\*:单值类型集合统一的三个方法:

addAll():将一个集合当中所有的元素添加到调用方法的集合当中，注意，它不是工具类的addAll()

retainAll():求两个集合都有的元素(交集)

removeAll():从一个集合当中删除另一个集合的所有元素

HashSet 它是Set接口的实现类 Set接口无序 唯一

1:验证HashSet无序且唯一的特点，并且强调它的基本用法

\*: Set集合不再提供get(int)，也就是无法直接得到第几个元素

2:HashSet所谓的唯一并不是真正的唯一，而是"唯一"，即便是内存当中完全不同的两个对象，也有可能被视作同一个对象

3:HashSet的唯一取决于程序员如何定义hashCode()和equals()两个方法，通过覆盖这两个方法，我们可以让不同的对象视作相同，同样可以让相同的对象视作不同~

4:HashSet添加元素的完整流程其实不止两个步骤 而是三个步骤

这三个步骤分别是: hashCode() == equals()

1st && (2nd || 3rd)

参见一张图!

5:HashSet添加元素的时候 如果认定了新元素重复，则新来的直接舍弃，不会替换原有的内容，HashSet先入为主，先到先得，后来的重复元素直接舍弃!

6:HashSet删除元素只提供了remove(Object)，而且这个方法同样也尊重hashCode == equals的三步比较规则~

7:使用迭代器遍历集合的过程当中，不能直接对集合进行添加删除操作，如果需求决定一定要去删除，使用迭代器自身的car.remove();

8:使用HashSet千万不要在添加元素之后直接修改其参与生成哈希码的属性

如果一定要去修改 三个步骤完成

a> 删除元素 b> 修改属性 c>重新添加回集合

[就用新的属性重新锁定位置了!]

9:HashSet构造方法的两个参数:

int 分组组数 默认16

float 加载因子 默认0.75F

\*: 阈值 : 达到扩容条件的最小临界值(元素总量哦)

阈值 = (int)(分组组数\*加载因子);

构造方法第一个参数 int类型 分组组数 默认情况16

这个参数可以指定 但是最终结果必须是2的n次方数

构造方法第二个参数 float类型 加载因子 默认0.75F

阈值 : 达到某个条件的最小临界值

它指的是哈希表扩容的最小元素总数

阈值 = 分组组数\*加载因子 = 16\*0.75F = 12

每次扩容的时候 元素个数\*2

但是请注意，分16组的时候在第一小组的元素，在分32的时候未必还应该在第一小组，所以一旦扩容，哈希表当中已经存在的所有元素需要重新散列，来确定自己应该在的位置!

我们作为程序员应该尽量避免扩容带来的重新散列影响效率

所以我们应该让 分组组数\*加载因子 > 元素总量!

如果分组组数变大 加载因子不变 => 消耗更多空间 但是能够提高效率

如果分组组数不变 加载因子变大 => 效率变低 但是更加节省空间~

\*:HashSet如何调整性能和空间的取舍?

============================================================================================

TreeSet 是 SortedSet 的实现类 它要求元素有序且唯一

\*: 只要是Set不论是什么Set都要求唯一!

1:验证其特点[有序 唯一]的同时 强调基本用法

\*: 虽然是有序的Set集合了 但是依然没有提供get(int)

\*: 提供了四个专属的方法: first() last() pollFirst() pollLast()

2:TreeSet所谓的唯一也不是真正的唯一 也是"唯一"

即便是内存当中完全不同的两个对象 也有可能被视作相同~

3:想要放进TreeSet的类型 必须要实现Comparable接口 并且覆盖compareTo()

这个方法返回的int数有三种情况

返回正数: 新元素认定更大 放在右子树[后面]

返回零: 新元素认定重复 直接舍弃

返回负数: 新元素认定更小 放在左子树[前面]

4:如何借助多个属性综合形成排序规则

优先尊重哪个属性就先描述假如哪个属性不同~

5:TreeSet的add()和remove()都尊重compareTo()返回的结果

我们应该尽量保证compareTo()能够返回0

但是万恶的需求经常让compareTo()没法返回0

此时必须借助迭代器删除元素 否则set.remove()找不到要删除的元素

6:使用迭代器遍历集合的过程当中 不允许对集合整体进行添加或者删除操作

否则会触发ConcurrentModificationException

如果一定要去删除 请使用迭代器提供的car.remove();

7:使用TreeSet千万不要在添加元素之后直接修改参与排序的属性

否则会破坏整个树的结构 如果一定要去修改

a> 删除 b> 修改 c> 重新添加[重新排序了]

8:一朝为根不代表永世为根

自平衡的二叉树 是可能进行旋转修复操作的...

