JAVA常用API 5.0new

Application Programming Interface

**String**

***@see java.lang.String***

它是一个**类**，但是是一个特殊的**类**。

1.**概述**：字符串，用于描述一个字符序列.

是一个引用类型，默认初始值为null

String使用private final char value[]来实现字符串的存储,也就是说String对象创建之后,就不能再修改此对象中存储的字符串内容,就是因为如此,才说String类型是不可变的(immutable)。

String类有两种创建方法。

1. String str = “常量/字面量/常量链接”

称之为静态字符串，会在常量池中创建新对象，若存在此常量，则直接重用，所以该字符串创建第一次使用时只会生成1个对象。

1. String str = new String(“常量/字面量”)

使用new关键字会创造新的对象，不管常量池里是否存在。所以该字符串创建时，会在堆中和常量池里分别创建一个对象，共2个对象。  
　　1)号方式是编译期,2)号方式是运行期!

java对String类型重载了+操作符,可以直接使用+对两个字符串进行连接。

运行期调用String类的intern()方法可以向String Pool中动态添加对象。

**要点:**

1.1单独使用""引号创建的字符串都是常量,编译期就已经确定存储到String Pool中；  
1.2.使用new String("")创建的对象会存储到heap中,是运行期新创建的；  
1.3.使用只包含常量的字符串连接符如"aa" + "aa"创建的也是常量,编译期就能确定,已经确定存储到String Pool中；  
1.4.使用包含变量的字符串连接符如"aa" + s1创建的对象是运行期才创建的,存储在heap中；

2.**相关API**:

0)***int length()***;获取当前字符串的字符数/长度。

1)***int indexOf()***; 查找给定的字符串在当前字符串中第一次出现的位置。从0开始。返回值为给定的字符串第一个字符在当前字符串的下标。由于java大小写敏感，所以必须全部匹配上才可以返回具体的下标。没有找到会返回-1。

重载方法:

1.1)*int indexOf(String str ,int from)*;从当前字符串指定下标位置开始查找第一次出现给定字符串的位置。同样的，若没有找到会返回-1。

1.2)*int lastIndexOf(String str)*;查找当前字符串中最后一次出现给定字符串的位置，没有找到也会返回-1。

2)***String substring(int start,int end)***;获取字符串的部分内容，“含头不含尾”。

重载方法:

***String substring(int start)***

3)***String trim()***;去除当前字符串中两边的空白。

4)***char charAt(int index)***;获取当前字符串中指定位置对应的字符。

5)***boolean startsWith(String str)***;

判断当前字符串是否是以给定的字符串开头的。

***boolean endsWith(String str)***;

判断当前字符串是否是以给定的字符串结尾的。

6)***String toUpperCase()***;将当前字符串中的英文部分转换为全大写。

***String toLowerCase()***;将当前字符串中的英文部分转换为全小写。

7)***static String valueOf()***;字符串提供了若干的**静态方法**valueOf()(建议类名.来访问)，作用就是将其他类型转换为字符串。

**SringBuffer、StringBuilder**

它是一个**类**。

StringBuffer是线程(Thread)安全的，其功能与StringBuilder一致。

***@see java.lang.StringBuilder***

**概述**: 提供一个构造方法:StringBuiler(String str)

内部维护一个可变的字符数组。为了解决频繁修改字符串内容带来的性能损耗。提供了用于编辑字符串的相关方法:增、删、改、插。

1)增: ***StringBuilder append(String str)***;在当前字符串末尾追加给定字符串。

2)删: ***StringBuilder delete(int start,int end)***;

删除当前字符串中给定范围内的字符串。

***StringBuilder deleteCharAt(index)***;删除当前字符串中给定下标的字符。

3)改: ***StringBuilder replace(int start,int end,String str)***;

将当前字符串中给定范围内的字符串替换为给定的字符串。

4)插: ***StringBuilder insert(int index,String str)***;

将给定的字符串内容插入到指定位置处。

5)倒置: ***StringBuilder reverse()***;将当前字符串内容反转。

**String Regex**

**概述**:

**正则表达式**:通常用字符串表示。

要点:Object类是所有java类的父类，没有继承时默认继承Object类。

所以可以说万物皆对象。

1.**表达式含义**:

1)**集合**

[...]表示满足[]内条件的一个字符

**[a-z][A-Z][0-9]**

分别表示a-z之间，A-Z之间，0-9之间的任意一个字符

**[abc][^abc][a-z&&^[abc]]**

abc中的任意一个字符，除了abc以外的任意一个字符

在a-z之间除了abc以外的任意一个字符

2)**字符集**

**.** 表示任意一个字符 要表达 "." 需要转义字符，即" \." JAVA编译时需要再加一个转义符 即 "\\."

**\d** digit 任意一个数字[0-9]，同样的，java编译时需要转义 即 "\\d"

**\w** word 任意一个单词[a-zA-Z0-9\_],包括 "\_",同样的，java编译时需要转义 即 "\\w"

**\s** space 空白字符， 同样的，java编译时需要转义 即 "\\s"

**\D \W \S** 分别是非数字，非单词，非空格。java编译时需要转义。

**\n** 回车 **\r** 换行 **\\** 反斜杠"**\**" **\'** 单引号"**'**" **\"** 双引号"**"**"

3)**数量词**

**x? x\* x+** 分别表示0或1个x，0或多个x，1或多个x

示例说明：

* **runoo+b**，可以匹配 runoob、runooob、runoooooob 等，+ 号代表前面的字符必须至少出现一次（1次或多次）。
* **runoo\*b**，可以匹配 runob、runoob、runoooooob 等，\* 号代表字符可以不出现，也可以出现一次或者多次（0次、或1次、或多次）。
* **colou?r** 可以匹配 color 或者 colour，? 问号代表前面的字符最多只可以出现一次（0次、或1次）

**x{n} x{n,} x{n,m}** 分别表示n个x，n到多个x，n到m个x

2.1)**相关API(String)**:

1）***boolean matches(String regex)***;

验证当前字符串是否满足给定的正则表达式的格式要求。

需要注意:在java中，无论正则表达式是否使用了"^","$"都是全匹配验证。

2) ***String[] split(String regex)***;

将当前字符串中满足正则表达式的部分"切掉"，保留剩下的部分。

3) ***String replaceAll(String regex,String str)***;

将当前字符串中满足正则表达式部分的内容替换为给定的字符串。

***String replaceFirst(String regex,String str)***;

将当前字符串中第一段满足正则表达式部分的内容替换为给定的字符串。

2.2) **重写toString方法:**

***String toString()***

String str = obj.toString();

不重写返回的是 类名@地址

当我们需要使用当前类的toString方法时，通常我们就需要重写该方法。

具体返回字符串的格式没有严格要求，可根据将来实际开发中的需求而定。

但通常返回的字符串中应当包含当前类的属性值，以便了解当前对象的信息。

2.3) **重写equals方法**:

boolean equals(Object obj);

equals方法的作用是比较当前对象与给定对象的内容是否一样(属性值)

若不重写则使用Object提供的，其内部就是用"=="实现的，所以并无意义。

equals方法在比较两个对象内容一致时不一定要求所有属性值都一样，这要根据实际需求而定。

**Integer**

***@see* *java.lang.Integer/Byte/Short/Long/Float/Double***

***extends java.lang.Number***

***java.lang.Character/Boolean***

***extends java.lang.Object***

**概述:包装类**

由于java中有8个基本类型，他们不是以对象的形式存在，没有面向对象的特性，所以不能参与面向对象的开发。为了解决这个问题，java针对他们定义了8个包装类。

**相关API:**

1.***Integer i = new Integer(123)***; 将基本类型int转换为包装类Integer。

***int i = Integer(123).intValue()***; 将包装类Integer转换成基本类型int。

包装类提供静态方法***static Integer valueOf(Int i)***来新建包装类对象。

valueOf在某些情况下是可以重用对象减少内存开销的。所以比new来说要强一些

tips:valueOf有取值范围，在-128~127之间会重用对象，之外会新建对象。

2.数字类型的包装类都支持两个常量:

