1. System方法

System.exit()停止虚拟机运行

实现Comparable接口，重写比较规则，用于设定TreeSet比较规则

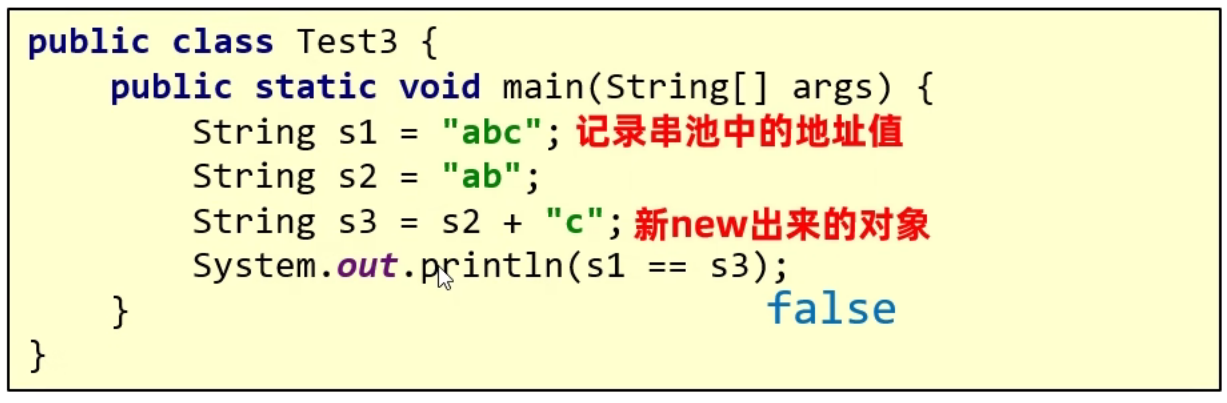
1. long的变量数据后面要加L，float的变量数据要加f

string的加法是连接，有string参与的+是连接 “string”+123 string123

有变量参与的拼接会创建新的字符串，也就是new出来一个字符串，会开辟新的地址

无变量参与，直接将右侧合并一个字符串再赋值





new出来的放到堆内存中

随机数1~20 Random r = new Random(); int rand = r.nextInt(20)+1;

Scanner说明 nextInt()整数 nextDouble小数 next()字符串 空格 制表符 回车即不接收

nextLine()字符串 回车不接收

“%s是%s”,a,b 其中a=“小明”，b=”学生” 实现类似C语言的输出

==号：对于基本数据类型，比的是数据值；对于引用数据类型，比的是地址值

Ctrl+左键 查看源码

Ctrl + p 看构造器的参数提示

赋值语句，写完右边 Ctrl + Alt + v快捷键生成左边

Break Flag;跳出Flag的语句块

快速创建函数：先写函数调用，然后alt + 回车

1. 数组的定义 int[] array = new int[] {1,2,3}；静态数组 int[] array = {1,2,3}；静态数组

Int[] a = new int[10];长度为10的动态数组

Int[][] a = new int[][];二维数组

数组名.length获取数组长度

Arr.fori 对数组遍历快捷键

1. 快捷键创建构造器和set get：alt + insert 或alt + Fn + insert

或者利用插件（ptg）：ptg to JavaBean即可 ctrl + shift + ,

对象内存运行时的操作过程：

1.加载class文件 2.声明局部变量 3.在堆内存中开辟一个空间 4.默认初始化

5.显示初始化 6.构造方法初始化 7.将堆内存中的地址值赋值给左边的局部变量

Ctrl+T快捷键显示new时构造器的参数列表

1. String
2. 创建
3. 直接用“”赋值，如果引号内容没有，则在串池创建一个然后赋值，如果有，则直接拿来用
4. String a = new String(char数组名或byte数组名) 创建一个字符串再赋值，每次都需要创建即可能会有内存浪费
5. 方法
6. boolean equal(对象名)和equalIngoreCase(对象名) 都是比较字符串的内容，返回布尔值

前者不忽略大小写，后者忽略（英文的）

1. char charAt(int index)根据索引返回字符
2. int length()返回字符串的长度
3. String substring(int beginIndex, int endIndex) 截取，左闭右开 返回这部分的字符串 还有一个重载substring(int beginIndex) 截取至末尾
4. String replace(旧，新) 新替换旧字符串，如果输入的旧值不在字符串中，则不替换
5. char[] toCharArray()将字符串变为字符数组
6. StringBuilder
7. 特点
8. 内容可变的字符串
9. 常运用于字符串拼接和反转
10. 构造
11. 空参 和 有一个String的参数
12. 方法
13. StringBuilder append(任意类型) 添加数据，返回对象本身
14. StringBuilder reverse() 反转内容
15. Int length()返回字符的个数
16. String toString() 把StringBuilder转为String
17. StringJoiner(JDK8)
18. 构造
19. （间隔符号）或者（间隔符号，开始符号，结束符号）
20. 方法

add，toString，length

1. 集合
2. 特点

类似数组，但只能存引用数据类型，长度随元素个数自动变化

存储的是数据的地址！

1. 构造

ArrayList<引用数据类型> name = new ArrayList();

1. 方法
2. Boolean add(E e) 添加元素，返回值代表是否成功
3. Boolean addAll(ArrayList array) 添加多个元素，传入集合
4. Boolean remove(E e) 删除元素，返回值代表是否成功
5. E remove(int index) 删除指定索引元素，返回被删除的元素
6. E set(int index, E e) 修改指定索引元素，返回原来的元素
7. E get(int index) 获取指定索引元素
8. Int size() 集合的长度，也就是元素个数
9. 包装类

Char -> Character int -> Integer 其余基本数据类型的首字母大写即可

1. 继承
2. 成员变量：都可以继承，非私有可以直接调用，private不能直接调用，需要通过set和get
3. 成员方法：非private、static、final都可以继承，也就是虚方法表的可以被继承
4. 构造方法不能被继承
5. 方法重写：只有虚方法表的可以被重写
6. Super()访问父类的构造方法，子类写构造方法时第一行必须写super()（你不写，虚拟机会自动调用父类的默认构造方法），也可以写super（参数）调用父类带参构造方法
7. 多态
8. 父类 name = new 子类（）
9. 调用成员变量的时候，编译看左边，运行看左边

调用成员方法的时候，编译看左边，运行看右边

1. 好处：定义方法时，用父类作为形参接收子类
2. 坏处：父类不能调用子类特有的方法

解决方案：1）if(对象 instanceof A类) 强制类型转换成A类;

If(对象 instanceof A类 新变量名)(JDK14才有

1. 权限访问符

Private<空着不写<protected<public



1. 代码块
2. 局部代码块 { 代码 } 可以结束代码的生命周期
3. 构造代码块 优先构造方法执行，可以提取构造方法相同的部分（目前通过this调用其他构造方法，或是调用成员方法实现，更加灵活）
4. 静态代码块 static{代码} 只执行一次，随类加载，给数据初始化、
5. 接口
6. public interface 接口名{}
7. public class 类名 implements 接口名（可以多个接口）{}
8. 接口的变量只能是而且默认是public static final；JDK7以前，方法只能是而且默认是public abstract
9. 接口可以再实现单个接口或是多个接口（类似多继承）
10. JDK7新增默认方法： public default void 方法名（）{方法体}