根节点是会变的~

9:TreeSet构造方法也可以传参 参数是比较器对象

Comparator 抽象方法: public int compare(o1,o2)

\*: 在无法直接修改当前类型的比较规则的情况下

我们就需要脱离开这个类的类体还能制定这个类比较规则的方式

----- 比较器

Comparable&Comparator的区别？

首先它们都是涉及到比较的两个接口

首先它们的包结构不同

java.lang.Comparable [语言基础包 lang包]

java.util.Comparator [工具包 util包]

其次它们的抽象方法不同

Comparable => compareTo(1)

Comparator => compare(1,2)

最重要的它们的语言语境完全不同

Comparable adj.形容词 有能力比较的

Comparator n. 名词 用来比较的工具

如果一个类实现Comparable接口 代表这个类的对象

自身就可以比较排序 所以能够直接放入TreeSet当中

如果一个类实现Comparator接口 代表这个类的对象

是用来比较另一个类型的对象的工具而已

应当在创建TreeSet的时候构造方法传参

class Student implements Comparable<Student>{

....;

}

代表Student对象是有能力比较的 所以Student对象可以直接放进TreeSet

那么添加的过程中会使用它的compareTo()方法完成排序

class Student implements Comparator<Teacher>{

....;

}

代表Student对象是用来比较Teacher对象的工具而已

那么集合当中是用来存放Teacher对象的!

这个Student只是为了给Teacher完成排序

它应当在构造方法当中传给TreeSet

===========================================================================================

\*: 请注意Comparator并不是TreeSet的专利哦~

也就是说 即便不使用TreeSet 也有可能会使用比较器

键值对集合 Map

Collection Map

List Set SortedMap

SortedSet

所有Map通用的操作~

如何添加元素:

map.put(k,v);

一次添加另一个集合

map.putAll(另一个Map);

\*:注意没有Collections.putAll();

如何得到元素个数

map.size();

如何通过指定的主键得到对应的值:

V value = map.get(k);

如何判断是否包含指定的键:

map.containsKey(k);

如何判断是否包含指定的值:

map.containsValue(v);

如何删除一个键值对

map.remove(k);

分解Map的三个重要方法:

keySet() : 得到所有的主键对象组成的Set集合

values() : 得到所有的值对象组成的Collection集合

entrySet() : 得到所有键值对记录组成的Set集合

注意键值对记录提供三个方法:

getKey() getValue() setValue()

#: Map集合添加元素的时候如果认定重复的主键

则新来的主键对象直接舍弃 但是新来的值对象替换原有值对象

#: 无论keySet() values() entrySet() 都不会创建新的集合

只是将原本的Map换个视角展现给我们而已

所以如果我们对得到的单值集合进行了删除 原本Map里也没啦

===========================================================================================

Java当中的内部类

为什么要使用内部类:

内部类是共享数据最最简单的方式之一~

Java当中内部类的分类:

内部类的名字: 能够共享外部类的哪些内容?

1.成员内部类 外部类的所有[静态+非静态]成员[属性+方法]

如何理解成员内部类与其所在的外部类的关系:

蛔虫和牛的关系~

如何创建对象: Outer.Inner in = new Outer().new Inner();

2.静态内部类 外部类的静态成员[属性+方法]

如何理解静态内部类与其所在的外部类的关系:

房客和房东的关系~ 寄居蟹和蛤蜊壳~

如何创建对象: Outer.Inner in = new Outer.Inner();

3.局部内部类

如果所在的是非静态方法就能访问外部类所有成员

如果所在的是一个静态方法就只能访问外部类静态成员

还有final修饰的局部变量 [8.0之前 8.0开始可以不加final但也不能修改]

如何理解局部内部类和所在的外部类的关系:

学生和老师的关系~

如何创建对象: Inner in = new Inner();

\*:注意位置限定!