***MAX\_VALUE，MIN\_VALUE***

分别用来表示其对应的基本类型的取值范围。

3.***static Integer parseInt(String str)***/parseDouble/Long/Short/Byte……

包装类提供了一个静态方法，将字符串转换为包装类对应的基本类型。

要想进行转换有一个要求，就是字符串中的所有字符必须能正确表示要转换的数字。否则会抛出数字格式异常。小数字符串是不能转换为整数的，整数可以转为小数类型的。

4. **拆箱装箱**

java1.5以后推出了一个新的特性，自动拆装箱。

当编译器在编译源程序时发现我们需要一个基本类型，而实际给的是引用类型时会自动帮我们补全代码:

**将引用类型转换为基本类型(自动拆箱)。**

int i = new Integer(123);

所以，上面的代码在编译后的class文件中的样子:

int i = new Integer(123).intValue();

当发现需要引用类型，而实际给的是基本类型时，会补全代码:

**将基本类型转换为引用类型(自动装箱)**

Integer i = 123;

上面的代码在class文件中的样子:

Integer i = Integer.valueOf(123);

所以，自动拆装箱是编译器认可的，不是JVM认可的

**Date、SimpleDateFormat、Calendar**

1.**Date**:

这是一个**类**。

***@see java.util.Date***

**概述**:其每一个实例表示一个时间点

内部维护一个long值，该值是从UTC至当前Date表示的时间点之间的毫秒差

UTC时间 1970/01/01 00:00:00

**相关API:**

1)Date date = new Date();当前系统时间。

Date方法只用它表示时间，其他的方法不用它干。

2) long getTime();获取当前Date内部维护时间的毫秒值。

3) Date setTime(long l);表示传入的long值表示的时间。

4) Date(long l); 支持传入long值的构造方法，表示传入的long值表示的时间。

5)数据库相关:

数据库使用的Date ***@see java.sql.Date***

使用数据库Date需要将***@see java.util.Dates***转换，由于名称相同，故使用完全限定名来区分。

***java.sql.Date sDate***

***= new java.sql.Date(new java.util.Date().getTime())***;

2. **SimpleDateFormat**

这是一个**类**。

***@see java.text.SimpleDateFormat***

**概述**:该类的作用是把给定的字符串格式在Data和String之间相互转换。

***SimpleDateFormat("yyyy-MM-dd E HH:mm:ss")***;

yyyy表示4位数年份，MM/dd表示两位数月份/日期，也可以用M/d表示一位数的月份/日期，两位数自动扩充；HH表示24小时制，hh表示12小时制mm/ss表示分钟/秒，E代表星期几。

**相关API:**

1) ***String format(Date date)***;

按照创建SimpleDateFormat时指定的日期格式，将当前给定的date对象表示的时间转换为字符串

**示例说明：**

SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd E HH:mm:ss");

Date date = new Date();

String format = simpleDateFormat.format(date);//2018-08-30 星期四 17:14:43

2) ***Date parse(String str)***; 注意抛出异常。

将按照SimpleDateFormat指定的日期格式的字符串解析为Date对象并返回

3. **Calendar**

这是一个**抽象类**。

implements Serializable(可序列化),Comparable<Calendar>(可排序)

***@see*** ***java.util.Calendar***

**概述:**日历类，用于时间参数的操作。

常用的是格里高里历(Gregorian)，即阳历，默认创建出来的Calendar的实例表示当前系统时间。

**相关API:**

1) Calendar的getInstance可以根据当前系统所在地区创建合适的实现类，通常都是格里高里历

***Calendar calendar = Calendar.getInstance()***;

直接print输出可以但是不能直观的看到具体的时间。

Calendar还有一个实现类:

***GregorianCalendar gc = new GregorianCalendar(int year, int month, int day)***;其中 month和day可以用Calendar常量代替，月份从0开始计算.

2) ***Date getTime()***;

Calendar提供的getTime方法可以将其表示的时间以date的形式返回。

3) ***void*** ***setTime(Date date)***;

Calendar提供的setTime()方法，可以使当前calendar表示给定的date所表示的时间

4) Calendar提供了一个add方法，用来计算时间。

***void add(int filed,int value)***

Calendar时间分量:

Calendar.YEAR

Calendar.MONTH 月份从0开始计算，0对应Jaunary

Calendar.DAY\_OF\_MONTH 当前月中的目前天数

Calendar.DAY\_OF\_WEEK 星期几 1表示星期日

Calendar.HOUR\_OF\_DAY 小时，24小时制

Calendar.HOUR 小时，12小时制

……

对指定的时间分量加上给定的值，若value是负数，则减去指定的值。

如果一个类import static java.util.Calendar.\*，那么可以直接使用Calendar常量，而不需要是用类名.常量的形式书写。

5) Calendar提供了一个get方法，可以获取给定时间分量所对应的值。

***int get(int filed)***

6) ***int getActualMaximum(int field)***；获取时间分量能达到的最大值。

***int getActualMinimum(int field)***；获取时间分量能达到的最小值。

7) Calendar提供了一个set方法，允许我们对当前Calendar的各个时间分量进行单独设定.

***void set(int field, int value)***;

**Collection**

1.**Collection**

概述:集合，用于存放一组元素，提供若干功能，比数组更加方便。

***@see java.util.Collection***

它是一个**接口**，有两个子接口：

**List** 可重复，有序集合

**Set** 不可重复，大部分无序，也有有序实现的集合。

常用实现类:ArrayList、LinkedList、HashSet、TreeSet

ArrayList:一种可以动态增长和检索的索引序列，内部以数组实现;

LinkedList:一种可以在任何位置高效插入和删除的有序序列，内部以链表形式实现;

HashSet:一种没有重复元素的无序序列，依赖于HashCode散列码;

TreeSet:一种有序集，是SortedSet的实现类，也是NavigableSet的实现类；

LinkedHashSet:一种可以记住元素插入次序的集合；

PrioRityQueue:一种允许高效删除最小元素的队列；

**相关API**:

集合存放的元素是对象的引用，修改对象的值，输出集合也会显示改变后的值。

1) ***boolean add(E e)***;

将当前集合中添加元素，若成功添加则返回true，若失败则返回false。

2) ***int size()***;获取当前集合的元素个数。

3) ***void clear()***;清空集合元素

4) ***booolean contains(E e)***;判断集合是否有给定元素。

判断依据是看给定的元素与集合中现有元素是否存在equals比较为true的，有则认为包含。

5) 集合的实现类重写了toString方法

集合的toString()格式，[元素1.toString(),元素2.toString()....]

6) 集合的批量操作

6.1) ***boolean addAll(Collection c)***;

将给定的集合中的所有元素添加到当前集合中。当执行完毕后，当前集合中的元素发生了改变，就返回true。发生改变是指元素种类改变而不是个数改变。

6.2) ***boolean containsAll(Collection c)***;

判断当前集合中是否包含给定集合的所有元素，全部包含返回true。这里的判断依据是依靠元素的equals比较的。

6.3) ***boolean removeAll(Collection c);***

将给定的集合从集合中删除，成功删除使当前集合中的元素发生了改变返回true。删除集合的判断依据是依靠集合内元素的equals比较的，即将两个集合之间相同元素删除。

6.4) ***boolean retainAll(Collection c); 保留与c相同的元素***

将给定集合与集合相同的元素删除，成功删除使当前集合中的元素发生了改变返回true。

7) ***boolean remove(E e)***;

将给定的元素从集合中删除，成功删除返回true。删除元素的判断依据是删除当前集合中第一个与给定元素equals比较为true的。

8) 集合与数组的互相转换

8.1) 集合转换为数组:***Object[] toArray()***;

常用方法,使用具体类型: ***String[] array = c.toArray(new String[c.size()])***;

8.2) 数组转换为集合:使用Arrays的静态方法***asList(Collection c)***;

需要注意，数组只能转换为List集合，不能转换成Set。

原因在于：

8.2.1) Set大部分是无序实现。

8.2.2) Set不允许存放重复元素，所以转换后可能会丢失元素.