注意：实现接口时，如果重写默认方法，不能再写default

如果实现多个接口，多个接口含有相同名字的默认方法，默认方法必须重写

1. JDK8新增静态方法：public static void 方法名（）{方法体}

注意：不能重写

1. JDK9新增私有方法：private void 方法名（）{方法体} 或者 private static void

前者给默认方法调用，后者给静态方法调用

1. 应用
2. 接口是行为的抽象，想让类拥有这个功能，就实现这个接口
3. 接口的多态，当方法的参数是接口时，可以传入实现这个接口的任意类的对象
4. 适配器

当接口有多个默认方法，我只想重写一个方法时，利用一个XXXAdapt抽象类去实现接口，再用类去继承这类，重写想要的方法

1. 内部类
2. 当B类表示的事物是A类的一部分，且B类单独存在没有意义时，B类设计成内部类，如人的器官。

A类不能访问B类的成员，必须先创建B类再访问。B类可以访问A类成员

1. 成员内部类
   1. 可以用权限访问符修饰，JDK16可以定义静态变量
   2. 获取成员内部类对象：外部类编写方法提供内部类对象

外部类.内部类 name = new 外部类（）.new 内部类（）（内部类是public）

* 1. 创建内部类对象的时候，会隐含存储外部类对象的地址值 Outer.this

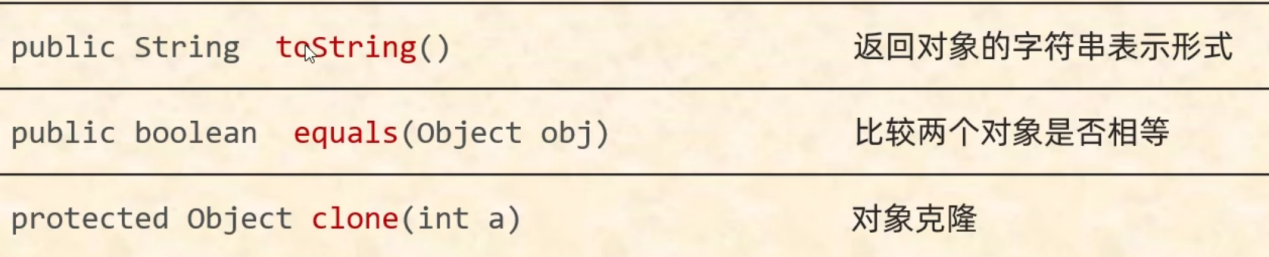
1. 静态内部类
   1. 创建 外部类.内部类 name = new 外部类.内部类（）
   2. 调用静态内部类的方法：非静态方法，对象名.调用；静态方法，外.内.调用
2. 局部内部类 写在成员方法里，类似局部变量
3. 匿名内部类
   1. new 类名或是接口名 （）{方法重写}；
   2. 本质是对象的创建，是一个类的子类对象或者接口的实现类对象
   3. 当方法的参数是接口或者类时，可以传递接口的实现类对象或者子类对象

如果实现类只要使用一次，就可以用匿名内部类简化代码

1. 常用API
2. Math abs,ceil(向上取整),floor(向下取整),round(四舍五入),max,random(0.0~1.0),pow
3. System exit(停止虚拟机),currentTimeMillis(返回当前系统的时间毫秒值) arraycopy(拷贝数组)
4. Runtime

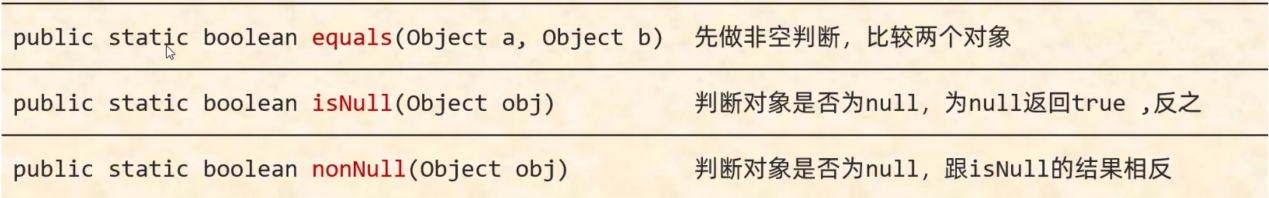


1. Object

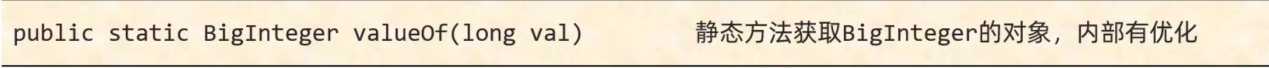


默认浅克隆，深克隆需要重写方法或者用第三方

1. Objects



1. BigInteger
   1. 构造



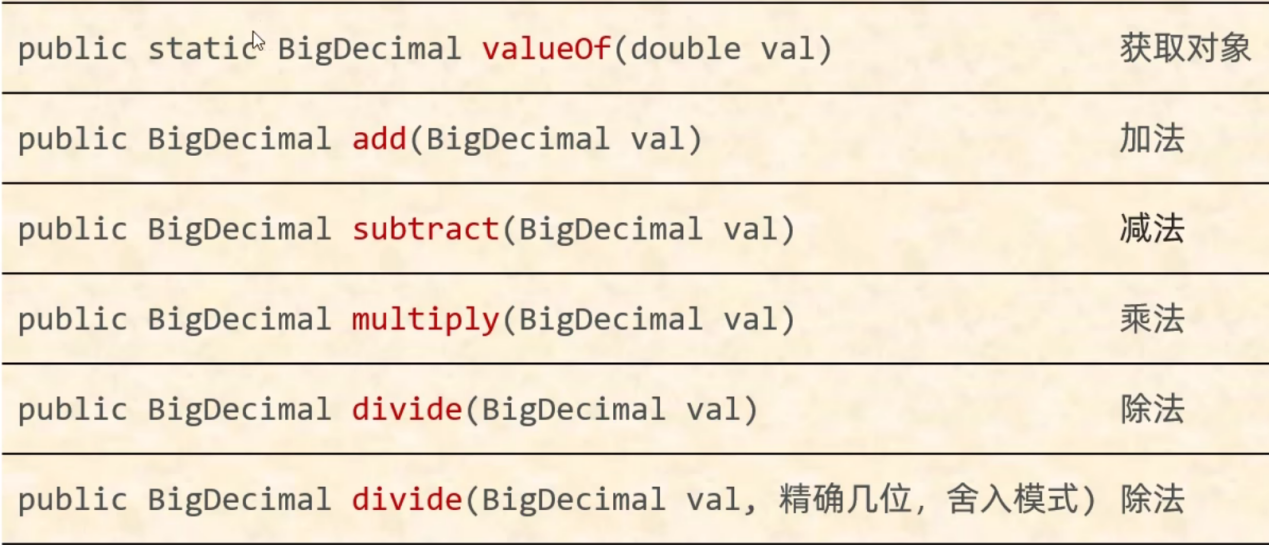
* 1. 方法



1. BigDecima
   1. 构造

new BigDecima(字符串)，BigDecima.valueof(没有超出double的double类型)

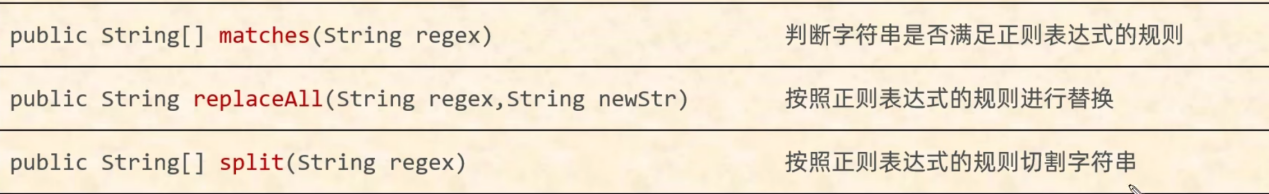
* 1. 方法



1. 正则表达式（string.matches(“规则”)），返回布尔值







(?i)忽略后面的大小写

捕获分组：(.).+\\1 \\1代表再用一次第一组（\*分组以左括号为基准\*）

正则内部使用\\数字，外部使用$数字 replaceAll(“(.)\\1+”,”$1”)

非捕获分组：(?=) (?:) (?!) 不占用组号 :用最多

[(28条消息) Java 正则表达式（深度长文）\_java正则表达式\_senxu\_的博客-CSDN博客](https://blog.csdn.net/senxu_/article/details/126109760)

1. 爬虫

String 要爬取的内容 Pattern p = Pattern,compile(正则表达式)；//爬取的规则

Matcher m = p.matcher(string) 拿m去读取string，找出符合P规则的字串

While（m.find()）{sout(m.group())//输出}

* 1. 条件爬取

?=8|11|17 爬取带有8或11或17的数据，但只获取前面的东西，不带数字

?:8|11|17 带数字

?!8|11|17 不爬带有8或11或17的数据

* 1. 贪婪与非贪婪

ab+ 贪婪 ab+? 非贪婪

1. JDK7时间类
   1. Date

Date d = new Date()//获取系统当前时间

Date d = new Date(long date)//获取指定时间 如0L

有set和get方法

* 1. SimpleDateFormat

空参构造获取默认时间格式，含参构造获取指定时间格式

y年M月d日H时m分s秒 参数：yyyy年MM月dd日 HH:mm:ss

成员方法：format(Date) 格式化Date返回string

parse(String) 从字符串变成date(构造时的时间格式要和string的一致)

