## 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")-🡪

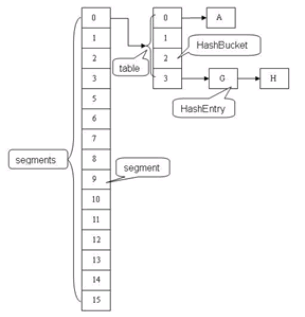
my name is zsd,age is 30

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;

## 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

## 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);

}