注意: 数组转换成集合后，对集合元素的任何操作也会影响原数组对应的元素；

数组转换的集合不能做任何增删操作。

8.3) 所有的集合都支持一个构造方法，参数要求传入另一个集合。这个构造方法的作用是：创建当前集合的同时将给定集合中的所有元素添加到当前集合中。

***List<T> new ArrayList<T>(Collecton c)***

2.**Iterator**

它是一个**接口**。

***@see java.util.iterator***

**概述:** 由于集合的具体实现类各不相同，set集合大部分实现又是无序的，所以不能像数组一样用下标的位置获取具体元素。

集合提供了统一获取集合元素的方式：遍历集合，遍历是使用迭代器完成的。

**iterator**是迭代器的接口，定义了用于遍历集合的相关方法，不同的集合都实现了一个可以遍历自身的迭代器实现类。

无需记住实现类的名字，只是当作迭代器使用即可。

迭代器迭代元素需要遵循：问，取，删，这些步骤。

其中删除不是必须的。

1)若需要**遍历集合**，可以调用集合的方法：

***Iterator iterator()***;

该方法会返回可以遍历当前集合的迭代器。

***Iterator it = c.iterator()***;

1.1) 问:***boolean hasNext()***;

询问迭代器，其遍历的集合是否还有元素没有遍历过，有则返回true。

1.2) 取:***E next()***;

取出集合的下一个元素。

1.3) 删:***remove()***;

在使用迭代器遍历集/合的过程中，不能使用集合的remove方法删除元素，否则会抛出异常。

不允许使用集合的方法修改其中元素数量。

可以使用迭代器提供的remove方法删除元素。遵循问取删，一次操作一个元素。

2) **for**

java1.5 以后推出了一个新特性：增强for循环。

又叫作：新循环、增强循环、for each。

新循环不能代替传统循环重复工作的需求。

新循环是用来遍历集合或者数组的。

***for(Object obj:array|collection){}***

obj的类型取决于array或者collection中元素的类型，遍历次数由数组或集合的长度决定。集合取出的元素类型写Object，或者知道具体类型。

新循环是编译器认可的，不是JVM认可的。

编译器编译源程序发现使用新循环遍历集合时，会将代码改变为迭代器方式遍历集合。

新循环本质上是迭代器遍历，遍历过程中不允许通过集合方式修改元素数量。

3) **ListIterator**

包含add等一些列方法的Iterator接口，可以从前后两个方向遍历集合。

void add(Object e);

将元素添加到迭代器所处位置的前面

3.**泛型**

1) ***<?>*** ?一般是一个字母，两个就***<?,?>***

泛型可以是各种类型,在使用时指定数据类型即可。

java1.5（5.0）开始支持的新特性；泛型是参数化类型，可以对一个类中的属性、方法的参数、方法的返回值进行约束。在使用时指定类型，只能是引用类型。

2) 常见面试题

当使用一个具有泛型的类时，若不指定，则按照默认的Object作为实际类型

3) **没有泛型数组（有例外）**

根本的原因是：数组在创建的时候必须知道内部元素的类型，而且一直都会记得这个类型信息，每次往数组里添加元素，都会做类型检查。但因为Java泛型是用**擦除（Erasure）**实现的，运行时类型参数会被擦掉。比如：

运行时，类型参数<String>都被擦掉，只有在最后读取内部元素的时候，才插入一个类型转换。看起来就像下面这样，

List l = new ArrayList<String>();

List.add("hello");

String str = (String)List.get(0);

所以，如果像下面这样初始化泛型数组的话，

List<String>[] l = new ArrayList<String>[10]; //Error

运行时编译器只能看到ArrayList，而看不到泛型的String类型参数。

数组由于无法确定所持有元素的类型，所以不允许初始化。

Java Language Specification明确规定：数组内的元素必须是“物化”的。

It is a compile-time **error** if the component type of the array being initialized is **not reifiable**.

对“物化”的第一条定义就是不能是泛型：

因为Java的范型会在编译后将类型信息抹掉，这样如果Java允许我们使用类似

**Map<Integer, String>[] mapArray = new Map<Integer, String>[20]; //error**

这样的语句的话，我们在随后的代码中可以把它转型为Object[]然后往里面放Map<Double, String>实例。这样做不但编译器不能发现类型错误，就连运行时的数组存储检查对它也无能为力，它能看到的是我们往里面放Map的对象，我们定义的<Integer, String>在这个时候已经被抹掉了，于是而对它而言，只要是Map，都是合法的。想想看，我们本来定义的是装Map<Integer, String>的数组，结果我们却可以往里面放任何Map，接下来如果有代码试图按原有的定义去取值，后果是什么不言自明。

**创建泛型数组的方法（不建议使用，最好使用集合）：**

**List<String>[] list = (ArrayList<String>[])new ArrayList[10];**

4.**List**

它是Collection的子**接口**。

***@see java.util.List;***

**概述:List**集合，是一种有序集，可重复集

List的特点是可以像数组一样，根据下标操作元素。

List提供一些独有的方法。

常用实现类：

ArrayList：内部由数组实现，查询快；

***@see*** ***java.util.ArrayList;***

LinkedList：内部由链表实现，增删快。

***@see*** ***java.util.LinkedList;***

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

1) ***E get(int index)***;获取指定下标对应的元素

2) ***E set(index, E e)***;将给定的元素设置到指定的位置上，返回值为原位置上的元素。只能在现有的元素上进行操作,所以set方法是**替换元素**的操作。

3) List提供的另一种方法：

***void add(int index,E e)***

向指定位置插入指定元素，原位置及后续元素顺序向后移动.

4) ***E remove(int index)***

删除指定位置的元素并返回该元素。

5) ***List subList(int start, int end)***

获取当前集合中给定范围内的部分子集，**含头不含尾**。

子集的任何操作都会影响原来的集合

快速删除集合中的某一块内容可以先把它当子集取出来调用clear方法

5.**Collections**:集合的工具类

***@see java.util.Collections***

**概述**:该类提供了若干静态方法，可以方便的操作集合。

其中**sort**方法是用来对List集合进行自然排序。

***List<Integer> list = new ArrayList<Integer>()***;

1)sort会对给定的集合进行自然排序，即从小到大的顺序

面试题：Collection与Collections的区别？

Collection是所有集合的接口，Collections是操作集合的工具类

***Collections.sort(List list)***;

2)Collections的***sort***方法若希望对集合进行排序，必须保证集合中的元素是可比较大小的。如果不可比较大小，就要求元素必须实现***Comparable***接口，并重写其中的比较大小的方法，才可以进行排序。

当集合存放的是自定义元素时，使用Collections的sort方法排序的注意事项:

当前元素若希望可以比较大小，需要实现Comparable接口。

Comparable支持泛型，而泛型的实际类型就是当前类。哪个类实现了Camparable接口，泛型就是哪个类。

***compareTo***方法的作用是使当前对象与给定的对象o进行比较大小。

返回值不关心具体值，而是关注取值范围。

返回值>0:当前对象比参数对象大;

返回值<0:当前对象比参数对象小;

返回值=0:两个对象相等;

虽然sort方法可以对集合中的元素进行自然排序，但是必须要求元素实现Comparable接口。这就出现了由于想使用排序功能而必须改变我们定义的类的内容，这种现象称为"侵入性"。

3)Collections的重载sort方法。

字符串String实现了Comparable接口，排序规则为按照首字母的编码比较大小。

Collections的重载sort方法要求我们传入两个参数：

1.等待排序的集合

2.比较器(***comparator***)

***Collections.sort(List list,Comparator com)***;

使用比较器要重写***compare***方法，int compare(E e1,E e2)

***@see java.util.Comparator;***

通常有两种情况会使用该方法排序集合：

1：元素已经实现了Comeparable接口，但是比较大小的规则不能满足我们对于排序的需求时；

2：元素没有实现Comparable接口，并且也不希望为了这里排序而强制修改元素，要求其实现接口时。

**Queue**

1.Queue

它是一种**接口**。

***@see java.util.Queue***

**概述:队列。**

保存一组元素，但是对于存取有要求，必须遵循**先进先出**原则。

***Queue<T> queue = new LinkedList<T>()***;

1) ***boolean offer(E e)***;向队列末尾追加一个元素

2) ***E poll()***;从队首取元素。注意，获取后元素就从队列里被移除了！出队操作。

3) ***E peek()***;获取队首元素.