* 1. Calendar（抽象类）

使用静态方法getInstance()获取当前时间的日历对象

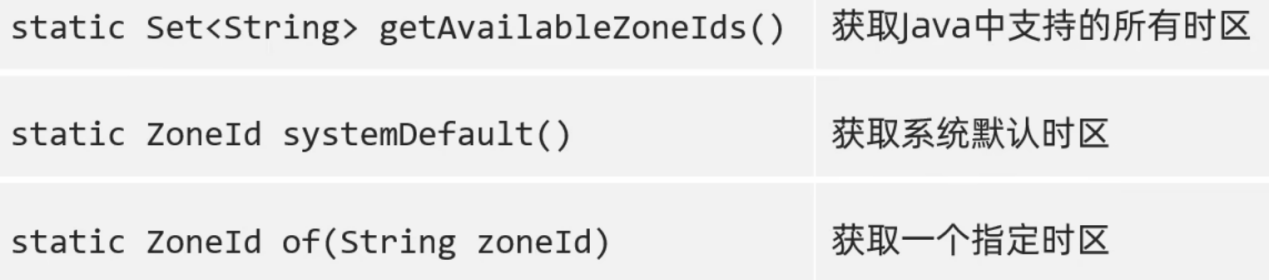
成员方法：



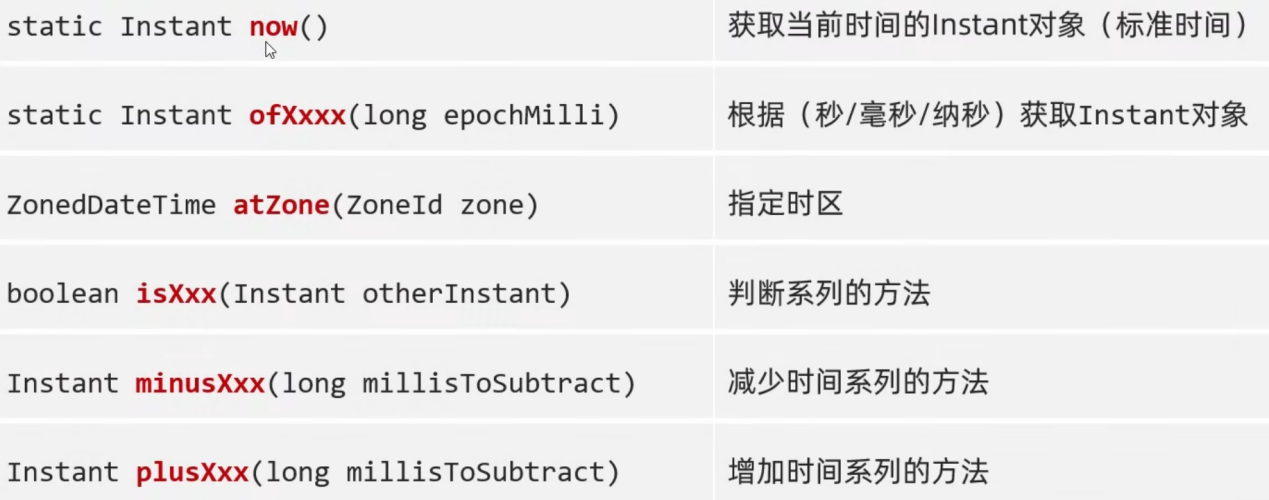
1-年（YEAR） 2-月(MONTH) 3-一年中的第几周(WEEK\_OF\_YEAR) 4-一个月中的第几周(WEEK\_OF\_MONTH) 5-一个月的第几天(DAY\_OF\_MONTH) DAY\_OF\_WEEK一周中的第几天（第一天为周日）

获取的月份为0~11，需要自己+1

1. JDK8时间类
   1. Date相关类
      1. ZoneId时区



* + 1. Instant时间戳



* + 1. ZoneDateTime带时区的时间

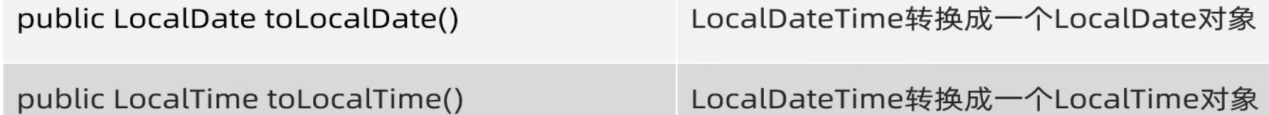


* 1. 日期格式化类

DateTimeFormatter用于时间的格式化和解析



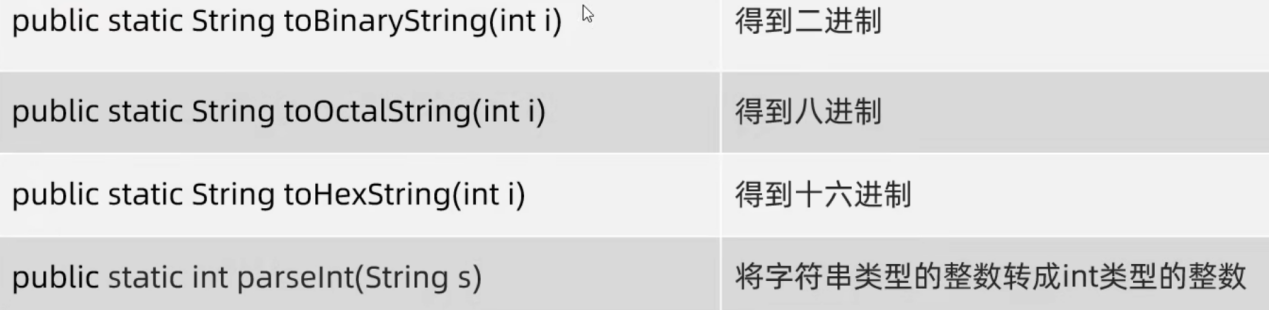
* 1. 日历类
     1. LocalDate年、月、日
     2. LocalTime时、分、秒
     3. LocalDateTime年、月、日、时、分、秒



* 1. 工具类
     1. Duration用于计算两个“时间”间隔（秒，纳秒）
     2. Period用于计算两个“日期”间隔（年、月、日）
     3. ChronoUnit用于计算两个“日期”间隔

1. 包装类

主要是Integer和Character



1. 算法
   1. 查找
      1. 二分查找 适合排序 mid=min + (max – min)/2
      2. 插值查找 适合排序且数据均匀，否则不如二分 mid=min + (key-a[min])/(a[max]-a[min]) + max
      3. 斐波那契查找
      4. 分块查找
      5. 哈希查找
      6. 树表查找
   2. 排序
2. 数据结构
   1. 树
      1. 遍历

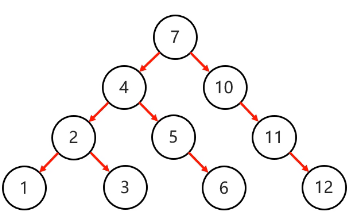
前序遍历：当前节点->左子节点->右子节点

中序遍历：左->当前->右

后续遍历：左->右->当前

层序遍历：一层一层，从上至下，从左至右

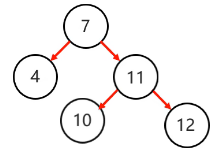
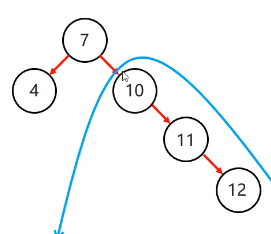
* + 1. 二叉树 一个节点包括父节点，左子节点，右子节点，值
    2. 二叉查找树 左子节点值<本节点值<右子节点值，左右子节点值相等则只存一个



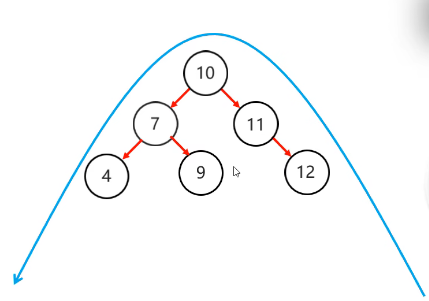
* + 1. 平衡二叉树 任意节点的左右子树高度差不超过1

确定支点：从添加的支点开始，不断往父节点找不平衡的节点

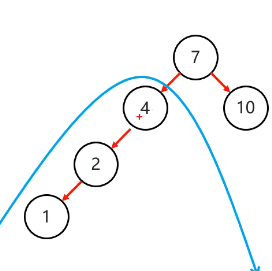
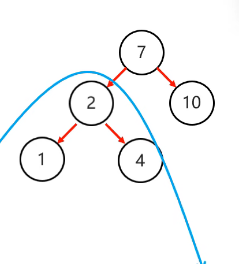
左旋：（1）以不平衡的点作为支点，把支点左旋降级，变成左子节点，晋升原来的右子节点



