## String.format

描述:格式化；

举例: **%[argument\_index$][flags][****width][.precision]****conversion**

**argument\_index:参数位置；**

**flags：- 左对齐；**

**width：字符宽度；**

**precision：保留字符数**

**conversion:类型，如果为d ，则flags可以用0 填充，默认都是空格**

String s= String.format("my name is %s,age is %s","zs","30");

String e= String.format("my name is %1$.3s,age is %2$s","zsddd","30")-🡪

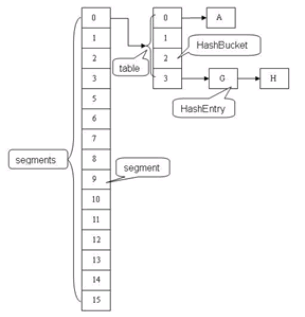
my name is zsd,age is 30

String.intern():调用”ab”.intern()方法的时候会返回”ab”，但是这个方法会首先检查字符串池中是否有”ab”这个字符串，如果存在则返回这个字符串的引用，否则就将这个字符串添加到字符串池中，然会返回这个字符串的引用。

## ConcurrentHashMap

描述:

ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁ReentrantLock，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁.



putIfAbsent:如果已存在key不会添加；不存在key会添加数据。

## AtomicReference

描述: 对"对象"进行原子操作;

compareAndSet(K,V) 返回 Boolean;如果实际值与期望值K相同，则更新为V；

get()获取值.

## Idea 快捷键

选中代码整体右移 tab；选中代码整体左移: shift+tab

## ReentrantReadWriteLock

描述：锁，多线程使用；对同一个对象加锁。

示例:

ReentrantReadWriteLock readWriteLock = new ReentrantReadWriteLock();

ReentrantReadWriteLock.ReadLock readLock = readWriteLock.readLock();

ReentrantReadWriteLock.WriteLock writeLock = readWriteLock.writeLock();

ReentrantReadWriteLock是Lock的另一种实现方式，我们已经知道了ReentrantLock是一个排他锁，同一时间只允许一个线程访问，而ReentrantReadWriteLock允许多个读线程同时访问，但不允许写线程和读线程、写线程和写线程同时访问。相对于排他锁，提高了并发性。在实际应用中，大部分情况下对共享数据（如缓存）的访问都是读操作远多于写操作，这时ReentrantReadWriteLock能够提供比排他锁更好的并发性和吞吐量.

支持公平和非公平的获取锁的方式;

公平锁即尽量以请求锁的顺序来获取锁。比如同是有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该所，这种就是公平锁。

　　非公平锁即无法保证锁的获取是按照请求锁的顺序进行的。这样就可能导致某个或者一些线程永远获取不到锁。

　　在Java中，synchronized就是非公平锁，它无法保证等待的线程获取锁的顺序。

而对于ReentrantLock和ReentrantReadWriteLock，它默认情况下是非公平锁，但是可以设置为公平锁。设置方法如下：ReentrantReadWriteLock lock = new ReentrantReadWriteLock(true);

1.可重入(Reentrant)锁

如果锁具备可重入性，则称作为可重入锁。像synchronized和 ReentrantLock都是可重入锁，可重入性在我看来实际上表明了锁的分配机制：基于线程的分配，而不是基于方法调用的分配。举个简单的例子，当一 个线程执行到某个synchronized方法时，比如说method1，而在method1中会调用另外一个synchronized方法 method2，此时线程不必重新去申请锁，而是可以直接执行方法method2

## Consumer<T>

描述:按照输入的lambda表达式，修改T的属性

方法：void accept(T t) 修改T属性；

Consumer<T> andThen(Consumer<? super T> after):

先按照自己的lambda对T修改；再用after对T进行修改

## Supplier<T>

描述：获取lambda创建的对象；

方法：T get();

Supplier<TestSupplier> sup= TestSupplier::new;

sup.get().

## Condition

描述：线程等待状态

Lock lock = new ReentrantLock();

Condition condition = lock.newCondition();创建

Condition.await() 线程进入挂起状态；

Condition.signal()唤醒线程 ；

Condition 必须在lock.lock 和lock.unlock之间使用。

## Callable和Future

Callable：V call() throws Exception;

用于执行线程时返回结果，与Future搭配使用；

ExecutorService excuter；

Future<T> future=Excuter.submit(callable);

等待线程执行的结果；

Future:方法：

boolean cancel(boolean mayInterruptIfRunning); 取消任务

boolean isCancelled(); 是否取消；

boolean isDone(); 是否完成任务；

V get() throws InterruptedException, ExecutionException; 获取结果，会一直阻塞

## File and Path

描述:文件操作;

方法: Path path = Paths.get("./test"); 创建路径；

path.isAbsolute() 是否是绝对路径；

Files.createDirectories(path); 创建文件夹；

path.toRealPath() 返回绝对路径；

path.resolve("1" path) 路径合并 ./test/1 如果 path为绝对路径则直接返回path；

path.getNameCount() 返回目录层级；

path.relativize(path1) 返回path1 相对path的相对路径；

path2.subpath(1,2) 截取目录层级的名称 前闭后开;

getAbsolutePath():E:\workspace\Test\.\test.txt

返回的是定义时的路径对应的相对路径，但不会处理“.”和“..”的情况

getCanonicalPath():E:\workspace\Test\test.txt

返回的是规范化的绝对路径，相当于将getAbsolutePath()中的“.”和 “..”解析成对应的正确的路径;