必须在类定义之后 所在方法结束之前

4.匿名内部类

想要学好匿名内部类 首先需要理解如下的两个场景:

a> 如果生个孩子就是为了拿来卖钱的 那么就没有必要起名字~

b> 有些情况 我们的名字根本不重要 父母的名字或者叔叔阿姨的名字才好使

new 父类/接口(){

完成方法覆盖;

}

优势2点:

第一 省掉了一个没有意义的类名

第二 能够让开发的流程自上而下 一气哈成

===========================================================================================

异常 => Exception => 例外

程序运行过程当中出现的例外情况

Java当中所有问题的分支(体系结构)

Throwable[可以抛出的]

Error Exception

[错误] [异常]

RuntimeException

[运行时异常]

#: Error和Exception的区别?

Error通常是指由于底层硬件环境或者系统原因导致的

程序员通过代码无法解决的问题

Exception仅仅是指运行过程当中出现了例外情况而已

#: 运行时异常和非运行时异常的区别?

运行时异常 在编译的时候不要求程序员给出处理方案

只是运行的时候直接将问题体现出来

非运行时异常 编译的时候如果不给处理方案

编译直接报错 无法通过编译

#: 我们见过的非运行时异常:

IOException

AWTException

CloneNotSupportedException

#: 我们见过的运行时异常:

ArithmeticException：

在数学当中0不能做除数，在Java语言当中0也不能做除数，否则触发算术异常ArithmeticException只有int类型0作为除数才会触发异常

NegativeArraySizeException：

当我们初始化一个数组的时候，需要在[]当中填写需要的元素个数，但是语法检查只会检查当中是否是个int，但是当我们提供一个负数的时候，就会触发负数数组大小异常

ArrayIndexOutOfBoundsException：

其实数组当中有三个元素的时候，合法的下标只有: 0 1 2，如果我们提供负数或者超出范围的下标值，就会触发数组索引值超出边界异常

NullPointerException：

Java当中的引用类型的变量(对象)，默认值就是null，拿着null的对象调用任何方法或者访问任何属性，都会触发空指针异常

StringIndexOutOfBoundsException：

String类提供了众多的方法，其中substring()和charAt()都是需要提供下标索引值，但是如果提供的索引值根本就不在合法的范围内，则会触发字符串索引值超出边界异常

NumberFormatException：

Integer.parseInt()能够将字符串转换为对应的int数，Double.parseDouble()能够将字符串转换成对应的double数,但是如果提供的字符串当中有不合法的内容,则会出现数字格式异常

ClassCastException：

其实类造型异常主要是因为程序员的强制类型转换，当我们将毫无关系的两个类型进行强转的时候就会出现ClassCastException 类造型异常

IllegalArgumentException：

ArrayList以及很多很多核心类库当中提供的类型，都允许传参，但是参数如果根本就不是，一个可以作为数组大小的int值，则ArrayList会直接抛出非法参数异常

\*:编译器是不会报错的 因为-55也是int

IndexOutOfBoundsException：

List集合提供get(int)，允许传入下标索引值得到一个对象，但是必须保证下标在合法范围内啊!否则将会出现IndexOutOfBoundsException

IllegalStateException：

当我们使用迭代器的car.remove()方法的时候，必须保证迭代器的光标正指向一个元素，否则删除的操作会触发非法(迭代器光标)状态异常

ConcurrentModificationException：

并发修改异常，多出现在集合边遍历边删除元素时

#: 为什么要处理异常?

1> 如果是非运行时异常 我们不做处理就连编译都无法通过..

