Java学习

数组学习

1. 静态新建一个List必须这样： List list = Arrays.asList(16, 17, 18, 19);
2. 这样新建一个数组 int[] x = new int[10];
3. 怎么把一个list 转为ArrayList

   List list = Arrays.asList("1", "2", "3", "4"); ArrayList arrayList = new ArrayList<>(list);
4. remove() removeFirst() poll()移除并返回 被删除的对象 但是 如果LinkeList 为空前两个会报错， poll()返回null
5. getFirst() element() peek() 类似上面的用法
6. 可以将LinkedList作为栈来用
7. 其实Set就是Collection 8.

```
public class PriorityQueueDemo {
      public static void printQ(Queue queue){
         while (queue.peek() != null)
         System.out.print(queue.remove()+ " ");
         System.out.println();
    }
        public static void main(String args[]){
         PriorityQueue priorityQueue =   new PriorityQueue<Integer>();
        Random random = new Random(47);
        for(int i = 0; i < 10; i++)
        priorityQueue.offer(random.nextInt(i + 10));
        printQ(priorityQueue);

        List<Integer> integerList = Arrays.asList(2,321, 3213, 2, 33, 44423, 23, 4,
34);

        priorityQueue = new PriorityQueue(integerList);
        printQ(priorityQueue);

        priorityQueue = new PriorityQueue(integerList.size(),
Collections.reverseOrder());
        priorityQueue.addAll(integerList);
        printQ(priorityQueue);

        String fact = "EDUCATION SHOULD ESCHEW OBFUSCATION";
        List<String> strings = Arrays.asList(fact.split(" "));
        PriorityQueue<String> stringPQueuet = new PriorityQueue<>(strings);
        printQ(stringPQueuet);

        stringPQueuet = new PriorityQueue<>(strings.size(),
Collections.reverseOrder());
        printQ(stringPQueuet);
```

```
        Set<Character> charSet = new HashSet<>();
        for(char c : fact.toCharArray()){
            charSet.add(c);
        }
        PriorityQueue     <Character> charPQ = new PriorityQueue<>(charSet);
        printQ(charPQ);
    }
```

9. Map的foreach循环2018/3/28 15:40:14

```
    public class EnvironmentVariables {
    public static void main(String [] args){
        for(Map.Entry entry: System.getenv().entrySet()){
            System.out.println(entry.getKey() + ": " +  entry.getValue());
        }
    }
}
```

10. queue.offer()将元素加到队尾

异常的学习

10. RuntimeException JVM自动捕获，不用显示的捕获，不捕获的RuntimeException会直接到达main（），在程序推出前将调用printStackTrace() RuntimeException()是编程错误，数组超限了，传过来空值都会引起。
11. finnlay在Java中的作用主要是将除内存外的资源恢复到它们最初的状态。
12. 接口中的方法没有声明异常，子类声明了，会编译报错。如果接口或者父类定义的方法声明了异常，子类的方法可以不用声明
13. 在创建需要清理的对象后，立即进入一个try-finally语块
14. 查找异常把那个不需要抛出的异常同声明的异常完全匹配

String学习

15. StringBuilder 比String更高效

类型信息学习

1. .class比Class.forName()更好一点，后者一声明直接就初始化，.class更慢一点，更加安全，在编译的时候就会受到检查。 Class Name = Shape.class
2. static finnal 是常量，不需要初始化就可以被读取
3. java可以不用显示的向上转型的赋值操作
4. 用isInstance（）可以判断是不是这个类或者是不是这个类的子类 但getClass（）返回的类就是单独的类，没有继承的信息。
5. 动态反射 调用Proxy.newProxyInstance()，要传递三个参数， 类加载器，一个希望被实现的接口列表，一个InvocHandler的实现 下面是一个具体的例子

```
package Java.RTTI;
        import java.lang.reflect.InvocationHandler;
        import java.lang.reflect.Method;
        import java.lang.reflect.Proxy;
```

```
    interface Interface {
        void doSomething();
        void somethingElse(String args);
    }
    class RealObject implements  Interface {

        @Override
        public void doSomething() {
            System.out.println("doSomething");
        }

        @Override
        public void somethingElse(String args) {
            System.out.println("somethingElse" + args);
        }
    }
    class SimpleProxy implements Interface{

        private Interface proxied;
        public SimpleProxy(Interface proxied) {
            this.proxied = proxied;
        }
        @Override
        public void doSomething() {
            System.out.println("SimpleProxy doSomething");
            proxied.doSomething();
        }

        @Override
        public void somethingElse(String args) {
            System.out.println("SimpleProxy somethingElse" + args);
            proxied.somethingElse(args);
        }
    }
    class DynamicProxyHandler implements InvocationHandler {
        private Object proxied;
        public  DynamicProxyHandler(Object o) {
            this.proxied = o;
        }
        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
            System.out.println("****** proxy : " + proxy.getClass()
                + ". method: " + method + ", args: " + args);
            if(args != null) {
                for (Object arg : args)
                    System.out.println(" " + arg);
```

```
            }
                return method.invoke(proxied, args);
        }
    }
    public class SimpleProxyDemo {
        public static void consumer(Interface inter) {
            inter.doSomething();
            inter.somethingElse("bonobo");
        }

        public static void main(String[] args) {
    /*        consumer(new RealObject());
            consumer(new SimpleProxy(new RealObject()));*/
            RealObject realObject = new RealObject();
            consumer(realObject);
            Interface proxy = (Interface) Proxy.newProxyInstance(
                    Interface.class.getClassLoader(),
                    new Class[] {Interface.class},
                    new DynamicProxyHandler(realObject)
            );
         consumer(proxy);
        }
    }
```