File file = new File(".\\test.txt");

## BlockQueue

描述：高并发队列；有ArrayBlockQueue和LinkedBlockQueue;

通用方法:

插入方法：

add(E e) : 添加成功返回true，失败抛IllegalStateException异常

offer(E e) : 成功返回 true，如果此队列已满，则返回 false。

put(E e) :将元素插入此队列的尾部，如果该队列已满，则一直阻塞

删除方法:

remove(Object o) :移除指定元素,成功返回true，失败返回false

poll() : 获取并移除此队列的头元素，若队列为空，则返回 null

take()：获取并移除此队列头元素，若没有元素则一直阻塞。

检查方法

element() ：获取但不移除此队列的头元素，没有元素则抛异常

peek() :获取但不移除此队列的头；若队列为空，则返回 null

ArrayBlockingQueue和LinkedBlockingQueue的区别：

1. 队列中锁的实现不同

    ArrayBlockingQueue实现的队列中的锁是没有分离的，即生产和消费用的是同一个锁；

    LinkedBlockingQueue实现的队列中的锁是分离的，即生产用的是putLock，消费是takeLock

2. 在生产或消费时操作不同

    ArrayBlockingQueue实现的队列中在生产和消费的时候，是直接将枚举对象插入或移除的；

    LinkedBlockingQueue实现的队列中在生产和消费的时候，需要把枚举对象转换为Node<E>进行插入或移除，会影响性能

3. 队列大小初始化方式不同

    ArrayBlockingQueue实现的队列中必须指定队列的大小；

    LinkedBlockingQueue实现的队列中可以不指定队列的大小，但是默认是Integer.MAX\_VALUE

## Lambda

@FunctionalInterface（可以不写）

public interface MyLambda {

int GetData(int x,int y);

}

Lambda 表现形式；

调用时可以书写lambda；(a,b) - > a+b

public int getData(MyLambda myLambda){

return myLambda.GetData(x,y);

}

## Manifest

描述:每个jar内都有这个文件，用于描述jar信息；

Attributes attributes = manifest.getMainAttributes(); 获取主属性

Map<String,Attributes> entry = manifest.getEntries(); 获取其他属性；

属性格式:key: test 冒号和内容之间要有空格;

主属性和其他属性之间要有一个空行;

属性结尾要有一个空行；

其他属性格式:

name: zs

key1: abc

key2: ddd

## @RequestMapping

接收数组型参数:

接收端:@RequestMapping(“list”) List<String> list

发送端:http://127.0.0.1:8080/nifi?list=1,2,3 以逗号分隔

## @XmlRootElement

用于生成XML文件；

@**XmlRootElement：根元素；**

**@XmlElement:子项元素；**

**@XmlType:用于子项内元素排序:@XmlType(propOrder={“name”,”age”})**

## BuffereInputStream BufferedOutputStream

buffin .mark(int readlimit); 标记开始读字节的位子 参数是 限制在什么范围内 标记有效果

buffin .reset() 重新回到Make时候的位子 读取字节

当超过buffereinputstream 里的缓存空间 就要判断mark限定的范围内 如果不超过就mark有效

## DataOutputStream

数据输出流允许应用程序以与机器无关方式将Java基本数据类型写到底层输出流;

void writeBytes(String s) 将字符串按字节顺序写出到基础输出流中;

void writeUTF(String str)

以与机器无关方式使用 UTF-8 修改版编码将一个字符串写入基础;

void write(byte[] b, int off, int len)

将指定 byte 数组中从偏移量 off 开始的 len 个字节写入基础输出;

void writeDouble(double v)

使用 Double 类中的 doubleToLongBits 方法将 double 参数转换为一个 long 值，

          然后将该 long 值以 8-byte 值形式写入基础输出流中，先写入高字节。

void writeChar(int v)

将一个 char 值以 2-byte 值形式写入基础输出流中，先写入高字节。

void writeBoolean(boolean v)

将一个 boolean 值以 1-byte 值形式写入基础输出流

## ExecutorService

描述：线程池；

线程关闭: ExecutorService.Shutdown 只是发出通知，线程关闭，不会接受新的任务，但是会等线程全部执行完；

ExecutorService. awaitTermination(time) 等待时间，设一个值，等线程执行完；

ExecutorService. shutdownNow（）立刻停止所有线程工作，但是线程会抛出

InterruptedException异常，线程内需要进行捕获。

生命周期：守护进程（Daemon）是运行在后台的一种特殊进程。它独立于控制终端并且  
周期性地执行某种任务或等待处理某些发生的事件;

User和Daemon两者几乎没有区别，唯一的不同之处就在于虚拟机的离开：如果 User Thread已经全部退出运行了，只剩下Daemon Thread存在了，虚拟机也就退出了。 因为没有了被守护者，Daemon也就没有工作可做了，也就没有继续运行程序的必要了