2> 程序运行过程当中一旦出现异常 就直接中断执行了

#: 如何处理异常:

a> 抛还上级 throws

\*: 它根本没有从实质上解决问题

就是将解决问题的时机进一步的延后了

\*: 它足以解决1 但不足以解决2

b> 自行处理 try catch finally

try{

可能出现异常的语句;

通常只有一句 除非需求需要跳过后面的操作;

}catch(要捕获的异常的类型 异常代号){

1: 隐瞒不报 啥都不写

2: 简要的审

System.out.println(异常代号.getMessage());

3: 详细的审

异常代号.printStackTrace();

}

\*: 一个try后面可以跟上多个catch

但是多个catch不能有包含的关系 必须前小后大或者并列

\*: JDK7.0开始 如果多种异常采用相同的方式处理

一根棍棍!

catch(NullPointerException | ArithmeticException e){

...;

}

#: throw和throws的区别

throws 用在方法签名的最后 用来表达本方法当中出现指定种类的异常

本方法当中不做处理 抛还给上级调用者处理

throw 用在方法体当中 在没有异常出现的情况下

主动制造异常出现的场景

#: finally当中的语句一定会执行吗? 如何不让它执行呢?

至少有两种情况下finally语句是不会被执行的：

（1）try语句没有被执行到，如在try语句之前return就返回了，这样finally语句就不会执行。这也说明了finally语句被执行的必要而非充分条件是：相应的try语句一定被执行到。

（2）在try块|catch块中有System.exit(0);这样的语句。System.exit(0)是终止Java虚拟机JVM的，连JVM都停止了，所有都结束了，当然finally语句也不会被执行到。

===========================================================================================

1st.

一个必须要使用静态初始化块的场景!

类体当中的静态变量，如果是调用有异常声明的方法来完成赋值的，那么类体当中我们没法直接try catch，也不可能在类上加throws，必须使用静态初始化块，因为初始化块当中可以写语句，自然可以try catch

2nd.

父类当中的方法如果没有抛任何异常子类的方法去覆盖这个方法的时候可以抛异常吗?

可以抛无数种运行时异常，但是这没有任何意义，其实Java当中每个方法都默认抛出所有RuntimeException~

3rd.

连环try

为了保证连续的多行语句无论是否出现异常，之后语句都能正常执行，我们必须学会，在finally当中嵌套使用try catch，这样会让代码格式非常混乱，但是却能保证一定执行所有语句

\*: IO流 JDBC 都需要使用这种方式~ (如果不使用TWR)

4th.

如果某个变量是使用有异常声明的方法完成赋值的，为了保证在try之后变量还能正常使用，必须将这个变量的定义拿到try之前，并且给它默认值赋值(所有引用类型默认值null)

\*: IO流 JDBC 都需要使用这种方式~

5th.

学会使用try catch代替标准的分支判断

有些情况下 使用异常处理机制来完成分支 更加简便~

===========================================================================================

线程

程序:保存在物理介质中的代码片段

进程:一旦程序运行起来，就变成操作系统当中的一个进程，而windows是多任务多用户的操作系统，所以同时能运行多个进程~

时间片! 宏观并行 而 微观串行

线程: 一个程序当中一条独立的执行线索称作一个线程，一个程序也可以兵分多路

线程的五大状态 = 线程的生命周期

新生 就绪 运行 消亡

Born Runnable Running Dead

阻塞

Blocking

如何创建线程:

1.extends Thread

@Override

public void run(){

部署线程作战计划;

}

2.implements Runnable

@Override

public void run(){

部署线程作战计划;

}

3.金针菇~(see you tomorrow)

#: 请使用两种不同的方式创建两个线程 并且启动它们

线程1 打印1-26 extends Thread

线程2 打印a-z implements Runnable

如何控制线程!~

0. setPriority(int) : 设置线程优先级别 可选范围1-10 默认5

\*: 优先级越高代表抢到时间片的概率越高

\*: 我们不推荐使用...什么都保证不了

1. static sleep(long) : 让当前线程休眠指定的毫秒数

2. static yield() : 让当前线程放弃已经持有的时间片 直接返回就绪

\*: 中文版的文档当中这个方法的描述是错的

3. join() : 让当前线程邀请另一个线程优先执行

在被邀请的线程执行结束之前

邀请别人的线程一直处于阻塞不再执行

\*: 请注意线程章节所有的静态方法 不要关注谁调用方法

谁调用方法都等价于类名.方法

我们要关注调用的语句出现在谁的线程体

别看谁点的 看写在谁里面

\*: 线程章节所有涉及到主动进入阻塞状态的方法

都必须进行异常处理

线程类其他常用的方法:

setName() + getName() : 用于设置和得到线程的名字~

\*: 在Thread类当中有一个String name属性

可以给线程起名字 但是属性是私有的

只能用公共的getter和setter

static activeCount() : 得到程序当中所有活跃线程的总数

\*: 什么是活跃线程 = 就绪 + 运行 + 阻塞

\*\*\*\*\* : 一个静态方法是不能得到调用方法的那个对象的

static currentThread() : 得到正在运行的线程对象 = 当前线程

1.在主方法当中用于得到主线程的线程对象!