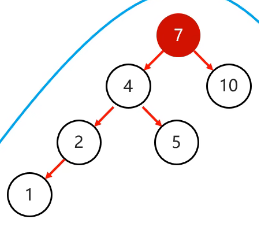
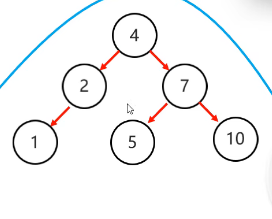
（2）以不平衡的点作为支点，将根节点的右侧往左拉，原先的右子节点变成新的父节点，并把多余的左子节点出让，给已经降级的根节点当右子节点

右旋：（1）以不平衡的点作为支点，把支点右旋降级，变成右子节点，晋升原来的左子节点

1. 以不平衡的点作为支点，就是将根节点的左侧往右拉，原先的左子节点变成新的父节点，并把多余的右子节点出让，给已经降级的根节点当左子节点

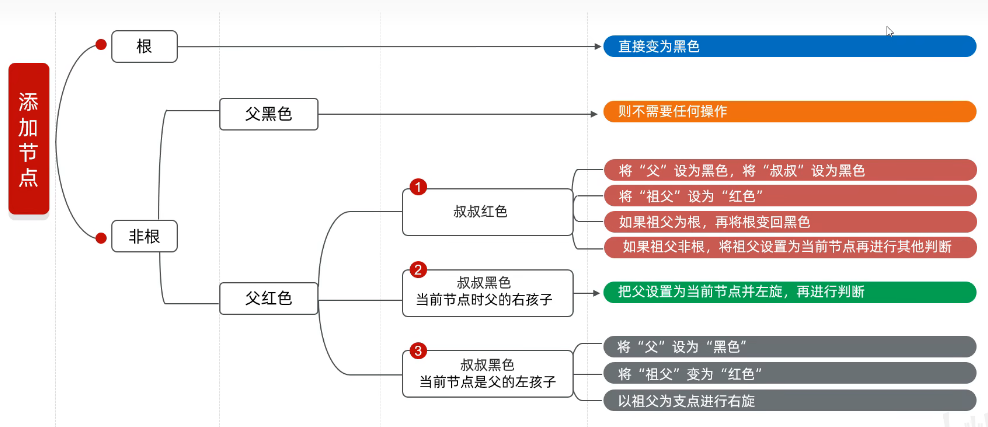
 

* 1. 左左（添加的元素在左子树的左子树）：一次右旋
  2. 左右（添加的元素在左子树的右子树）：局部左旋变成左左，再右旋
  3. 右右：一次左旋
  4. 右左：局部右旋变成右右，再左旋
     1. 红黑树 特殊的平衡二叉树，有特殊的旋转规则：红黑规则

红黑规则：

* 每一个节点或是红色，或是黑色
* 根节点必须是黑色
* 如果一个节点没有子节点或者父节点，则该节点相应的指针属性值为Nil，这些Nil视为叶节点，每个叶节点是黑色的
* 不能出现两个红色节点相连的情况
* 对每一个节点，从该节点到其所有后代叶节点的简单路径上，均包含相同数目的黑色节点

添加的节点：默认红色



1. 泛型<>
   1. <>中只能写引用数据类型，目的统一数据类型
   2. 泛型类

修饰符 class 类名<E> { } E为任意数据类型

* 1. 泛型方法

修饰符 <E> 返回值类型 方法名（类型 变量名）{ }

* 1. 泛型接口

修饰符 interface 接口名 <E> { }

用法:实现类也为泛型类 实现接口的时候明确类型 implements 接口名<明确类型>

* 1. 通配符

<?: extends E>泛型为E的子类 或者 <?: super E>泛型为 E 的父类

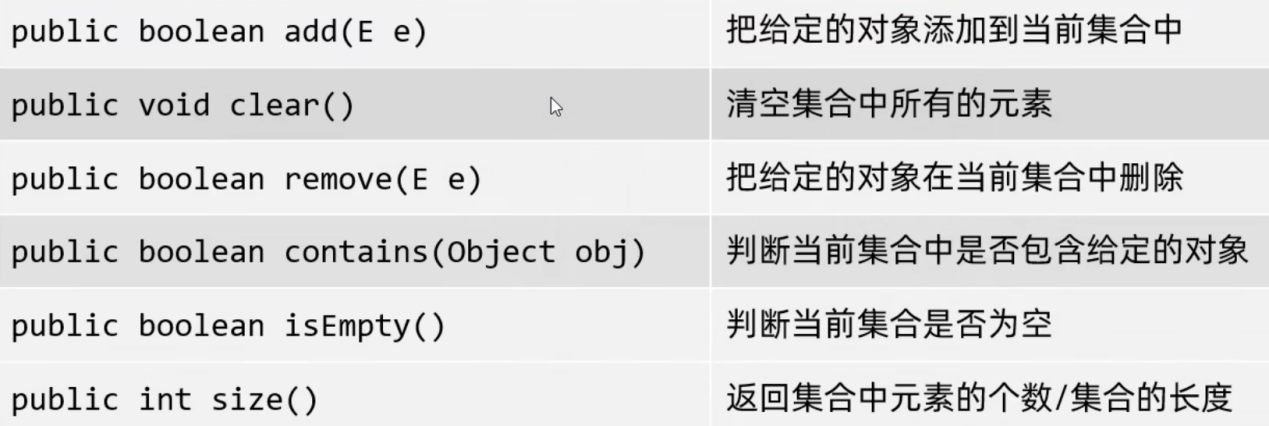
1. 集合进阶 Collection

单列集合List系列：添加的元素是有序、可重复、有索引

单列Set系列：添加的元素是无序、不重复、无索引

双列Map

单列集合的方法：



Add细节：List永远返回true（允许重复），set如果重复就会返回false

Remove细节：删除的元素不存在返回false

Contains细节：如果对象是自定义类的实例，需要重写equals方法

* 1. 单列集合（ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet）
     1. List方法

List专门的迭代器：列表迭代器 ListIterator

该迭代器也有类似的方法：next() hasnext() add() remove() previous() hasprevious() 注意previous必须要迭代器指针指向末尾才能用（迭代器创建时，指针默认指向第一个）

LinkedList（底层数据结构是双向链表，用于增删操作多的数据）

方法：

Void addFirst(E e) 在该列表开头插入指定元素

Void addLast(E e) 将指定元素追加到此列表的末尾

E getFirst() 返回此列表的第一个元素

E getLast() 返回此列表的最后一个元素

E removeFirst() 从此列表中删除并返回第一个元素

E removeLast() 从此列表中删除并返回最后一个元素

* + 1. HashSet 无序、不重复、无索引 （用得最多，用于去重）

底层是哈希表，哈希表是数组+链表+红黑树 添加元素根据index = (length-1) & 哈希值

哈希值（hashCode()方法获取）默认使用地址值计算，不同对象的哈希值不同

重写hashCode()方法，相同属性的对象的哈希值相同，但小部分情况下，不同属性值或地址值的哈希值也相同（哈希碰撞）

底层原理：

* 创建一个默认长度为6，默认加载因为0.75的数组，数组名为table
* 根据元素的哈希值跟数组的长度计算出应存入的位置
* 判断当前位置是否为null；如果为null直接存入
* 如果不为null，表示有元素，则调用equals方法比较属性值
* 一样，不存；不一样，存入数组，形成链表

JDK8以前：新元素存入数组，老元素挂在新元素下面

JDK8及以后：新元素直接挂在老元素下面。链表长度超过8，且数组长度≥64，自动转换为红黑树

如果集合中存储的是自定义对象，必须重写hashCode()和equals()

* + 1. LinkedHashSet （继承于HashSet）有序、不重复、无索引

有序：保证存储和取出的元素顺序一致

底层仍是哈希表，但每个元素多了一个双链表记录存储顺序

* + 1. TreeSet 可排序、不重复、无索引

底层是红黑树，不是哈希表，不需要重写HashCode和equals

对于Integer、Double默认从小到大排序

对于字符、字符串类型，按照字符在ASCII表中的字典序排序

默认比较顺序为：数字的从小到大和字典序

当存储对象时，对象类实现Comparable接口，重写方法即可根据需求比较，默认从小到大

或是TreeSet<E> ts = new TreeSet<>(new Comparable<E>(){重写方法});

