != 0;  }  **public** **static** **boolean** isOdd(**int** i){  //此时判断的效率最高，因为与操作之后只会留下最后一位，如果是0则为偶，如果是1则为奇  **return** (i & 1) != 0;  }  } |

### 浮点型的精确求值/找零时刻

|  |
| --- |
| **import** java.math.BigDecimal;  **public** **class** Example {  /\*  在需要精确答案的地方，要避免使用 float 和 double；对于货币计算，要使用 int、long 或 BigDecimal。  \*/  **public** **static** **void** main(String args[]){  //浮点型表示的是近似值，而不是和整数型一样表示的是准确值，因此最后的结果是个无限接近的近似值  System.***out***.println(2.00 - 1.10);  //浮点运算在一个范围很广的值域上提供了很好的近似，但是它通常不能产生精确的结果，其不适合货币计算  System.***out***.printf("%.2f%n",2.00 - 1.10);  //货币运算时尽量将其化为最小值的货币单位，使其变为整数型的运算  System.***out***.println((200 - 110) + "分");  //一定要用BigDecimal(String)构造器，而千万不要用 BigDecimal(double)，后者还是一个近似值。这种方式速度慢一些  System.***out***.println(**new** BigDecimal("2.00").subtract(**new** BigDecimal("1.10")));  }  } |

### long型数整除/长整除

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String args[]){  //虽然最后的结果并不会超出long的存值范围，但是运算时是以int型运算的，然后将int型的结果转换为long型  //此时int型的结果在转换为long型之前发生了溢出，因此最后的结果不是1000，而是5  **final** **long** MICROS\_PER\_DAY = 24 \* 60 \* 60 \* 1000 \* 1000;  //只要将其中的一个数显式声明为long型，就会在运算时就将其所有数值转换为long型  **final** **long** MICROS\_PER\_DAY2 = 24 \* 60 \* 60 \* 1000 \* 1000L;  //MILLIS\_PER\_DAY的结果并没有溢出int的范围，但是最好也要将其中一个数显式声明为long型  **final** **long** MILLIS\_PER\_DAY = 24 \* 60 \* 60 \* 1000L;  System.***out***.println(MICROS\_PER\_DAY/MILLIS\_PER\_DAY);  System.***out***.println(MICROS\_PER\_DAY2/MILLIS\_PER\_DAY);  }  } |

### long型的假象/1和l分不清

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String args[]){  //long型声明时最好用大写的L，小写的l容易与1弄混，导致结果异常  System.***out***.println(12345+5432l);  //声明变量时也不要只用一个小写的l来表示这个变量  **int** l = 1;  }  } |

### 十六进制的难题？？？

|  |
| --- |
| /\*  \* 置位（set） ：将某一位设置为1  \* 复位（reset）：将某一位设置为0  \*/  **public** **class** Example {  **public** **static** **void** main(String args[]){  //混合类型的计算可能会产生混淆，尤其是十六进制和八进制字面常量无需显式的减号符号就可以表示负的数值  System.***out***.println(Long.*toHexString*(0x100000000L + 0xcafebabe));  /\* 解决这个问题只要在右操作数的末位后添加L就好  \* 1111111  0xffffffffcafebabeL  + 0x0000000100000000L  ---------------------  0x00000000cafebabeL  \*/  System.***out***.println(Long.*toHexString*(0x100000000L + 0xcafebabeL));  System.***out***.println(Long.*toHexString*(0x100000000L));  System.***out***.println(Long.*toHexString*(0xcafebabe));  }  } |