2.在run()调用的其它方法中 用这个方法得到线程对象

X.这个方法永远不会出现在run()当中 得到的结果等价于this

在主方法当中我们很轻松的能够得到任何一个线程对象 例如et

因为就在主线程当中创建的它们啊

但是一旦我们需要得到主线程对象呢? -\_-||

主线程根本就不是我们自己创建的...

只好借助currentThread()

setDaemon(true) : 设置线程成为守护线程~

\*: 首先线程的存在是给其它线程提供服务的，当程序当中只有守护线程的时候，守护线程自行结束~

\*: gc是个系统级别的守护线程~

interrupt() : 中断打断线程的阻塞状态

\*: sleep() join() 都能让线程阻塞

大家使用传参完成数据共享的时候务必注意，谁告诉你传参只能通过构造方法了?!给属性赋值的王道不是构造方法 而是 setter

多线程共享数据导致的并发错误~-----------------------------\*\*\*

为什么会出现并发错误?

根本原因:多个线程共享同一个对象

直接原因:线程体当中相邻的两行 未必能够连续执行~

导火线:时间片突然耗尽

如何解决并发错误?

临界资源: 多个线程共享的数据 被称作临界资源

锁标记: 互斥锁 互斥锁标记 锁旗标 监视器 Monitor

1st.synchronized修饰符

修饰代码块:

synchronized(临界资源){

需要连续执行的操作1;

需要连续执行的操作2;

}

修饰方法:

public synchronized void add(Object obj){

...;

}

等价于从方法第一行到最后一行全部加锁

而且是对调用方法的对象(this)加锁

public void add(Object obj){

synchronized(this){

...;

}

}

\*: StringBuffer StringBuilder

Vector ArrayList

Hashtable HashMap 在提到他们的区别的时候

线程同步特性不同 因为StringBuffer Vector Hashtable

底层大量的使用了synchronized修饰方法

\*: synchronized是一个很奇特的修饰符

synchronized特性不能被子类继承

\*: 单例模式的懒汉式需要使用synchronized修饰~

确保不会得到多个不同的实例~

2nd.java.util.concurrent.locks.ReentrantLock JDK5.0

- Doug Lea

Java包的工具包的并发包的锁包的可重入锁

ReentrantLock ->

lock(); unlock();

上锁 释放锁

互斥锁标记使用过多或者使用不当就会导致一个线程持有另一个线程想要得到的资源不释放的情况下，又去申请另一个线程已经持有的资源，从而双双进入对方资源的锁池当中，产生永久的阻塞 -> 死锁 DeadLock

1st.中美科学家联合国饿死事件

2nd.泉城路奔宝事件

3rd.AABB事件

如何解决死锁:

涉及到一块空间和三个方法

等待池 + wait() notify() notifyAll()

等待池:存放那些为了成全其它线程而主动释放掉自己已经持有的锁标记的那些线程的

wait():让当前线程放弃已经持有的锁标记并且进入调用方法的那个对象的等待池当中

\*:Benz线程体当中出现east.wait();

代表奔驰放弃east锁标记

并且进入east的等待池当中

notify():从调用方法的那个对象的等待池当中随机的唤醒一个线程

notifyAll():从调用方法的那个对象的等待池当中唤醒所有线程

\*: 注意这三个方法都必须在已经持有锁标记的情况下才能使用，也就是说如果abc.wait()或者abc.notify()，就必须先拿到abc对象的锁标记，否则无法操作成功，也就是说abc.wait()或者abc.notify()必然出现在synchronized(abc){..}

锁池 & 等待池 的区别?