但是与poll不同的是并不会将该元素从队列中删除,所以只能去取队首元素。

4)遍历队列,一次性操作，**操作完成队列为空**

可以使用for循环，while循环

for循环注意：i初值为队列长度，因为队列每取一个元素就会改变长度，初值为0无法完全遍历。

建议使用while循环。

***PriorityQueue***是一种优先级队列，元素可以按任意顺序插入，但总是按排序的顺序检索。使用优先级最典型的地方是任务调度，每个任务都有一个优先级，任务以随机的顺序添加到队列中，每启动一个新任务，就将优先级最高的任务从队列中删除(默认**1为优先级最高**的标记，即**删除数值最小的元素**)。

***PriorityQueue<E> pq = new PriorityQueue<E>();***

1. ***PriorityQueue(int inticalCapacity)***;

构造一个实现comparable对象的优先级队列;

1. ***PriorityQueue(int inticalCapacity, Comparable<? super E> c)***;

构造一个优先级队列，并用指定的比较器对元素进行比较;

2.Deque

***@see java.util.Deque***

Deque是Queue的**子接口**

双端队列的特点是，队列的两端都可以进出元素。

若我们只调用一侧的进出方法，就形成的栈结构。

**栈**:存储一组元素，存储必须遵循**先进后出，后进先出**的原则。

通常用来实现具有后退功能的地方。

***Deque<T> stack = new LinkedList<T>();***

1) ***void push(E e)/offerFirst(E e)***

将元素"压入"栈中，入栈操作。新入栈的元素会在栈顶（栈中的第一个元素）

2）***E pop()/E poolFirst()***

出栈操作，获取栈顶元素，获取后该元素就从栈中删除了。

也可以使用peek方法获取栈顶元素而不做出栈操作。

**Map**

***@see java.util.Map***

Map是一个**接口**。

Map看起来像是一个多行多列的表格

每条记录分别分为两部分：key，value

其中在一个Map中和**key是不允许重复**的（equals比较）

常用的实现类：

***@see java.util.HashMap***(散列算法)

***@see java.util.TreeMap***(二叉树算法)

HashMap默认长度是16，建议存储数据不超过3/4，可以在新建时传入参数，确定容量。

***Map<K,V> map = new HashMap<K,V>()***;

1) ***V put(Key key, Value value)***;

将给定的Key与value存入到Map中。

由于Map中key不允许重复，所以会出现两种情况：

1：存入的key在Map中还不存在，那么直接将这对key-value存入Map，返回值为null;

2：存入的key在Map中已经存在，那么会将已经存入的value替换掉，并将被替换的value值返回。

基本类型接收返回值会报空指针异常，要使用包装类。

存入顺序与内部顺序不一样。

***V putAll(Map<? extends K, ? enxtends V> enties)***

将给定映射表添加到现有映射表中。

2) ***V get(K k)***

根据给定的key去获取对应的value。

若当前Map中不含有给定的key，则返回值为Null

3）***V remove(K k)***

将给定的key对应的这一对内容从Map中删除，并将对应的value返回。

4) ***boolean containsKey(K k)***

查看当前Map是否包含给固定的key，有则返回true;

包含的判断是一句key的equals比较的结果

***boolean containsValue(V v)***

查看当前Map是否包含给固定的value，有则返回true;

5）遍历**Map**

有三种方式:遍历所有key;遍历每一组键值对；遍历所有的Value（相对不常用）

5.1）遍历所有key

***LinkedHashMap***

***@see java.util.HashMap***

内部使用一个LinkedList维护顺序，可以做到遍历的时候与put的顺序一致。若不需要顺序，通常不使用该类。

***Map<K,V> map = new LinkedHashMap<K,V>();***

遍历所有的key：

***Set<K> keySet()***

该方法会将所有的key存入一个Set集合后返回，所以遍历该集合就相当于遍历到所有的key了

5.2）遍历每一组键值对

***Set<Entry> entrySet()***

Entry是Map的内部类，每一个实例表示一组键值对。

其中包含两个属性，key，value。

将每一组键值对（Entry实例）存入一个Set集合后将其返回。

5.3） ***Collection<V> values()***

将当前Map中所有的value值存入一个集合后返回。

6). **HashMap hashCode() equals()**

重写equals与hashcode的注意事项

当我们使用自定义类型的实例作为HashMap中的key值时，这个类的equals与hashcode()结果直接影响者HashMap查询的效率。

要避免这一事件的产生：

hashcode值一样，但是equals比较不为true时，会严重影响HashMap的效率。

JAVA API文档中对于两个方法的重写有着要求：

1.当我们重写一个类的equals方法时就应当连同重写hashcode方法。

还应该满足：

当两个对象equals比较为true时，hashcode返回的值应当相等。

反过来hashcode值相等时，不强制euquals值为true，但最好也相同，否则影响HashMap性能。

2.hashcode方法返回的应该是一个稳定的值。

当确定equals方法比较结果的属性值没有发生改变的前提下，多次调用hashcde方法返回的值也应当相同。

HashMap默认长度是16，建议存储数据不超过3/4（75%），可以在新建时传入参数，确定容量。

6). **SortedMap与TreeMap**

6.1)***TreeMap(Comparator<? super K> c);***

构造一个树映射表，并指定比较器对key进行排序;

6.2)***TreeMap(Map<? extends K, ? enxtends V> enties)***;

构造一个树映射表，并将指定映射表所有条目添加到树表中;

6.3)***TreeMap(SortedMap<? extends K, ? enxtends V> enties)***;

构造一个树映射表，并将指定的有序映射表的所有条目添加到树表中,并使用给定映射表相同的比较器;

6.4)***SortedMap Comparator<? super K> comparator()***;

返回对key进行排序的比较器。如果key是用Comparable的compareTo方法进行排序的则返回null;

6.5)***SortedMap k firstKey()/lastKey();***

返回映射集最小和最大元素;

7). 弱散列映射表

**WeakHashMap**

如果有个值，它的key不再使用了，或者说是消亡了，那么不再有任何key能指向这个值，并且由于程序中没有任何位置出现这个key，垃圾回收器也无法删除它，需要有程序长期存活的映射表里删除这些无用的值，可以使用WeakHashMap完成这件事情。

链接散列集和链接映射表

**LinkedHashSet**与**LinkedHashMap**用来记住插入元素项的顺序。

**File and Stream for IO**

**1.File**

**写在前面：**

**创建File对象后，需要使用createNewFile创建文件/ mkdir创建目录，只创建File对象时不能生成文件或目录的**

它是一个**类**。

***@see java.io.File***

该类用于描述文件系统中的一个文件或者目录。

**File可以访问文件或目录的属性信息。**

**File可以操作文件或目录。**

**File可以查看一个目录有哪些子项**。

**File不能访问文件内容！**

1)描述目录时应当注意两点：

1.尽量不要使用绝对路径，因为不同操作系统有差异；

2.目录色层级分隔符不要直接写"/"或"\"，因为不同系统也有差异，所以使用File的常量：File.separator就可以了。

名词释意：separator n.分隔符

***File file = new File("."+File.separator+"demo.txt")***;//创建一个File对象

2)获取文件名

***String File.getName()***;

3)获取文件大小(文件占用硬盘空间的字节量)

***long File.length()***;

4)查看当前File表示的文件或目录是否在硬盘中已经存在

***boolean File.exists()***;

5)获取最后的修改时间 UTC 1970到现在的毫秒差

***long File.lastModified()***;

6)是否是隐藏文件

***boolean file.Hidden()***;

是否可读

***boolean file.canRead()***;

是否可写

***boolean file.canWrite()***;

是否可运行

***boolean file.canExecute()***;

7)使用File创建一个新文件

***boolean createNewFile()***;

当前项目根目录下创建文件:test.txt

若直接书写文件名，默认就是在当前目录下，所以".+File.separator是可以忽略不写的。

8)删除目录中的test.txt文件

***boolean delete()***;

9)创建新目录

***boolean mkdir()***;

10)在当前根目录下创建目录:a/b/c/d/e/

***boolean mkdirs()***方法在创建目录时会生成当前目录所在的多级前置目录

11)获取一个目录下的所有子项

获取当前项目根目录下的所有子项/

File dir = new File(".");