### 多重强制转型？？？

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String args[]){  /\*  \* 如果最初的数值类型是有符号的，那么就执行符号扩展；如果它是 char，那么不管它将要被转换成什么类型，都执行零扩展  \* 因为 byte 是一个有符号的类型，所以在将 byte 数值-1 转换成 char 时，会发生符号扩展。  \* 作为结果的 char 数值的 16 个位就都被置位了，因此它等于 2的16次方-1，即 65535。  \*/  System.***out***.println((**int**)(**char**)(**byte**) -1);  //如果将一个 char 数值 c 转型为一个宽度更宽的类型，可以考虑使用一个位掩码  **int** i = 'c' & 0xffff;  System.***out***.println(i);  **int** i2 = (**short**)'c';//转型将引起符号扩展  System.***out***.println(i2);  **byte** b = 50;  **char** c = (**char**)(b&0xff);  System.***out***.println(c);  }  } |

### 位运算/互换内容

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String args[]){  /\*  按位与&　 :　 两位全为１，结果为1，否则为0  按位或| : 两位有一个为1，结果为1，否则为0  按位异或^ : 两位一个为0,一个为1，结果为1，否则为0  按位取反~ : 0->1 ,1->0  \*/  **int** x = 1984; // (0x7c0)  **int** y = 2001; // (0x7d1)  //将xy异或即可交换数值  x = x ^ y;  y = y ^ x;  x = y ^ x;  System.***out***.println("x= " + x + "; y= " + y);  /\* 赋值运算符的运算顺序是从右往左  Java 中 x^= y^= x^= y 的实际行为  int tmp1 = x ; // x 在表达式中第一次出现  int tmp2 = y ; // y 的第一次出现  int tmp3 = x ^ y ; // 计算 x ^ y  x = tmp3 ; // 最后一个赋值：存储 x ^ y 到 x  y = tmp2 ^ tmp3 ; // 第二个赋值：存储最初的 x 值到 y 中  x = tmp1 ^ y ; // 第一个赋值：存储 0 到 x 中  最后实际的运算是x^x，结果为0  \*/  x^= y^= x^= y;  System.***out***.println("x= " + x + "; y= " + y);  }  } |

### 条件表达式

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String args[]){  /\*  1、如果第二个和第三个操作数具有相同的类型， 那么它就是条件表达式的类型,通过绕过混合类型计算来避免麻烦。  2、如果一个操作数的类型是T，T表示byte、short或char，而另一个操作数是一个int类型的常量表达式，  它的值是可以用类型T表示的，那么条件表达式的类型就是T。  3、否则，将对操作数类型运用二进制数字提升，而条件表达式的类型就是第二个和第三个操作数被提升之后的类型。  \*/  **char** x = 'X';  **int** i = 0;  //调用的是重载的println(char)  System.***out***.println(i==0 ? x : 0);  //调用的是重载的println(int)  System.***out***.println(i!=0 ? i : x);  }  } |

### 复合赋值

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String args[]){  //复合赋值表达式自动地将它们所执行的计算的结果转型为其左侧变量的类型  /\*  尽量不要将复合赋值操作符作用于 byte、short 或 char 类型的变量上。在将复合赋值操作符  作用于 int 类型的变量上时， 要确保表达式右侧不是 long、 float或 double 类型。在将  复合赋值操作符作用于 float 类型的变量上时，要确保表达式右侧不是 double 类型  \*/  **short** x = 0;  **int** i = 123456;  //x = x + i;这种会报错，因为int型的结果无法自动转换为short型  x += i;  //x += i;不会报错，因为运算完成后会进行强制类型转换，但是因为123456超出short的范围，所以会发生溢出  System.***out***.println(x);  }  } |

### 复合赋值/字符串相加

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String args[]){  /\*  \* 复合赋值操作符要求两个操作数都是原始类型的，例如 int，或包装了的原始类型，  \* 例如 Integer，但是有一个例外：如果在+=操作符左侧的操作数是 String类型的，  \* 那么它允许右侧的操作数是任意类型，在这种情况下，该操作符执行的是字符串连接操作。  \* 简单赋值操作符（=）允许其左侧的是对象引用类型，这就显得要宽松许多了：你可以使用  \* 它们来表示任何你想要表示的内容，只要表达式的右侧与左侧的变量是赋值兼容的即可。  \*/  Object x = "Buy ";  String i = "Effective Java!";  x = x + i;  System.***out***.println(x);  x += i;  System.***out***.println(x);  }  } |