锁池和等待池都是Java当中每个对象都有一份的空间，用于存放线程的

锁池存的是:想要拿到对象锁标记而没有成功的线程

等待池存的是:已经拿到锁标记然而为了成全其它线程又主动放弃的线程

1.进入的时候是否会释放资源

锁池不会释放资源 所以才会形成死锁

等待池需要释放资源才能进入

2.离开的时候是否需要调用方法

锁池不需要 一旦锁标记再度可用 线程自动离开

等待池必须有其它线程notify()或者notifyAll()

3.离开之后去往哪了

离开锁池之后线程直接返回就绪

离开等待池之后线程将直接前往锁池

线程池 => 提前预留活跃资源的资源池

假如一个线程的完整执行时间为T

T = t1 + t2 + t3 总共执行时间由三部分组成

t1 : 创建一个线程所消耗的时间

t2 : 线程的核心逻辑的执行时间 run()

t3 : 销毁一个线程所消耗的时间

那么如果run()当中的代码非常简练 则t2所占T比例就会很小

那么t1和t3就显得喧宾夺主 主次不分

已知Vector类的add()和remove()都是synchronized修饰的

现在有两个线程t1和t2 以及一个Vector对象 名叫 v

在t1线程调用v对象的add() 方法开始执行了 但是还没执行结束呢

时间片就耗尽了 此时t2线程抢到了时间片

问:

t2线程能不能调用v对象的add()? false

t2线程能不能调用v对象的remove()? false

已知Vector类的add()和remove()都是synchronized修饰的

现在有两个线程t1和t2 以及两个Vector对象 名叫 v1和v2

在t1线程调用v1对象的add() 方法开始执行了 但是还没执行结束呢

时间片就耗尽了 此时t2线程抢到了时间片

问:

t2线程能不能调用v1对象的add()? false

t2线程能不能调用v1对象的remove()? false

t2线程能不能调用v2对象的add()? true

t2线程能不能调用v2对象的remove()? true

三种线程池：

Executors.newFixedThreadPool(); 可重用的

Executors.newCachedThreadPool(); 缓存机制的

Executors.newSingleThreadExecutor(); 单一实例的

class Moon{

private Moon(){}

private static Moon only;//懒汉式

public static synchronized Moon getOnly(){

if(only == null)

only = new Moon();

return only;

}

}

class Sun{

private Sun(){}

private static Sun only = new Sun();//醉汉式

public static Sun getOnly(){

return only;

}

}

第三种创建线程的方法

class ThreadThree implements Callable<String>{

它修正了run方法的两个不足之处:

1st.run被定义为void方法 所以执行后不能返回数据

2nd.run根本没有throws声明 逼迫我们try catch

@Override

public String call()throws Exception{

for(int i = 0;i<1000;i++){

System.out.println("我是创建线程的第三种方式");

}

return null;

}

}

===========================================================================================

java.io.File

构造方法:

new File(String 路径)

new File(String 父目录,String 文件名)

new File(File 父目录对象,String 文件名)

\*: 相对路径 or 绝对路径

绝对路径 是从盘符或者根目录开始定位的

相对路径 相对于程序当前所在的位置开始定位的

\*: 操作系统兼容性问题 => 路径分界符的使用

/ 兼容windows和unix \ windows的专属

Sun在File类当中还提供了专门的解决途径

File类特等优先级方法:

static listRoots():列出当前计算机的所有根目录

String[] list():列出一个目录下所有的文件名字

File[] listFiles():列出一个目录下所有的文件对象

\*: 请注意list()和listFiles()都必须是目录才能调用

必须保证调用它们的File对象isDirectory();

File类一等优先级方法:

exists():判断File对象指代的文件或者目录是否已经存在

\*: File对象可以指代一个已经存在的文件，也可以指代一个当前根本不存在的文件，我们就需要有一个方法能够检测文件是否存在

isFile():判断File对象指代的是不是一个文件

isDirectory():判断File对象指代的是不是一个目录

\*: 你要明白Java当中的File，既可以代表一个文件，也可以代表一个目录，那么就需要有方法能够判断究竟是文件还是目录

\*: 如果文件不存在 则这两个方法通通返回false

length():得到File对象指代的文件的字节个数 注意long类型!

length length() size() length()