* 1. 双列集合

一次添加可添加两个元素，一个为键（不可重复），另一个为值（可重复），二者一一对应，称为键值对、键值对对象

方法：

V put(K key, V value) 键不存在，直接添加元素，返回null；键已经存在，覆盖旧值，返回旧值

V remove(Object key) 根据键删除键值对元素，返回被删除的值

void clear() 删除索引键值对元素

boolean containsKey(Object key) 判断集合是否包含指定的键

boolean containsValue(Object Value) 判断集合是否包含指定的值

boolean isEmpty() 判断集合是否为空

int size() 集合的长度，也就是集合中键值对的个数

* + 1. HashMap 键的缘故导致，无序、不重复、无索引

方法等同Map的方法

如有需要，键需重写HashCode()和equals()；但值不需要重写，值可以重复

底层也是哈希表，和HashSet一样

* 1. 单列集合遍历
     1. 迭代器

Collection获取迭代器：Iterator<E> iterator() 返回迭代器对象，默认指向集合0索引

常用方法：boolean hasNext() 判断当前位置是否有元素

E next() 获取当前位置的元素，并将迭代器对象移向下一个位置、

Remove() 删除当前位置的元素

\*\*\*迭代器遍历的时候不要用集合的方法添加或删除元素，用迭代器的方法\*\*\*

* + 1. 增强for循环

只有单列集合和数组能用

for(数据类型 变量名：集合/数组){ }

快捷键 集合名.for

For中的变量名只是个第三方变量，不会影响集合中的元素

\*\*\*遍历的时候不要用集合的方法添加或删除元素\*\*\*

* + 1. Lambda表达式遍历

default void forEach(Consumer<?: super T> action):

Consumer是一个函数式接口，因此参数为实现类对象，可以用匿名内部类和Lambda表达式

* + 1. 五种遍历的选择

遍历的时候想要删除元素，用迭代器

遍历的时候想要添加元素，用列表迭代器

仅仅想遍历，用增强for或者Lambda

遍历的时候想操作索引，用普通for

* 1. 双列集合遍历
     1. 将键存入单列集合

Set<E> set = map.Keyset(); //返回存入键的单列集合

通过单列集合遍历，获取对应的值

E value =map.get(Key);

* + 1. 通过键值对对象

存入单列结合

Set<Map.Entry<K, V> entries = map.entrySet() //返回存有键值对对象的单列集合(这一步可以省略，右部分可以写入增强for中)

遍历集合，得到键值对对象

E key = entry.getKey();//得到键

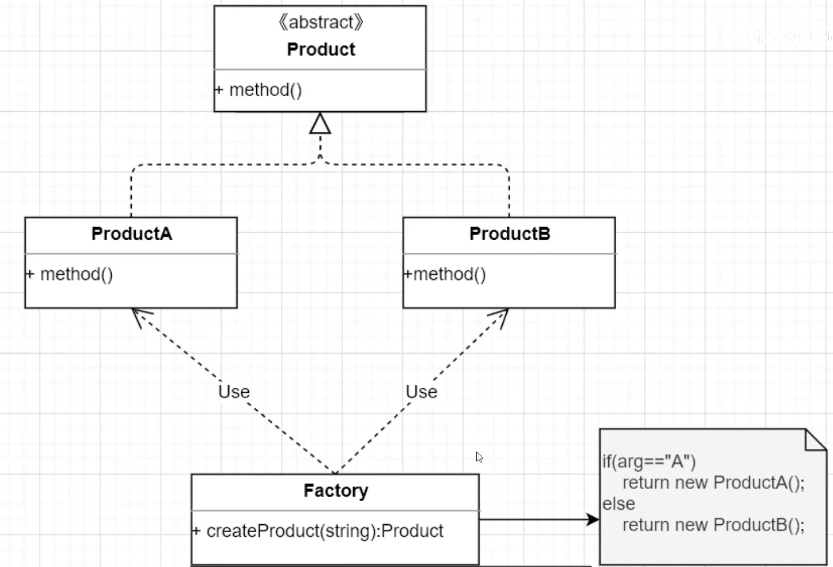
E value = entry.getValue();//得到值

* + 1. Foreach与lamba表达式

用法等同单列集合

Map.foreach(new BigConsumer<K key, V value>(){});

1. 工厂模式
   1. 简单工厂模式



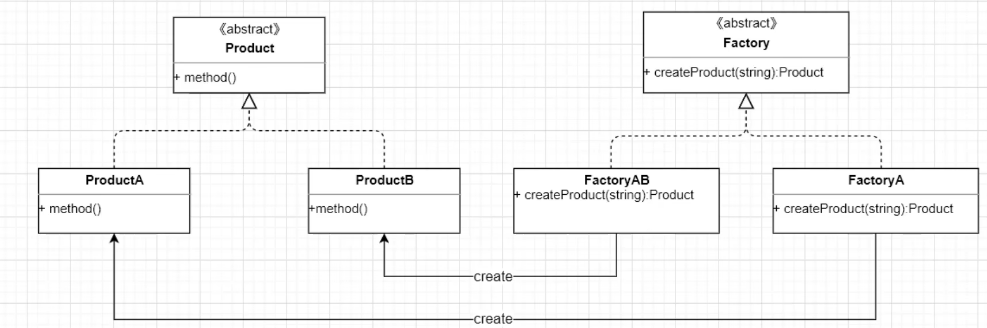
角色：简单工厂 抽象产品 具体产品

Create()方法：利用if或switch创建并返回具体产品

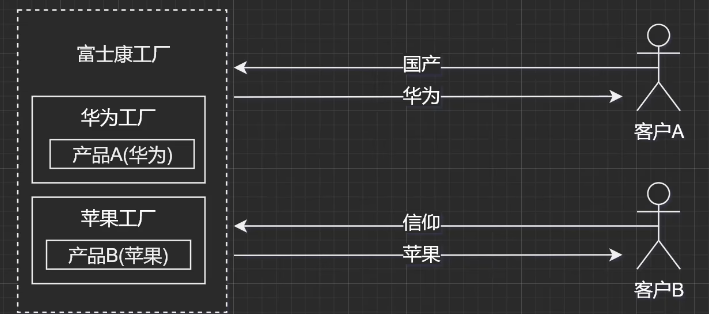
优点：对象的创建和使用分离

缺点：添加产品类需要修改工厂类，一旦产品多，逻辑将会很复杂，不符合开闭原则

* 1. 工厂方法模式



例如，Collection是一个抽象工厂接口，LinkedList和ArrayList（两个具体工厂）实现这个接口。当创建Iterator时，两个具体工厂根据自己的特点，实现Iterator接口，创建具体的LinkedListIterator和ArrayListIterator。



public class Client {

public static void main(String[] args) {

Factory factory = new HuaweiPhoneFactory();

Phone phone = factory.creatPhone();

phone.print();

}

}

interface Phone {//抽象产品

public abstract void print();

}

class Iphone implements Phone {//具体产品

@Override

public void print() {

System.out.println("苹果手机");

}

}

class HuaWeiPhone implements Phone {

@Override

public void print() {

System.out.println("华为手机");

}

}

abstract class Factory {//抽象工厂

public abstract Phone creatPhone();

}

class IphoneFactory extends Factory {//具体工厂

@Override

public Phone creatPhone() {

return new Iphone();

}

}

class HuaweiPhoneFactory extends Factory {

@Override

public Phone creatPhone() {

return new HuaWeiPhone();

}

}

角色：抽象工厂 具体工厂 抽象产品 具体产品

在以下情况下可以使用工厂方法模式：

* **一个类不知道它所需要的对象的类：**在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体工厂类创建；客户端需要知道创建具体产品的工厂类。
* **一个类通过其子类来指定创建哪个对象：**在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。
* 将创建对象的任务委托给多个工厂子类中的某一个，客户端在使用时可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定，可将具体工厂类的类名存储在配置文件或数据库中。