### 字符数组/多态

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String args[]) {  String letters = "ABC";  **char**[] numbers = { '1', '2', '3' };  System.***out***.println(letters + " easy as " + numbers);  System.***out***.println(letters + " easy as " + String.*valueOf*(numbers));  System.***out***.print(letters + " easy as ");  // 此处的println方法是重载后的方法，并不是打印字符串，而是打印字符数组  System.***out***.println(numbers);  }  }  **class** ABC2 {  **public** **static** **void** main(String[] args) {  String letters = "ABC";  Object numbers = **new** **char**[] { '1', '2', '3' };  System.***out***.print(letters + " easy as ");  //此处的println方法是Object为参数的打印方法，最后的结果仍为地址  System.***out***.println(numbers);  }  } |

### 运算符优先级/字符串常量比较

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String args[]) {  **final** String pig = "length: 10";  **final** String dog = "length: " + pig.length();  **final** String cat = "length: 10";  //因为+的优先级高于==，所以先执行的字符串连接，再执行的判断  System.***out***. println("Animals are equal: " + pig == dog);  //在比较对象的引用时，应该优先使用equals方法而不是==  System.***out***.println("Animals are equal: " + pig.equals(dog));  //因为两者都是指向常量池的字符串常量，所以两者相同  System.***out***. println(pig == cat);  System.***out***.println(dog);  //dog因为赋值时是由字符串常量和不确定的字符串连接的，因此指向的是堆中指向常量池中的字符串的引用  //而cat直接指向常量池中的字符串，因此两者不相同  System.***out***. println(dog == cat);  }  } |

### 转义字符/编码表

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String args[]) {  // \u0022 是双引号的 Unicode 转义字符  System.***out***.println("a\u0022.length()+\u0022b".length());  //上面的语句实际上打印的是"a".length()+"b".length()，因为\u0022被转换成了双引号  //中间的+是算术运算符，执行的是1+1  System.***out***.println("a\".length()+\"b".length());  //此时\"表示双引号，在UTF-8中占用1个字符，因此最后结果是14  }  } |

### 字符串奶酪/编码表/字符集

|  |
| --- |
| **public** **class** Example {  //每当你要将一个 byte 序列转换成一个 String 时，你都在使用某一个字符集，不管你是否显式地指定了它。  //如果你想让你的程序的行为是可预知的，那么就请你在每次使用字符集时都明确地指定。  **public** **static** **void** main(String[] args) **throws** UnsupportedEncodingException {  //返回默认字符集，本机默认字符集是UTF-8  System.***out***.println(java.nio.charset.Charset.*defaultCharset*());  **byte** bytes[] = **new** **byte**[256];  //可以试着将256改为129，或其他数  **for** (**int** i = 0; i < 256; i++) {  bytes[i] = (**byte**) i;  System.***out***.println(bytes[i]);  }  //ISO-8859-1 是唯一能够让该程序按顺序打印从 0 到 255 的整数的缺省字符集，它更为大家所熟知的名字是 Latin-1[ISO-8859-1]。  String str = **new** String(bytes, "ISO-8859-1");  **for** (**int** i = 0, n = str.length(); i < n; i++)  System.***out***.println((**int**) str.charAt(i) + " ");  }  } |