数组 字符串 集合 文件

int int int long

\*:这个方法只能用于文件 不能用于目录 否则结果不可预知

delete():删除File对象指代的文件或者目录

\*: 这个删除根本不知道回收站是何物

\*: 如果要删除的是一个目录 则必须保证目录为空 否则删除失败

renameTo():重命名文件或者目录

\*: 这个方法需要两个File对象 a.renameTo(c);而且要求a必须exists() && c必须!exists()

\*: 这个方法可以传入不同的目录结构从而实现移动

mkdirs():创建多层不存在的目录结构

getName():得到文件或者目录的名字

getParent():得到文件或者目录的父目录

getAbsolutePath():得到文件或者目录的绝对路径~

setLastModified():设置文件的最后一次修改时间~

lastModified():得到文件的最后一次修改时间~

\*: 课题转变 如何解析时间戳

1> java.util.Date

getYear() + 1900

getMonth() + 1

getDate()

getHours()

getMinutes()

getSeconds()

2> java.util.Calendar

get(1) get(2)+1 get(5) get(11) get(12) get(13)

3> java.text.SimpleDateFormat

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

format(time);

===========================================================================================

IO流 => Input 输入 Output 输出

流是一个很形象的概念 流是数据从源点传输到汇点的管道

IO流的分类:

按照方向分: 输入流 输出流 \*:参照物=>Java程序

按照单位分: 字节流 字符流

按照功能分: 节点流 过滤流(包装流 处理流)

InputStream 所有字节输入流统一的父类 抽象类

int read()

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

int read(byte[] data,int off,int len)

OutputStream 所有字节输出流统一的父类 抽象类

write(int data)

write(byte[] data)

write(byte[] data,int off,int len) \*\*\*\*\*

FileInputStream 输入流 字节流 节点流

FileOutputStream 输出流 字节流 节点流

\*:它们的构造方法都允许提供String文件路径或者File文件对象

\*:虽然它们贵为节点流，能够连接文件，但是绝对不能连接目录!否则直接出现异常

\*:节点输出流有极强的杀伤性，它创建对象的时候，如果连接的文件不存在，能够自动创建

#:File类其实有个方法叫createNewFile()

能够创建新的空白文件 - 咱没讲 一辈子不用

但是如果连接的目录结构都不存在 则直接异常

#:File类有个方法叫mkdirs() 咱们讲过 你要用!

节点输出流创建对象的时候如果连接的文件已经存在，也会被新的空白文件直接替换，而且没有提示，如果需求是追加操作也就是在原有内容之后继续写内容，那么可以构造方法传参指定追加模式开启

new FileOutputStream("3.mp3",true); //追加模式

new FileOutputStream("3.mp3"); //替换模式

\*: FileInputStream 以-1作为读取结束的标识

\*: FileInputStream 最常用的是read(byte[] data) 因为无参的太慢

\*: FileOutputStream 最常用的却是write(byte[],int,int)

\*: 无论何时何地 千万不要让节点输入流和输出流同时连接同一个文件 否则怎样都是错

BufferedInputStream 字节流 输入流 过滤流(包装流 处理流)

BufferedOutputStream 字节流 输出流 过滤流(包装流 处理流)

\*:作为过滤流的它们，是为了给原本的节点流添加巨大的缓冲空间，从而提高每次读写的吞吐量，进而提高效率的

\*:作为过滤流的它们，不能直接连接文件，只能连接其它的流

\*:它们的构造方法都允许传入第二个参数，指定缓冲空间大小

\*:BufferedInputStream 同样以-1作为读取结束

\*:BufferedInputStream 最常用的是read()

\*:BufferedOutputStream 最常用的就是write(int data)

DataInputStream 字节流 输入流 过滤流(包装流 处理流)

DataOutputStream 字节流 输出流 过滤流(包装流 处理流)

\*:作为过滤流的它们 是为了给原本的节点流添加读写基本数据类型的功能的

\*:boolean char byte short int long float double

\*:DataInputStream 提供的核心方法叫readXxxx()

\*:DataOutputStream 提供的核心方法叫writeXxxx()

\*:DataInputStream 不再以-1作为读取结束的标识，而是如果已经到达文件结尾还继续尝试读取，将直接出现EOFException => End Of File

\*:在文件当中没有任何标识哪一个是个数据，意思就是如果我们写出一个int，可以读取回来一个int 或者两个short或者四个byte

ObjectInputStream 输入流 字节流 过滤流(包装流 处理流)