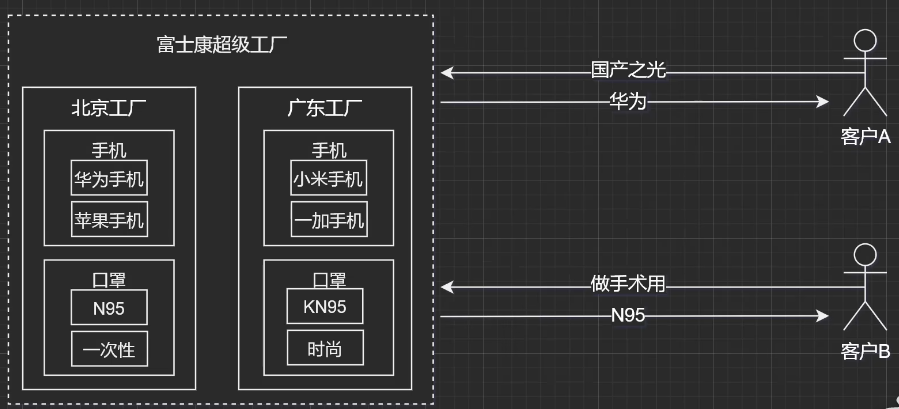
使用场景：

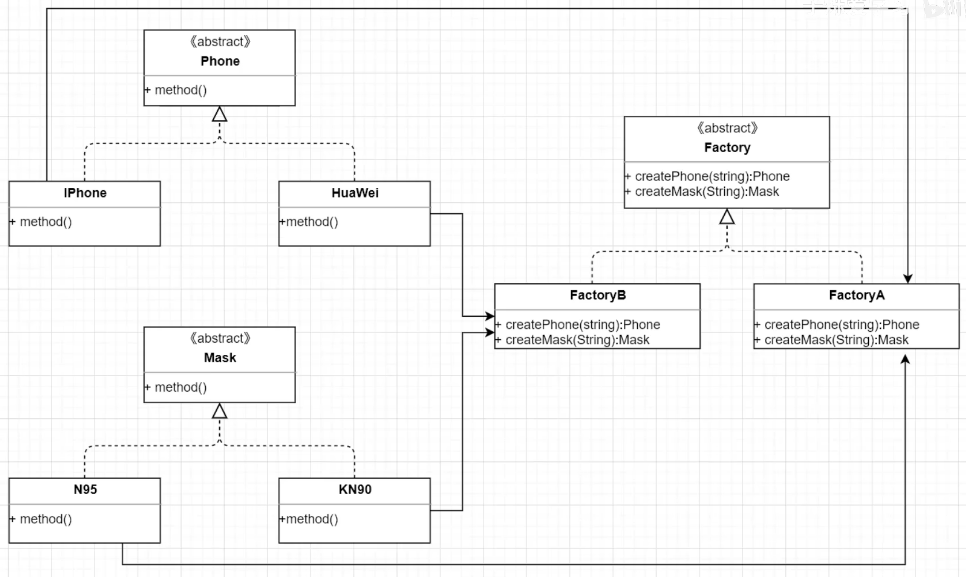
* 日志记录器：记录可能记录到本地硬盘、系统事件、远程服务器等，用户可以选择记录日志到什么地方。
* 数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据库可能有变化时。
* 设计一个连接服务器的框架，需要三个协议，"POP3"、"IMAP"、"HTTP"，可以把这三个作为产品类，共同实现一个接口。
* 比如 Hibernate 换数据库只需换方言和驱动就可以

优点：创建新产品C时，只需要创建C类和创建C的工厂类即可，符合开闭

缺点：一个具体工厂只能创建一个产品。

* 1. 抽象工厂模式





优点：扩展了工厂的功能，不单单生产手机。与工厂模式的区别是，新增了一个产品体系（口罩）

缺点：一旦新增产品体系，需要修改工厂逻辑

在以下情况下可以使用抽象工厂模式：

* 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有类型的工厂模式都是重要的。
* 系统中有多于一个的产品族，而每次只使用其中某一产品族。
* 属于同一个产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来。
* 系统结构稳定，不会频繁的增加对象。

1. 注解
   1. 内置注解

@Override 检查重写方法

@Deprecated 标记过时方法

@SuppressWarnings() 指示编译器去忽略注释解中声明的警告

参数

* deprecation 使用了不赞成使用的类或方法时的警告
* unchecked 执行了未检查的转换时的警告，如集合没有用泛型指定类型
* fallthrough 当Switch程序块直接通往下一种情况而没有Break时的警告
* path 在类路径、源文件路径等中有不存在的路径时的警告
* serial 当在可序列化的类上缺少serialVersionUID定义时的警告
* finally 任何finally子句不能正常完成时的警告
* all 所有警告

@FunctionalInterface java8，标识一个匿名函数或函数式接口

* 1. javaAPI提供的元注解，用于修饰注解

@Retention 标识这个注释解怎么保存，是只在代码中，还是编入类文件中，或者是在运行时可以通过反射访问

@Documented 标识这些注解是否可以包含在用户文档中

@Target 标识这个注解的作用范围

@Inherited 标识注解可被继承类获取

@Repeatable 标识某注解可以在同一个声明上使用多次

* 1. 自定义注解

格式：

元注解

public @interface 注解名 {}

属性：

允许int，String，enum，@interface和他们数组的返回值属性

{

int age();//需要给参数

String name() default “null”;//不给参数，默认为””的内容

}

使用：

@注解名(age = 12) //当只有一个属性且属性名为value时，@注解名(12)

1. 异常 Exception

Exception是异常的最高父类，有两个子类RuntimeException(运行时异常)和其他异常(编译时异常)

Java文件 –(javac命令，可能产生编译时异常)> 字节码文件 –(java命令，可能产生运行时异常)> 运行结果

* 1. 作用
     1. 查询bug的关键参考信息
     2. 作为方法内部的一种特殊返回值，以便通知调用者底层的执行情况 throw new RuntimeException();
  2. 异常处理方式
     1. 虚拟机默认方式：打印异常名称、原因和位置，并结束程序
     2. 自己处理（捕获异常）（选中异常代码ctrl+alt+T自动生成格式）：类似默认方式，但可以继续运行程序

Try{

可能出现的异常代码

} catch(异常类名 变量名) {

对异常的处理

}

特点：若try中某行出现异常，程序就会在该处创建一个对应的异常类对象实例，拿着这个对象实例与catch的()对比，看是否能接收。若能接受，该异常就被捕获，执行catch{}中的代码

四种情况：

* 如果try中没有问题，会执行try中的代码，不会执行catch中的代码
* 如果try中有多个问题，需要多个catch一一对应捕获（若异常存在父子关系，父类一定要写在下面。多态…） 一个catch也可以捕获多个异常，用同样的处理方式 catch(异常类1 | 异常类2 变量名){}
* 如果try中的问题没有catch捕获，会执行虚拟机默认方式
* 如果try中遇到了问题，不会再执行问题下边的代码，会跳到catch
  + 1. 抛出异常

throws：写在方法定义出，表明声明一个异常。告诉调用者，使用本方法可能会有哪些异常

public void function() throws 异常类1, 异常类2 {}

throw：写在方法内，结束方法。手动抛出异常，交给调用者。方法下面的代码不再调用执行

public void function() {throw new RuntimeException();}

* 1. Throwable的异常方法

String getMessage() 返回此Throwable的详细字符串

String toString() 返回此Throwable的简短描述

Void printStackTrace() 打印异常的错误信息（但不停止程序运行）

* 1. 自定义异常（目的：让异常见名知意）

定义异常类 -> 继承 -> 空参构造 -> String的含参构造，打印错误信息

* 1. Finally

Try{}

Catch(){}(可有可无)

Finally{}

一般情况下，都会执行finally里面的代码块

即便try{}和catch{}里面有return，也会执行finally，且先finally再return(finally的return会覆盖它们两个的return)

Finally是为了保证try后一定能执行的代码块，用于关闭连接、关闭文件和释放线程的操作

但finally也可能不能执行：

在finally之前（try里面，或catch里面）执行了System.exit(1)或Runtime.getRuntime.halt(1)或者有无限循环，以及守护线程

1. 多线程

可以让程序同时做多件事情，提高效率

* 1. 两个概念

并发：在同一时刻，有多个指令在单个CPU上交替执行

并行：在同一时刻，有多个指令在多个CPU上同时执行

* 1. 实现方式
     1. 继承Thread类的方式

定义类继承Thread，重写run方法，在该方法里写要执行的代码

创建类对象，调用start()方法，完成多线程操作

* + 1. 实现Runable接口

定义类实现Runable接口，重写run方法。相当于定义多线程的任务

创建类对象和Thread类的对象，将类对象传入Thread对象，调用start()方法

* + 1. 实现Callable接口 可以获取多线程运行的结果

定义类实现Callable接口（泛型），重写call方法（有返回值，表示多线程运行结果）

创建类的对象（表示多线程要执行的任务）和FutureTask对象（作用管理多线程运行的结果）

创建Thread对象，start()启动线程



* 1. 方法

String getName() 返回此线程的名称

Void setName(String name) 设置线程名称

默认名称： Thread-X(X为序号，从0开始)