### 转义字符/替换字符串内容

|  |
| --- |
| **public** **class** Example {  **public** **static** **void** main(String[] args) {  //问题在于String.replaceAll接受了正则表达式作为第一个参数，而非接受了字符序列字面常量。  System.***out***.println(Example.**class**.getName().replaceAll(".","/") + ".class");  //其正确行为与平台相关。不是所有的系统都使用斜杠符号来分隔层次结构的文件名组成的。  //要获取一个正在运行平台上的有效文件名，应使用正确的平台相关的分隔符号来代替斜杠符号。  System.***out***.println(Example.**class**.getName().replaceAll("\\.","/") + ".class");  //Pattern.quote静态方法，接受字符串作为参数，并可以添加必需的转义字符，  //它将返回一个正则表达式字符串，该字符串将精确匹配输入的字符串。  System.***out***.println(Example.**class**.getName().replaceAll(Pattern.*quote*("."),"/")+ ".class");  //File.separator被定为公共的String域，含平台相关文件名分隔符，平台不同会显示两种行为中的一种。  //此行报错System.out.println(Example.class.getName().replaceAll("\\.", File.separator) + ".class");  //Matcher.quoteReplacement，它将字符串转换成相应的替代字符串。  System.***out***.println(Example.**class**.getName().replaceAll("\\.",Matcher.*quoteReplacement*(File.***separator***)) + ".class");  //String.replace(CharSequence,CharSequence)，和String.replaceAll相同，但将模式和替代物都当作字面含义的字符串处理。  System.***out***.println(Example.**class**.getName().replace(".", File.***separator***) + ".class");  //早期版本的Java用下面的方法，完全不用正则表达式，而是使用String.replace(char,char)  System.***out***.println(Example.**class**.getName().replace('.', File.***separatorChar***) + ".class");  }  } |

### switch/API/基本数据类型

|  |
| --- |
| **public** **class** Example {  **private** **static** Random *rnd* = **new** Random();  **public** **static** **void** main(String[] args) {  StringBuffer word = **null**;  System.***out***.println(*rnd*.nextInt(2));  //rnd.nextInt(2)只会取值到0和1，因此永远不会执行到分支为2的选项  **switch**(*rnd*.nextInt(2)) {  //因为case后面都没有break，因此最后执行的结果都会被default覆盖掉  **case** 1: word = **new** StringBuffer('P');  **case** 2: word = **new** StringBuffer('G');  //StringBuffer有一个无参数的构造器，一个接受一个 String 作为字符串缓冲区初始内容的构造器，  //以及一个接受一个 int 作为缓冲区初始容量的构造器。换句话说，new StringBuffer('M')返回  //的是一个具有初始容量 77 的空的字符串缓冲区。因此要将单引号改为双引号  **default**: word = **new** StringBuffer('M');  }  word.append('a');  word.append('i');  word.append('n');  System.***out***.println(word);  //更简洁的语句同样的效果  System.***out***.println("PGM".charAt(*rnd*.nextInt(3)) + "ain");  }  } |

## 高级难题

### float类型底层存储

float类型数字在计算机中用4个字节存储。遵循IEEE-754格式标准：一个浮点数有2部分组成：底数m和指数e

底数部分 使用二进制数来表示此浮点数的实际值

指数部分 占用8bit的二进制数，可表示数值范围为0-255

但是指数可正可负，所以，IEEE规定，此处算出的次方必须减去127才是真正的指数。所以，float类型的指数可从-126到128

底数部分实际是占用24bit的一个值，但是最高位始终为1，所以，最高位省去不存储，在存储中占23bit

科学计数法。

格式：

SEEE EEEE EMMM MMMM MMMM MMMM MMMM MMMM

S表示浮点数正负

E指数加上127后的值得二进制数据

M底数

举例：

17.625在内存中的存储

首先要把17.625换算成二进制：10001.101

整数部分，除以2，直到商为0，余数反转。

小数部分，乘以2，直到乘位0，进位顺序取。

在将10001.101右移，直到小数点前只剩1位：

1.0001101 \* 2^4 因为右移动了四位

这个时候，我们的底数和指数就出来了

底数：因为小数点前必为1，所以IEEE规定只记录小数点后的就好。所以，此处的底数为：0001101

指数：实际为4，必须加上127(转出的时候，减去127)，所以为131。也就是10000011

符号部分是整数，所以是0

综上所述，17.625在内存中的存储格式是：

01000001 10001101 00000000 00000000

# Unknown

|  |  |
| --- | --- |
| **方法摘要** | |
| void | Stru |