ObjectOutputStream 输出流 字节流 过滤流(包装流 处理流)

\*:它们都是过滤流 它们存在是为了让原本节点流添加读写对象的功能

\*:它们都是过滤流 所以不能直接连接文件 只能连接其它的流

\*:ObjectInputStream -> readObject();

\*:ObjectOutputStream -> writeObject();

\*:一个对象想要被持久化到磁盘上，就必须要先序列化(Serializable)如果这个对象有其它引用类型的属性，就连这个属性的类型，也必须要实现序列化接口，如果某些属性无关紧要，不需要参与持久化

可以使用:transient修饰 短暂的 转瞬即逝的 不参与持久化的

如果要持久化的是一个集合对象，则要求集合当中的元素类型，必须都实现序列化接口，如果要持久化的是一个使用了比较器的TreeSet或者TreeMap，就连比较器对象也必须要序列化

\*:ObjectInputStream同样不以-1作为读取结束的标识

也是EOFException异常的机制

io流必须要学会的事情？

1st.使用标准的FileInputStream+FileOutputStream \*\*\*\*\*

结合一个大数组完成文件复制

2nd.使用FileInputStream+BufferedInputStream结合 \*\*\*

FileOutputStream+BufferedOutputStream

一次一个字节的完成文件复制

3rd.将内存当中的int变量 保存到文件当中 \*\*\*

4th.从文件当中读取那个int变量回来 \*\*\*

5th.将一个对象保存到文件当中 \*\*\*

6th.读取那个对象到程序当中来 \*\*\*

7th.以行为单位读取文本文件 \*\*\*\*\*

8th.以一行为单位写出文本文件 \*\*\*\*\*

新内容登场 : 字符流

Reader 所有字符输入流统一的父类 抽象类

int read()

int read(char[] data)

int read(char[] data,int off,int len)

Writer 所有字符输出流统一的父类 抽象类

write(int data)

write(char[] data)

write(char[] data,int off,int len)

FileReader 字符流 输入流 节点流

FileWriter 字符流 输出流 节点流

\*: 此处省略一大堆一大堆的字! 需要拿过来吗?

\*: 已经学过了字节流了 为什么还要学字符流呢?

\*: 那么如果字符流这么牛 为啥还要学字节流呢?

复制文件用字节流 读取文字用字符流

BufferedReader 字符流 输入流 过滤流

BufferedWriter 字符流 输出流 过滤流

\*: 作为过滤流的它们是为了给原本的节点流添加变长的缓冲空间

从而实现以行(hang)为单位的读写

\*: BufferedReader -> String readLine()

\*: BufferedWriter -> write(String) + newLine()

\*: BufferedReader不再以-1作为读取结束 String只能返回null

\*: BufferedWriter 带缓冲的输出流 请注意及时关流 或者 flush();

PrintStream

PrintWriter 究竟好在哪里?

1.它的构造方法既能传入String或者File也能传入其它的流

\*:既能当做节点流 又能当做过滤流

2.它的构造方法既能传入FileOutputStream 也能传入FileWriter

\*:既能连接字节流 又能连接字符流

3.当它作为节点流使用的时候 构造方法第二个参数可以指定字符集

\*:精通多国语言

4.当它作为过滤流的时候 可以指定自动清空缓冲

new PrintWriter(new FileWriter("aaa.txt"),true);

5.它的方法很实用 println() = write() + newLine();

6.我们对于她的孪生姐妹特别熟悉!

System.out 的out属性 是PrintStream类型的

\*\*\*\*\*:新内容来了：桥转换器，字节流通向字符流的桥梁，使用桥转换器能够将字节流变成字符流

InputStreamReader r = new InputStreamReader(is);//InputStreamReader OutputStreamWriter

===========================================================================================

java.net.URL => 统一资源定位符 Uniform Resource Locator

=> 网址

URL url = new URL("网址");

URLConnection uc = url.openConnection();

InputStream is = uc.getInputStream();

OutputStream os = uc.getOutputStream();

URL url = new URL("http://www.etoak.com/resources/images/zkmax.png");

URLConnection uc = url.openConnection();

InputStream is = uc.getInputStream();