***File[] listFiles()***;

该方法会获取当前目录下所有子项。

由于一个目录下包含的无非还是文件或目录，那么谁都可以有一个File对象表示，所以获取一个目录里所有子项就等于获取了若干File对象，每一个表示其中一个子项。

12) **FileFilter 文件过滤器**

它是一个**接口。**

File有一个重载的listFiles方法，允许我们传入一个文件过滤器，并将满足过滤器要求的子项返回。不满足的则忽略。

获取当前目录下名字以"."开头的文件或目录。

FileFilter filter = new FileFilter(){

过滤器的过滤条件：

当给定的file对象满足要求，返回true即可。

***public boolean accept(File file) {***

***String name = file.getName();***

***return name.startsWith(".");***

***}***

获取给定过滤器条件的所有子项

}；

***File[] subs = file.listFiles(filter)***

13)**删除多级目录**

当删除一个含有子项的目录时，直接使用delete方法是删除不了的，需要将该目录中的所有子项删除以后才能成功删除该目录。

这里使用递归算法。

***递归***

***方法内部调用该方法。***

***注意:***

***1.不能100%调用本身，必须在一定条件下调用，以免造成死循环。***

***2.能用循环就用循环，尽量不使用递归，比较耗损系统资源。***

***3.不要深入思考递归，容易蒙圈，只要相信自己写的方法能够完成目的就行。***

public static void main(String[] args) {

删除当前目录中的a目录

File dir = new File("a");

deleteFile(dir);

}

删除给定的File表示的文件或目录

public static void deleteFile(File file){

if (file.isDirectory()){

File[] subs = file.listFiles();

for(File sub : subs){

deleteFile(sub);

}

}

file.delete();

}

2. **RandomAccessFile**

它是一个**类**。

***@see java.io.RandomAccessFile***

该类用于读写文件数据。

创建该类有两种常用模式：

1.只读，对文件仅进行读取操作"r",read

2.读写，对文件数据读写"rw",read,write

常用构造方法：

***RandomAccessFile(File file,String mode)***

***RandomAccessFIle(String path,String mode)***

1）对当项目根目录下的文件test.dat文件进行操作

***void write(int i)***

该方法向文件中写出1个字节，即8位的二进制。

写的是该int值的二进制形式中的低8位。

00000000 00000000 00000000 00000001

***raf.close()***;//方法结束关闭调用，节省资源。

***int read();***

该方法向文件中读出1个字节。

2）批量**写/读**数据

***void write(byte[] dat)***;

将给定的字节数组中的所有字节一次性写出。

重载方法:

***void write(byte[] dat, int start, int len)***

将给定的字节数组中字节从start开始连续len个字节写出。

**Tips:**

字符串String提供了一个方法：

***byte[] getBytes()***;根据默认的字符集将当前字符串转换为对应的字节。

String提供了一个重载getBytes()方法，允许根据给定的字符集将当前字符串转换为对应的字节。

***byte[] getBytes(String charsetName)***

字符集

**UTF-8** 万国码，中文3个字节，英文1个字节。1-4个字节构成不同国家语言的文字。

**GBK** 国标编码，中文2个字节，英文1个字节。

**ISO8859-1** 欧洲常用字符集，不支持中文。

***int read(byte[] data)***

一次性尝试读取给定的字节数组data的长度的字节量，并将这些字节存入该数组中。

返回值为实际读取到的字节量。若返回值为-1，则读取到了文件的末尾（EOF end of file）

***int read(byte[] data, int start, int len)***

一次性尝试将给定的字节数组data从start开始连续len个字节读出并存入该数组中。

**Tips:**

String提供了一个构造方法，允许我们将给定的字节数组中的字节转换为对应的字符串:

***String(byte[] data)***

按照当前系统默认字符集将字节数组中所有字节转换为字符串。

String重载的构造方法

***String(byte[], int start, int len)***

按当前字节数组中从start处开始的字节，连续取len个，然后将它们按当前系统默认的字符集转换为字符串.

***String(byte[], int start, int len, String charsetName)***

将字节数组中部分内容按照给定的字符集转换为对应的字符串

3) RandomAccessFile总是基于指针当前位置进行读或写操作的。

并且每当读或写当前字节后，指针会自动向后移动。

***long getFilePointer()***

获取当前RandomAccessFile的指针位置(指向文件的第几个字节)。

刚创建出来的RAF默认指向文件的第一个字节，下标从0开始。

只要进行一次写操作/读操作，指针就会移动位置

***void writeInt(int i)***

连续写出4个字节，将给定的int值完整的写出。

写出其他基本类型的方法也有。

read也一样

raf.write(max>>>24);//>>>位运算符。

读取上面写入的long值，我们需要先将指针移动到long值的第一个字节处，然后连续读取8个字节将其读取出来。

***void seek(long pos)***

该方法可以将指针移动到指定位置。

***long readLong()***

从指针当前位置处开始，连续读取8个字节并转换为long值返回

3.输入输出**流(Stream)**

1) ***FileInputStream(String fileName)***

***@see*** ***java.io.FileInputStream***

低级流，以字节为单位读取文件数据的流。

FileInputStream fis = new FileInputStream("fos.txt");

若该地址没有文件，则自动新建一个文件。其他同此。

2)

2.1) ***@see*** ***java.io.FileOutputStream***

文件的节点输出流。用于向文件中写出字节。是一个低级流。

***FileOutputStream(String fileName)***

默认创建的FOS是覆盖写操作。

该覆盖操作是在写内容前先将源文件中所有数据清除，然后重写写入新的内容。

流使用完后也要关闭，释放资源！

释放的资源是JVN底层资源（C语言控制，直接访问硬件），以腾出内存空间，防止内存泄漏。

***fos.close()***;

2.2) ***FileOutputStream(String fileName, boolean b)***

FOS有一个重载的构造方法，传入第二个参数是个boolean值。

若该值为true，则是追加写操作，即:通过该流写出的数据会在当前文件的末尾写入，而不是将原来的数据清除。

2.3)使用文件流完成复制操作

使用缓冲流加快读写效率完成复制操作

输入缓冲流:

***BufferedInputStream(FileOutputStream fis)***;

输出缓冲流:

***BufferedOutputStream(FileOutputStream fos)***;

BIS和BOS内部都有一个缓冲区(字节数组)。

当我们调用bis.read()方法尝试读取一个字节时，BIS会尝试让FIS一次性读取若干字节回来，并存入内部的缓冲区，然后将第一个字节返回。

这样当我们再次读取一个字节时，就直接将数组中的第二个字节返回了，而不是再读取。

所以还是一次读取了若干字节，减少读取次数最高的读取效率。

写出也是一个原理。当我们写出一个字节时，

BOS是将该字节放入内部维护的字节数组中，只有该数组存满了才会让FOS将这一组字节一次性写入文件。

关闭流的时候只用关闭最外层的高级流即可

2.4)缓冲流写出数据的注意事项

***void flush()***

该方法会强制将缓冲区中的数据一次性写出，不管缓冲区是否已经装满。

频繁调用flush方法会提高写出次数从而降低写出效率，但是会保证写出的及时性。

当关闭缓冲流的时候，它会首先调用flush将剩余在缓冲区的数据写出，再将它处理的流关闭后，才将自己关闭。

***bos.close()***;

4. **对象流**

1) ***@see*** ***java.io.ObjectOutputStream***

该高级流的作用是将给定的java中的对象转换为一组字节，然后通过其他从处理的流写出。

将一个对象转换为一组字节的过程为:对象序列化。

传入的对象需要实现一个接口***Serializable***，该接口不含任何内容。

将该对象转换为一组字节然后写入文件，这里需要两个流的组合来实现:

1.***ObjectOutputStream***

将一个对象转换为一组字节；

2.***FileOutputStream***

将一组字节写入文件

ObjectOutStream提供了一个独有的方法

***void writeObject(Object obj)***

该方法会将给定的对象obj转换为一组字节，然后在通过其处理的流写出

2) ***@see java.io.ObjectInputStream***

该类的作用是读取一组字节（必须是由OOS将对象序列化的一组字节），

将其还原为对象。

读取person.obj文件，将person对象还原从字节到对象的过程叫做对象的"反序列化"

***Object readObject();***

ObjectInputStream 提供了一个方法，该方法可以从其处理的流中将字节读取回来并转换为对象返回。

返回的时候是以顶级父类Object形式，返回后需要自己造型。

该方法需要我们处理一个异常:

**ClassNotFoundException**

该异常通常出现的原因是，读取的那一组字节并不能转换为对象。这组字节不是由OOS的writeObject方法序列化的字节。

Person p = (Person)ois.readObject();

ois.close();//使用完后要关闭流。

3) 对象流要点:

若当前类的对象需要被ObjectOutputStream进行序列化，当前类必须实现java.io.Serializable接口，否则在序列化的过程中会抛出异常。

该接口没有任何抽象方法需要重写，这种接口的目的是给子类打上一个标签，表示认可这件事。而不是用来约束子类应具有哪些行为。

除了当前类之外，该类的属性若也是自定义类型的，那么该类也必须是先***Serializable***接口才可以。

serializable n. 可串行化的。

当我们的类实现了***Serializable***接口后，就应当定义下面的常量。

**long serialVersionUID**

这个常量是版本号，最好自行维护，该版本号直接影响反序列化的结果。

当我们反序列化一个对象时，OIS首先会检查要反序列化的那一组字节表示的对象的版本号是否和当前类的版本号一致。

不一致会直接抛出异常，告知版本号不匹配，无法反序列化。

若版本号一致，则会使用兼容默认，即:

原来对象中有的属性，现在类还有的，则还原它的值；

原来对象中有的属性，现在没有了，则忽略；

原来对象中没有的属性，现在有了，则使用默认值。

若不自行维护版本号，那么OOS在进行序列化时，会根据当前类的结构生成一个版本号。这样做的问题在于，当我们的类结构发生改变后，版本号也会改变，因为不可控，所以不能启用兼容模式。

当一个属性被***transient***关键字修饰后，该属性在序列化的时候将被忽略。忽略不必要的属性是为了对象在序列化为字节时"瘦身"。

5.字符流

***Writer,Reader*** 字符输出、输入流的父类。

字符流特点:以字符为单位读写数据。

字符流**只能读写文本数据**。所有不要用字符流读取非文本数据。

字符流只是简化我们对字符串的读写操作，不用自行在字节与字符之间转换。

所以字符流都是高级流，底层本质还是要用和字节读写数据。

1) ***@see java.io.OutputStreamWriter***

字符输出流，按照给定的字符集，将字符串转换为一组字节后写出

***OutputStreamWriter(FileOutputStream, String charsetName)***;

***@see java.io.InputStreamReader***

字符输入流，按照指定的字符集读取字符

***InputStreamReader(FileInputStream, String charsetName)***;

为了方便读取字符流，可以用字符流输入。同样的，读取到文件末尾会返回-1。

2)

2.1) 缓冲字符输出流:***BufferedWriter***

***@see java.io.BufferedWriter***

由于内部维护了一个字符数组作为缓冲区，所以写出字符串的效率较高，而且还有一个特点就是可以以行为单位写出字符串。

***PrintWriter***:

高级流

具有**自动行刷新功能的缓冲字符输出流**，由于该流的缓冲功能是使用BufferedWriter实现的，并且其提供了更多功能，所以比较常用。

创建PrintWritter时，总是先创建BufferedWritter作为缓冲功能。

直接操作文件的构造方法

***PrintWriter(File file)***//参数:File对象

***PrintWriter(String pathName)***//参数:文件地址

直接操作文件的这两个构造方法都支持第二个参数，就是指定字符集。

2.2) ***PrintWriter***可以处理其他流

既可以处理字节流，也可以处理字符流，但是直接处理字节流的弊端在于不能指定字符集。

当创建PrintWriter并处理其他流时：

即：构造方法第一个参数为流，那么这些构造方法都支持第2个参数，是一个boolean值。当该值为true时，PrintWriter就具有自动行刷新功能。

***PrintWriter(OutputStreamWriter osw,boolean b)***

当第二个参数为true时，该PrintWriter就具有自动行刷新功能，即:

每当调用println()方法时，将字符串写出后自动flush()。

需要知道:自动行刷新会提高写出次数，降低写出效率，但是可以提高写出的及时性。

***void println(String str)***;

PrintWriter提供一个方法用来逐行打印输出。

3) 缓冲字符输入流***BufferedReader***

***@see java.io.BufferedReader***

需要处理一个字符流，并且可以以行为单位读字符串

***String readLine()***

BufferedReader提供了该方法可以以行为单位读取字符串。

该方法会顺序读取若干字符，直到遇到换行符位置，然后将换行符之前的所有字符转换为一个字符串返回。

该字符串中不包含换行符。

若返回值为null，表示读取到文件末尾。

**Exception**

***@see java.lang.Throwable***

***implements Serializable***

***@see java.lang.Error***

***@see java.lang.Exeption***

1. 异常捕获机制 ***try...catch***

***try***中包含的是可能出错的代码片段;

***catch***用来捕获try中出现的错误，并处理。

***try{***

程序执行到try中出错代码后会跳到catch中继续执行，try中错误后面的代码不会再执行了

***}catch(…){***

catch语句可以写多个，针对不同的异常，若有不同的处理手段，可以分别捕获。

但是应当有一个好的习惯，在最后一个捕获中来捕获Exception，这样至少可以保证程序不会因为一个未捕获的异常导致中断。

当catch中的异常存在父子继承关系，要保证父类异常在下面，子类异常在上面

***}catch(…){***

***......***

***}catch(Exception e){***

***……***

***}***

2. ***finally***块

***finally***块必须定义在异常捕获机制的最后。

可以定义在try之后，也可以定义在最后一个catch之后。

***finally***可以保证其内部的代码无论try中语句是否出现异常都会执行。

通常像释放资源这类代码就应当放在finally中，比如关闭流。

3. ***throw***

通常两种情况会使用throw抛出一个异常：

1.程序中遇到一个满足语法要求，但是不满足业务逻辑要求的情况时，可以自行创建一个异常实例并抛给调用者。

2.程序执行过程中确实出现了一个异常，但是不应的在当前方法中解决时，可以将其抛给当前调用者。

4. ***throws***

4.1) 当一个方法上使用throws声明抛出某些异常时，我们在调用该方法是要求必须处理这些异常。

处理手段有两种:

1.自行try-catch捕获并处理；

2.在当前方法上继续使用throws声明将这些异常抛出。

**实际工作中不要throws异常到main方法外**。

4.2)子类重写父类含有throws抛出异常的方法时的注意事项。

***重写时抛出的异常必须小于或者等于父类***。

重写时允许子类不抛出任何异常。

重写时也可以仅抛出父类方法抛出的部分异常。

重写时可以抛出父类型方法中抛出的异常的子类异常。

重写时**不允许**抛出额外异常。即:父类没有的，也不是父类抛出的异常的子类异常。 重写时**不允许**抛出父类抛出的异常的父类异常。

等于是可以的。

4.3) 自定义异常，通常用来描述某个业务逻辑级别的错误。

public class IllegalAgeException extends Exception{

...//由于继承了Serializable，需要自己维护版本号;

//由于继承了Exception，需要重写方法，用eclipse生成。

}

5. **异常常用API**

5.1）异常检测机制，分为可检测异常(必须捕捉)，非检测异常(不受编译器要求)。

常用的非检测异常：RuntimeException

IllegalArgumentException 方法传递参数异常

NullPointerException 空指针异常

ArrayIndexOutOfBoundsException 数组下标越界异常

ClassCastException 父类强转为子类异常/无实例子类强转异常

NumberFormatException 数据转换异常

5.2) ***Throwable***类

它有两个子类 Error:系统级别异常。Exception:程序级别异常。

通常说的都是Exception

5.3) ***void PrintStackTrace()***;

Throwable类定义个这个方法用于输出错误信息，常用。

5.4)***String getMessage()***;

Throwable类定义个这个方法用于获取异常信息。

5.5)***Throwable getCause()***;

Throwable类定义个这个方法用于检索异常信息，可以深入到底层异常。

**Thread**

***@see java.lang.Thread***

***implements Runnable***

1. **线程**(***Thread***) (异步)

这是一个**实现类。**

线程存在于进程之中，而进程为程序执行开辟的独立运行资源空间。

每个进程必有一个首要线程。

并发运行，可以“同时”完成多个任务。

线程有两种创建模式。

1.1) 模式1：

定义一个类并继承Thread，然后重写run方法，在内部编写该程序要执行的任务.