也可以直接传入参数给构造方法

Static Thread currentThread() 获取当前线程的对象

当JVM虚拟机启动后，会启动多条线程，其中一条叫main

Static void sleep(long time) 休眠指定时间，单位毫秒

哪条线程执行到这个方法，哪条线程就会休眠指定时间（其他线程会抢到执行权）

Void setPriority(int newPriority) 设置线程的优先级

最小1，最大10.代表执行线程的优先概率，但不是绝对的

Final int getPriority() 获取线程的优先级

Final void setDaemon(boolean on) 设置线程为守护线程

当非守护线程执行结束，守护线程就会陆续结束，不需要完整结束

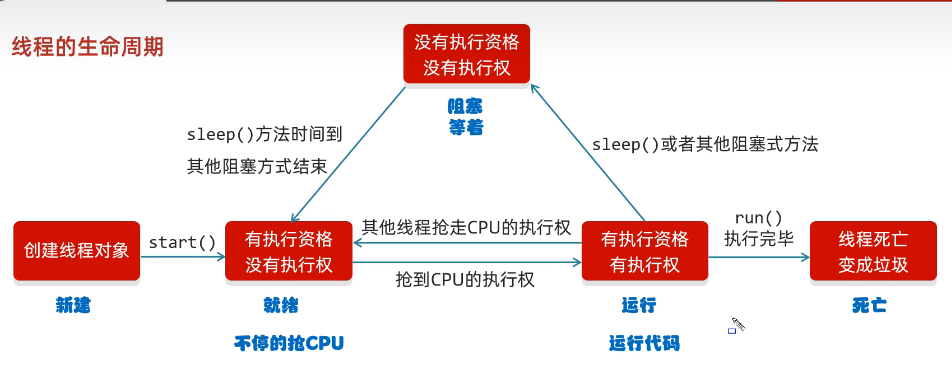
Static void yield() 礼让线程

执行少数几次，CPU就执行其他线程。尽可能保持均衡

Final void join() 插入线程

调用的线程先执行完，再执行当前线程

* 1. 生命周期



* 1. 安全问题

操作共享的数据时，由于每一条语句执行完毕后，都有可能被抢夺执行权，因此可能产生安全问题

解决：

同步代码块

Synchronized(锁，任意对象，但要唯一，如用static) {

操作共享数据的代码

}

锁默认打开，有一个线程进去了，锁自动关闭

只有{}里面的代码全部执行完毕，线程出来，锁自动打开

同步方法

Private synchronized void method() {}

非静态方法，锁为this，静态方法，锁为this.class字节码

Lock锁

目的：清晰看到哪里上锁，哪里释放锁

创建ReentrantLock对象，对象名.lock()上锁，对象名.unlock()释放锁

同样地，锁对象唯一

释放锁操作最好写在finally{}中，确保上锁后一定能释放锁

Tips：尽量不要使用锁的嵌套

* 1. 等待唤醒机制（生产者和消费者）



锁的方法：

Void wait() 当前线程等待，直到被其他线程唤醒

Void notify() 随机唤醒单个线程

Void notifyAll() 唤醒所有线程

* 1. 阻塞队列 继承于Iterable Collection Queue BlockingQueue接口

实现类ArrayBlockingQueue<E> 底层是数组，有界

LinkedBlockingQueue<E> 底层是链表，最大界限时int最大值

方法：

put(Object o) 加入队列 （底层有锁）

take() 取出数据 （底层有锁）

当队列满了，就不能加入，线程等待；当队列为空，就不能取出，线程等待

利用阻塞队列，实现等待唤醒机制：先创建阻塞队列对象（参数为队列容量），将对象传入Cook和Foodier（利用构造），然后start()

* 1. 线程的状态

新建（new） ——> 创建线程对象

就绪（runnable） ——> start()

阻塞（blocked） ——> 无法获得锁对象

等待（waiting） ——> wait()

计时（timed\_waiting） ——> sleep()

结束（terminated） ——> 全部代码运行完毕

* 1. 线程池
     1. 简单使用

创建线程池

ExecutorService pool1 = Executors.newCachedThreadPool()//无上限的线程池

ExecutorService pool2 = Executors.newFixedThreadPool(int nThreads)//有上限

提交任务

pool1.submit(Runnable对象) //当添加的任务大于线程数量时，任务会排队

销毁线程池

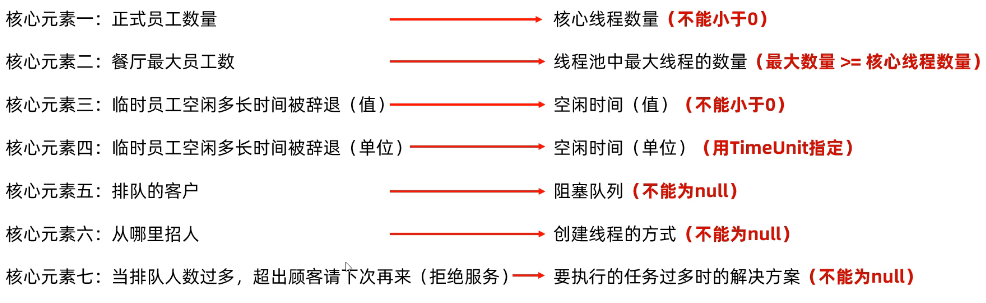
pool1.shutdown();//(一般不销毁)

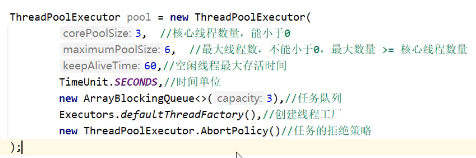
* + 1. 自定义线程

创建一个空的池子，有任务提交时，线程池会创建线程去执行，执行完毕归还线程

不断的提交任务，会有以下三个临界点：

* 当核心线程满时，再提交任务就会排队
* 当核心线程满，队伍满，会创建临时线程
* 当核心线程满，队伍满，临时线程满时，会触发任务拒绝策略



 创建自定义线程池

* + 1. 线程池多大合适？

CPU密集型运算： 最大并行数+1

I/O密集型运算： 最大并行数\*期望CPU利用率\*（CPU计算时间+等待时间）/CPU计算时间

1. File
   1. 构造方法

File(String pathname) //根据文件路径创建文件对象 相对路径：项目名称\\文件名

File(String parent, String child) //根据父路径名称字符串和子路径名称字符串创建文件对象 如D:\\Users\\Docuement\\a.txt，父路径：D:\\Users\\Docuement 子路径：a.txt 底层会进行拼接

File(File parent, String child) //根据父路径对应文件对象和子路径名称字符串创建文件对象

* 1. 成员方法

判断、获取相关：

* boolean isDirectory() //判断此路径名表示的File是否为文件夹
* boolean isFile() //判断此路径名表示的File是否为文件
* boolean exists() //判断此路径名表示的File是否存在
* long length() //返回文件的大小，字节数量
* String getAbsolutePath() //返回文件的绝对路径
* String getPath() //返回定义文件时使用的路径
* String getName() //返回文件的名称，带后缀
* long lastModified() //返回文件的最后修改时间（时间毫秒值）

创建、删除

* boolean createNewFile() //创建一个新的空的文件

如果文件不存在，则创建成功；如果文件已存在，则创建失败返回False。

如果父路径不存在就抛出异常。

如果文件不带后缀名，也能创建成功，创建一个无后缀名的文件

* boolean mkdir() //创建单级文件夹

windows中路径是唯一的，如果当前路径已经存在，则创建失败，返回false

不能创建多级文件夹

* boolean mkdirs() //创建多级文件夹

既可以创建单级，也可以创建多级

* boolean delete() //删除文件、空文件夹

默认删除文件和空文件夹，直接删除不走回收站

删除非空文件夹，返回False

获取并遍历

* File[ ] listFiles() //获取当前路径下所有内容

当调用者Files表示的路径不存在时，返回null

当调用者File表示的路径是文件时，返回null

当调用者File表示的路径是一个空文件夹时，返回一个长度为0的数组

当调用者File表示的路径是一个有内容的文件夹时，将里面所有文件和文件夹的路径放在File数组中返回，包括隐藏文件

当调用者File表示的路径是需要权限才能访问的文件夹时，返回null



1. IO流
   1. 概念

i: input o: output

流：像水流一样传输数据 用于读写数据（本地文件、网络）

按照流向分：输入流，输出流

按照操作文件类型分：字节流（什么都能读），字符流（只能读纯文本文件，用记事本打开没有乱码的）

* 1. 字节输出流

步骤：

1. FileOutputStream f = new FileOutStream(“文件路径”);

//传入File对象也可以

//文件如果不存在，得保证父级路径存在，就会创建新文件

//如果文件已经存在，就会清空原文件

1. f.write(int类型的参数);