***Thread t1 = new MyThread()***;

注意:启动线程不要直接调用run方法，而是调用start方法，该方法会将当前线程注册到线程调度中，使其可以并发运行。

start方法调用完毕后意味着当前线程进入了runnable状态，一旦被线程调度分配时间片就开始运行。

线程对于线程调度的工作不可控：

CPU时间片分配个哪个线程不可控；

时间片长短不可控。

不过线程调度会尽可能的均匀的将时间片分配给每个线程。

***void start()***;

第一种创建线程的方式存在两个弊端:

1.由于java是单继承的，所以我们继承了Thread就无法再继承其他类，这对于项目开发而言是不能容忍的。

2.重写run方法会导致线程与线程要执行的任务有一个耦合关系，这导致当前线程只会完成该任务，不利于线程的重用。

第一种创建线程的方式也存在一个优点:

定义简便。

当我们需要临时使用一个线程完成某个任务时通常会使用**匿名内部类**，这时候用第一种方式创建比较理想。

1.2) 模式2:

***Runnable***接口

包含一个***void run()***需要重写。

**线程**的第二种创建模式:

1.单独定义线程要执行的任务。

定义一个类，实现Runnable接口，并重写run方法。

2.创建线程并将任务指派进去再启动。

优点:

1.单独定义任务，在用的时候才指派给线程，所以和线程没有必然的耦合关系(相互依赖关系)，利于线程的重用；

2.由于任务类是实现Runnable接口，java接口是多实现的，所以无论是继承它类还是实现其他接口都不再受影响。

1.3）使用**匿名内部类**的形式创建线程

//模式1

Thread t1 = new Thread(){

public void run(){

....

}

};

//模式2

Runnable runn = new Runnable(){

public void run(){

....

}

};

Thread t2 = new Thread(runn);

t1.start();

t2.start();

1.4) Thread提供了一个静态方法

***static Thread currentThread()***，

该方法可以获取运行这个方法的线程

实际上main方法也是靠一个线程运行的，只是该线程不是由我们直接创建的。

Thread提供了一个构造方法，可以传入参数命名线程

***Thread(String str)***;

1.5)获取线程状态的相关方法

获取线程ID(long)

***long t.getId()***;

获取线程名字(String)

***String getName()***;

获取线程优先级(int)

***int getPriority()***;

获取线程是否活动(boolean)

***boolean isAlive()***;

获取线程是否后台运行(boolean)

***boolean isDaemon()***;

获取线程是否被中断(boolean)

***boolean isInterrupted***();

1.6) 线程优先级(***Thread Priority***)

线程优先级有10个等级:1-10，其中1最低，10最高，默认是5.

有3个常量可以对应最低，最高，默认优先级。

**Thread.MIN\_PRIORITY** ===>1

**Thread.MAX\_PRIORITY** ===>10

**Thread.NORM\_PRIORITY** ===>5

设置线程优先级的方法

***void setPriority(int Priority)***

可以改善线程获得时间片分配的几率

**2.常用API**

2.1）Thread提供了一个静态方法sleep，该方法可以使运行该方法的线程阻塞指定毫秒。

当阻塞时间超时后，该线程会自动进入Runnable状态，等待再次分配时间片然后运行。

***static void sleep(long time)***;

传递参数是一个时间毫秒值。

注意:sleep做不到严格意义上的间隔。

它可以保证1s的阻塞，但是解除阻塞后线程回到runnable状态，等待分配时间片才能进入running状态。这个等待的时间就是误差。

2.2）守护线程（Daemon Thread）

当一个进程中的所有前台线程都结束时，进程就要结束，无论后台线程是否还处于运行状态，都要强制中断。

Thread提供了一个方法，参数是一个boolean值，为true时表示守护进程。

***setDaemon(boolean b)***;

2.3) join方法

join方法会阻塞运行该方法的线程，允许该线程在另一个线程上等待，直到那个线程的工作结束，当前线程才会解除阻塞状态继续运行。

***void join()***;

该方法通常为了实现两个线程间同步(有先后顺序工作)的情况。

2.4) ***sychronized***关键字

**多线程并发的安全问题**。

当多个线程同时操作一段数据时，由于线程切换的不缺定性，可能导致逻辑混乱，严重的时候可能会导致系统崩溃。

为了解决多线程的安全问题，我们需要将**异步**变成**同步**。

意思是从各干各的变成有先后顺序排队干。

2.4.1)

***synchronized***关键字

有两种方式:

1.修饰方法，当该关键字修饰方法后，该方法变为同步方法，多个线程不能同时调用当前方法。

2.***synchronized***块

2.4.2)

方法上使用***synchronized***修饰有一个弊端，就是该方法所有代码都需要同步，除非真的需要这样，否则这个方法的执行变得比较低下。

**synchronized块**允许我们自由选择需要同步的代码片段，这样在保证线程并发执行安全的前提下缩小了同步范围，提高了代码的执行效率。

**同步锁**

当多个线程看到***synchronized***修饰的同一个方法或一个代码片段时，那么这是同步效果。指的是多个线程不能同时访问该方法或者该段代码，这个时候***synchronized***就是"**同步锁**"。

***synchronized***无论是直接修饰方法还是使用块的形式，都需要有一个"**锁对象**"。这个对象可以是任意对象，但是想要保证多个线程间具有同步效力，那么必须保证这些线程看到的***对象是同一个***。

直接修饰方法时，锁对象就是该方法所属的对象，所以使用块的形式也可以使用该对象，一般可以用***this***表示。()里面指向***synchronized***锁对象。

**互斥锁**

***synchronized***也可以达到让不同代码间存在***互斥效果***。

即:当一个线程执行一段代码时，另一个线程除了当前代码不能执行外，另一段代码也不能执行。

互斥的关键:锁对象一致。

静态方法上添加***synchronized***，那么锁的是用于描述当前类的一个Class的实例。

Class的每一个实例都可以描述java中的一个类。而且每个类JVM中只有一个Class实例表示。所以静态方法跟对象无关，一定具有同步效力。

若多个静态方法都被***synchronized***修饰后，他们之间一定具有互斥效果。

***synchronized* 锁**

不同线程看到***同一个对象***:同一段代码就是**同步锁**，不同代码就是**互斥锁**。

锁效果要生效，必须在所有要锁的代码段之前加上***synchronized***修饰词。一方加锁一方不锁是没有效果的。

2.5） ***Object***中定义了两个方法***wait,notify***，配合使用可以达到协调线程同步的目的。

***void wait();***

***void notify()***;

当一个线程调用了一个对象的wait方法后，该线程进入阻塞状态，直到这个对象的notify方法被调用才可以解除阻塞。对于协调线程同步而言，这种方式比join的即时性强，因为join必须被动的等待其他线程完成工作。

一个方法中的局部内部类若想引用这个方法的其他局部变量，那么这个变量必须是final的。java1.8新特性可以不加final

***synchronized(obj){***// 必须放在锁内，谁调用wait方法就锁定谁

***obj.wait();***

***}***

nofity方法也一样。

2.6)线程安全

StringBuffer 是同步的，是线程安全的。

StringBuilder 不是同步的，是线程不安全的。

ArrayList,LinkedList 都不是线程安全的。

将集合或Map转换成线程安全的，使用Collections工具类的以下静态方法:

***Collections.synchronizedCollection(Collection col)*** ,返回col

***Collections.synchronizedList(List list)***,返回list

***Collections.synchronizedSet(Set set)***,返回set

***Collections.synchronizedMap(Map map)***,返回map

3.线程池

***ExecutorService extends*** ***Executor***

它们都是接口

***@see java.util.concurrent.ExecutorService***

***@see java.util.concurrent.Executors***

使用线程池来维护线程:

1.线程池可以控制线程数量;

2.线程池可以重用线程。

当频繁创建销毁线程时，会给系统带来过度消耗资源、以及过度切换线程的危险，这种情况下就应当使用线程池来维护线程。

ExcuteorService实现：

***Executors.newCachedThreadPool():***

创建一个根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。

***Executors.newFixedThreadPool(nThreads)：***常用，参数传入数字

创建一个可重用固定线程集合的线程池，以共享无界队列的方式来运行这些线程。

***Executors.newScheduledThreadPool(corePoolSize)：***

创建一个线程池，它可以安排在给定延迟后运行命令或者定期的执行。

***Executors.newSingleThreadExecutor():***

创建一个单线程的线程池，以无界队列方式来运行该线程。

线程池的excute方法:

***void execute(Runnable runn)***

该方法会将我们给定的线程体用其维护的线程来运行，前提是线程池中有空闲线程。

若没有，则会将该任务放入队列等待，直到有空线程为止。

4.双缓冲队列

BlockingQueue

***@see java.util.concurrent.BlockingQueue***

***@see java.util.concurrent.LinkedBlockingQueue***

以双缓冲队列实现线程存取，实现同存同取存取的同步效果，提高性能

一般实现为：

BlockingQueue ***LinkedBlockingQueue；***不定大小的双缓冲队列，默认大小为int型的最大值，可带int型参数限制大小。所含对象是以先入先出顺序排序的。

5.Fork/Join框架

**TCP & UDP**

Socket

***@see java.net.Socket***

ServerSocket

***@see java.net.ServerSocket***

**概述：Socket**通常称为**套接字**，封装了TCP协议的socket，通过它连接服务端的ServerSocket并创建输入输出流与服务端通讯。

**ServerSocket**是运行在服务端的，其作用是向系统申请服务端端口，以便监听该端口，等待客户端的连接。一旦一个客户端连接，就会创建一个Socket与该客户端进行通信。

1.初始化Socket:

需要传入两个参数:

1.服务端的IP地址;(host)

2.服务端的端口号;(port)

首先要清楚:

IP地址的作用是让我们通过网络可以找到服务器，而端口可以让我们找到运行在服务器上的服务端应用程序。

创建Socket实例的过程就是与服务器连接的过程，若可以成功与服务器连接上，则会创建Socket实例，否则构造方法会抛出异常。

***Socket socket = new Socket(String host,int port)***;

2. 初始化ServerSocket:

初始化ServerSocket的同时需要指定服务端口，该端口不能与当前系统使用的TCP协议的其他程序申请的端口冲突，否则会抛出端口被占用的异常。

***ServerSocket server = new ServerSocket(int port)***;

2:相关***API***:

1)Socket提供了该方法，或来获取输出流向服务端发送数据

***OutputStream getOutputStream()***;

Socket提供了该方法，或来获取输入流向服务端接收数据

***InputStream getInputStream()***;

2)通过socket可以得知远端计算机信息

***InetAddress socket.getInetAddress()***;

获取远程计算机IP

***int inetAddress.getHostAddress()***;

3).***Socket accept()***

ServerSocket提供的该方法用来监听打开的服务端口(port)。

该方法是一个阻塞方法，直到一个客户端尝试连接才会解除阻塞，并创建一个Socket与刚连接的客户端进行通讯。

accept方法每次调用都会等待一个客户端连接，所以若希望服务端能接收若干客户端的连接，就需要多次调用该方法，来分别获取对应这些客户端的Socket与他们通讯。

**TIPS:**

无论是linux客户端，还是windows的客户端，当与服务端断开连接后，

都应该将与该客户端交互的Socket关闭，来释放底层资源。

**TCP通信(通用)**：

客户端：

1.创建客户端socket；

2.连接；

3.创建输出输入流发送读取数据；

服务端：

1.创建服务端ServerSocket;

2.阻塞方法Socket accept();

3.等待客户端连接

4.创建输入输出流读取发送数据；

**UDP通讯**：

标准流程:

客户端：

1.创建Socket

***DatagramSocket***

***@see java.net.DatagramSocket***

***DatagramSocket socket = new DatagramSocket()***;

2.准备数据

3.准备地址(目标地址)

4.将数据和地址设置到包中

***DatagramPacket***

***@see java.net.DatagramPacket***

***DatagramPacket pack = new DatagramPacket***

(***data***, // 发送数据对应的数组 byte[] data = str.getBytes(“UTF-8”);

***data.length***, // 数组中多少字节被发送

***address***, // 远端计算机地址

InetAddress address =InetAddress.getByName("localhost")

***port***); // 远端计算机端口

5.通过Socket发送包

***socket.send(pack)***;

服务端：

1 创建Socket

2 准备一个接收包

3 通过Socket接受数据并存如接收包中

***socket.receive(pack)***; 这是一个阻塞方法

这个过程中包里就有变化了:a.包中就有数据了b.包记录了数据从哪来的

4 从包中取出数据

***String message = new String(pack.getData(), 0, pack.getLength(),"uft-8")***;

**Reflect**

Reflect

***@see java.lang.reflect***

Class

***@see java.lang.Class***

Reflect library:

反射库，提供了一个非常丰富且精心设计的工具集，能够编写动态操纵java代码的程序。

反射机制：能够分析类能力的程序被称为反射(reflective)

1.Class类

获取class对象的3种方式:

* 1. ***Class clazz = T.getClass();***
  2. ***Class clazz = Class.forName(String className);***

该方法会抛出异常(ClassNotFound)，要求className必须是类名或者接口名

* 1. ***Class clazz = T.class;***

注意:getName方法用于数组类型会返回一个很奇怪的名字：

eg：int[].class.getName() 🡪 “[I”

2.相关api：

(java.lang.Class)

static Class forName(String className)

返回描述类名为className的Class对象

(Class) Object newInstance()

返回这个类的一个新实例(适用于没有重写构造方法的类)

(java.lang.reflect.Constructor)

Object newInstance(Object[] args)

使用这个类的构造器构造所属类的新实例，必须是Constructor对象使用;

(Class) Field[] getFields()

(Class) Filed[] getDeclaredFields()

getFields方法将返回一个包含Field对象的数组，这些对象记录了这个类或者其超类的公有域(public修饰的属性)。getDeclaredFields方法则是记录这个类的全部域(属性)，包括paivate等修饰的。

(Class)Method[] getMethods()

(Class)Method[] getDeclareMethods()

getMethods方法将返回所有的公有(public)方法,包括从超类继承而来的公有方法；getDeclaredMethod返回这个类或接口的全部方法，但不包括由超类继承的方法。

(Class)Constructor[] getConstructors()

(Class)Constructor[] getDeclaredConstructors()

getConstructors方法返回包括Constructor对象的数组，其中包含了Class对象所描述类的所有公有(public)构造器；getDeclaredConstructors方法返回所有构造器。

(java.lang.reflect.Constructor)

Class getDeclaringClass()

返回一个用于描述类中定义的构造器、方法或者域(属性)的Class对象。

(Constructor) Class[] getExceptionTypes()(在Constructor和Method类中)

返回一个用于描述方法抛出的异常类型的Class对象数组。

(Constructor) int getModifiers()

返回一个用于描述构造器、方法或域(属性)的修饰符的整型数值。使用Modifier类的toString方法可以根据int值打印修饰符。另外还有isPublic..isPrivate等判断方法。

(Constructor) String getName()

返回一个用于描述构造器、方法或域(属性)的字符串。

(Constructor) Class[] getParameterTypes()(在Constructor和Method类中)

返回一个用于描述参数类型的Class对象数组

(Constructor) Class getReturnType() (在method类中)

返回一个用于描述返回类型的Class对象。

Annotation

annotation

***@see javase 1.5***

**注解**是Java 5的一个新特性。注解是插入你代码中的一种注释或者说是一种元数据（meta data）。这些注解信息可以在编译期使用预编译工具进行处理（pre-compiler tools），也可以在运行期使用Java**反射**机制进行处理。

概述

eg:

@MyAnnotation**(**name**=**"sonmeName"**,** value**=**"Hello World"**)**

public class MyClass**{**

**}**

@Retention**(**RetentionPolicy**.**RUNTIME**)**

@Target**(**ElementType**.**TYPE**)**

public @interface MyAnnotation **{**

public String name**();**

public String value**();**

**}**

在interface前面的@符号表名这是一个注解  
**@Retention(RetentionPolicy.RUNTIME)**表示这个注解可以在运行期通过反射访问。如果你没有在注解定义的时候使用这个指示那么这个注解的信息不会保留到运行期，这样反射就无法获取它的信息。  
**@Target(ElementType.TYPE)** 表示这个注解只能用在类型上面（比如类跟接口）。你同样可以把Type改为Field或者Method，或者你可以不用这个指示，这样的话你的注解在类，方法和变量上就都可以使用了。