//传入整数，但写入的是整数对应的ASCII码字符

1. f.close() //释放资源

写入的另外两种方式：

void write(byte[ ] b) //字节数组数据

void write(byte[ ] b, int off, int len) //字节数组的部分数据，起始索引和数据的个数

续写：new FileOutStream(“文件路径”，True) //True表示打开续写，不清空，默认False

换行：Windows \r\n Linux \n Mac \r

* 1. 字节输入流

步骤：

FileInputStream f = new FileInputStream(“文件路径”); //当文件不存在时，会报错

int n = f.read(); //读取文件内容，返回对应的ASCII码，一次一个字符。读到末尾时，返回-1

f.close() //释放资源

利用while循环配合末尾返回-1循环读取，但要注意每次read后，指针都会后移

一次读取多个字符（即用字符数组）

byte[ ] bytes = new byte[2]

int len = f.rean(bytes) //尽可能装满数组，返回字符个数

fos.write(bytes, 0, len) //从0索引到len-1，组成字符串，避免最后一次读取，没装满数组的情况

* 1. 字符集
     1. ASCII和GBK
* 在计算机中，任意数据都是以二进制的形式来存储的
* 计算机中最小的存储单元是一个字节
* ASCII字符集中，一个英文占一个字节
* 简体中文版Windows，默认使用GBK字符集
* GBK字符集完全兼容ASCII字符集

一个英文占一个字节，二进制第一位是0

一个中文占两个字节，二进制高位字节的第一位是1

* + 1. Unicode字符集

数据采用UTF-8编码格式进行编码

一个英文占一个字节，二进制第一位是0，转成十进制是正数

一个中文占三个字节，二进制第一位是1，第一个字节转成十进制是负数

* + 1. 如何不产生乱码

不要用字节流读取文本文件

编码解码时使用同一个码表，同一个编码方式

* + 1. String类中的编码和解码方法

编码：

* byte[ ] getBytes() //使用默认方式进行编码
* byte[ ] getBytes(String charseName) //使用指定方式进行编码 如GBK

解码：

* String(byte[ ] bytes) //使用默认方式进行解码
* String(byte[ ] bytes, String charsetName) //使用指定方式进行解码
  1. 字符输入流
     + 1. 构造方法

new FileReader(“路径”)； //或是File对象

* + - 1. read()方法

无参 read()

1. 细节：  
   默认一个字节一个字节的读取，如果遇到中文就会一次读取多个
2. 在读取之后，方法底层还会进行解码并转成十进制

最终把这个十进制作为返回值

这个十进制的数据也表示在字符集上的数字

英文：文件里面二进制数据 0110 0001

* + 1. read()方法进行读取，解码并转成十进制97

中文：文件里面的二进制数据 11100110 10110001 10001001

* + 1. read()方法进行读取，解码并转成十进制27721

1. 如果想看到中文，对返回值强转(char)即可

有参 read(char[ ] buffer)

读取数据，解码，强转三步合并了，把强转之后的字符放到数组当中

等于空参的read + 强制类型转换

sout(new String(chars, 0, len)); //打印字符，0到len避免多打

* 1. 字符输出流
     + 1. 构造方法

new FileWriter(“路径”,true)；//或者传入File对象，true即打开续写，默认是关闭续写

* + - 1. write()

可以传入int、String、char[ ]。String可以指定传入的起始位和终位

**代码**：

public class Test {

public static void main(String[] args) throws IOException {

FileWriter fw = new FileWriter("a\\c.txt");

fw.write("我好想你在人群里",0,4);

fw.close();

}

}

* 1. 字符流底层
     1. 输入

关联文件，并在内存创建缓冲区（长度为8192的字节数组）

如果缓冲区没有数据：就从文件中获取数据，装到缓冲区中，每次尽可能装满。如果文件也没数据了，返回-1

如果缓冲区有数据：就从缓冲区读取

* + 1. 输出

内存中也有缓冲区

* 1. 高级流：字节缓冲流

是对字节流的包装，提高数据处理的性能

1. 构造

new BufferedInputStream(InputStream is)

new BufferedOutputStream(OutputSream os)

1. 使用跟字节流一致，只是底层的内存多了缓冲区
   1. 高级流：字符缓冲流

是对字符流的包装，提高数据处理的性能

1. 构造

new BufferedReader(Reader r)

new BufferWriter(Writer w)

1. 特有的方法

readLine() //直接读一整行数据，返回字符串

newline() //跨平台换行

* 1. 高级流：转换流（jdk11以前的用法）

实现字节流->字符流（字符缓冲流）的转换

能让字节流使用字符（缓冲）流的方法

var in = new InputStreamReader(new FileInputStream(“a.txt”), “UTF-8”); //能从字节流转换到字符流，以指定字符集读取

var ou = new OutputStreamWriter(new FileOutputStream(“a.txt”), “GBK”); //能从字节流转换到在字符流，以指定字符集编写

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

jdk11之后：

FileReader fr = new FileReader(“a.txt”, Charset,forname(“UTF-8”));

//实现一样的功能

* 1. 高级流：序列化流与反序列化流（对象操作流）
     1. 序列化流 ObjectOutputStream

方法：final void writeObject(Object obj) //将对象写入文件中

构造：new ObjectOutputStream(OutputStream os)

注意，被操作的对象类应实现Serializable标记型接口，表示类可被序列化

* + 1. 反序列化流 ObjectInputStream

方法：final object readObject() //返回object对象

构造：new ObjectInputStream(InputStream is)

* + 1. 细节

1. 使用序列化流将对象写到文件时，需要让Javabean类实现Serializable接口。否则，会出现NotSerializableException异常
2. 序列化流写到文件中的数据是不能修改的，一旦修改就无法再次读回来了
3. 序列化对象后，修改了Javabean类，再次反序列化会出现问题，抛出InvalidClassException异常

解决：添加 private static final long serialVersionUID = …（IDEA自动加更好）

1. 如果想要让对象的某个成员变量不被序列化，需要加transient关键词修饰
2. 如果想写入多个对象，将多个对象存入ArrayList中，再将ArrayList写入文件，保证文件只有一个对象，避免出现读取异常
   1. 打印流
      1. 字节打印流 PrintStream

构造:PrintStream(OutPutStream/File/String) //三种都可以

(…,boolean autoFlush,Charset charset) //自动刷新和字符集

方法：println() print() printf()

* + 1. 字符打印流 PrintWriter

构造:PrintWriter(OutPutStream/File/String)

(…,boolean autoFlush,Charset charset) //由于有缓冲区，需要开启自动刷新

方法：write() println() print() printf()

* 1. 解/压缩流
     1. 解压流

先创建一个File对象，指向xxx.zip文件，和另一个File对象，指向目的地文件。

再创建ZipInputStream对象，读取zip文件的内容getNextEntry()，用while循环，停止条件为读到null。

如果nextEntry.isDirctory()读到文件夹，就在目的地创建文件夹new File(目的地File对象, nextEntry.toString())

如果读到文件，用while循环配合字节输出流拷贝，停止条件为zip.read() == -1

操作完一个zip内容对象，就closeEntry();zip结束就close()

代码：

public class Test {

public static void main(String... args) throws IOException {

File zip = new File("laji.zip"); //要解压的文件

File dest = new File("lajistation\\laji"); //目的地

unzip(zip,dest);

}

private static void unzip(File zip, File dest) throws IOException {

ZipInputStream zos = new ZipInputStream(new FileInputStream(zip));

ZipEntry nextEntry = null;

while ((nextEntry = zos.getNextEntry()) != null) {

System.out.println(nextEntry);

if (nextEntry.isDirectory()) {

File file = new File(dest,nextEntry.toString());

file.mkdirs();

System.out.println("??");

}

else {

FileOutputStream fos = new FileOutputStream(new File(dest,nextEntry.toString()));

int ch = 0;

while ((ch = zos.read()) != -1) {

fos.write(ch);

}

fos.close();

}

zos.closeEntry();

}

zos.close();

}

}

* + 1. 压缩流

先创建两个File对象，分别指向被压缩的文件和压缩的地址

再创建ZipOutputStream对象，生成压缩包，需要new ZipOutputStream(new FileOutputStream(new Filel(目的地地址,压缩包全名称)))

然后创建压缩包里的文件对象var ze = new ZipEntry(“文件名”)

put文件进压缩包里 zip.putNextEntry(ze)

用字节输入流和zip.read()将字节文件拷贝进ze里面

最后close

代码：

public class Test {

public static void main(String... args) throws IOException {

File zip = new File("lajistation\\");

File file = new File("lajistation\\laji.txt");

tozip(file,zip);

}

public static void tozip(File file,File zip) throws IOException {

ZipOutputStream zis = new ZipOutputStream(new FileOutputStream(new File(zip,"laji.zip")));

ZipEntry ze = new ZipEntry("laji.txt");

zis.putNextEntry(ze);

FileInputStream fos = new FileInputStream(file);

int ch = 0;

while ((ch = fos.read()) != -1) {

zis.write(ch);

}

fos.close();

zis.closeEntry();

zis.close();

}

}