6. 所有的类都是在对其第一次使用时，动态的加载到VM中的。当程序创建第一个对类的静态成员引用时，就会加载这个类，这也证明构造器也是类的静态方法。
7. Class<?> 优于Class
8. xxClass.newInstance()返回xxClass的确切类型，不能用模糊的类来接收

```
        FancyToy ftClass = new FancyToy();
        Class<? super FancyToy> up = ftClass.newInstance()
```

9. 如何访问不是public的函数

```
    Method g = a.getClass().getDeclaredMethod(methodName);
        g.setAccessible(true);
```

2018/4/10 15:16:19 更新

泛型学习

1. 泛型类在创建对象时就得指定类型参数的值，使用泛型方法时不需要，java有自动打包机制
2. 类型推断只对赋值操作有效，对作为参数传递给另一个函数会报错。
3. 在泛型方法中，可以显式地指明类型。要显式地指明类型，必须在点操作符与方法名之间插入尖括号，然后把类型置于尖括号内。如果是在定义该方法的类的内部，必须在定义该方法的类的内部，必须在点操作符之前使用this关键字，如果是使用static的方法，必须在点操作符之前加上类名。不过只有在编写非赋值语句时才需要这样额外的说明。
4. 一个例子

```
package Java.Generic;
```

```
class CountedObject{
    private static int count = 0;
    private final int id = count++;
    public int id() { return  id;}

    @Override
    public String toString() {
        return "CountObject " + id;
    }
}
public class BasicGenerator<T>  implements Generator<T>{

    public Class<T> type;
    public BasicGenerator(Class<T> tClass) {
        this.type = tClass;
    }
    @Override
    public T next() {
        try {
            return type.newInstance();
        }catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    public static <T>BasicGenerator<T> create(Class<T> type) {
        return new BasicGenerator<T>(type);
    }
    public static void main(String[] args) throws InstantiationException,
IllegalAccessException {
        Generator<CountedObject> generator =
BasicGenerator.create(CountedObject.class);
        for(int i = 0; i < 5; i++)
            System.out.println(generator.next());
    }
}
```

5. java使用泛型，但不知道这个泛型具体是什么。 在泛型代码内部，无法获得任何有关泛型参数类型的信息 是因为JAVA把任何具体的类型信息都擦出了。
6. @SuppreWarings("Unchecked") 注解放在产生警告的方法上，而不是整个类上
7. java在get()返回值会进行转型，将Object类转为声明的类,对传递进来的值进行额外的编译检查，并插入对传递出去的值的转型
8. 类型标签可以匹配泛型参数, 所谓类型标签就是在类内声明一个Classs type, 并给其赋值
9. 要创建一个泛型数组，赋值时最好用Array.newInstance()
10. Java建议使用显示的工厂，限制其类型，使得只能接受实现了这个工厂的类
11. 如果实际运行的数组类型是Apple[]，在编译期间可以将Fruit类及其子类放入这个数组中，但是在运行期间不能向其添加非Apple类的数据
12. 泛型的目标之一，就是将运行时发现插入错误的类型提前到编译时就能发现。
13. <? extends T>能使用get()一个泛型对象，但是不能set(), 因为set()参数意味着它可能是任何类型的事物，编译器无法验证
14. <? super T> 可以使用set() 方法，因为编译器知道这个对象是 T或者T的子类，但是get方法不能用T t = get()； 因为持有的类型可能是任何超类型，所以可以用Object o = get()来接受

15. <?>意味着持有某种特定的类型，所以也只能用get()方法，只能用Object来接收
16. 任何带有泛型类型参数的转型或instanceof不会产生任何效果
17. 古怪的循环泛型 CRG Class Subtype extends BasicHolder< Subtype > 基类用导出类代替其参数，这就让基类变成了一个模板，既能get确定的子类，又能set确定的子类、
18. 自限定 SelfBounded<T extends SelfBounded< T > >
19. 在编译期就检查出容器内插入的类不适合 Collections.checkedList(new ArrayList(), T.class);
20. 混形，就是声明几个接口，然后实现这几个接口，定义一个类，然后继承这几个接口，在这个类中实例话这几个接口的具体实现类，声明方法，调用接口的方法，这样就把几个类混合成为一个类了
21.

数组学习

1.对象数组保存的是引用，基本类型数组直接保存基本类型的值 2.int[] a = new int[10] int[] a = {1, 2, 3}

3. Arrays.fill(a, ?)相当于C++中的memset()给数组批量赋值
4. equal比较的是内容，deepEqual比较多维数组
5. Arrays.sort(a)升序排序， Arrays.sort(a, Collections.reverseOrder()), 与前面相反的排序
6. 可以之定义类实现Comparator接口，然后sort可以安装这个规则来进行排序

```
class TComparator implements Comparator< T > {
    public int compare(T t1, T t2) {
        return t1 < t2 ? -1 : (t1 = t2 ? 0 : 1);
    }
}
Arrays.sort(a, TComparator);
```

7. System.arraycopy()用这个函数复制数组比for循环快很多
8. Array.equals()基于内容的比较
9. Arrays.binarySearch()可以对排好序的数组进行快速查找 ##容器学习
10. 填充容器 Collections.nCopies(), Arrays.fill()只能替换已经存在的元素

并发学习

1. 什么时候使用synchronized
    1. 多线程时
    2. 多个线程同时访问同一个资源时
    3. 实例创建之后状态可能变化
    4. 需要确保线程安全时
2. 为什么使用synchronized 会耗费时间
    1. 锁住资源需要花费时间
    2. 想要获取资源的线程等待，引起堵塞。
3. HashTable 和currentHashMap都是线程安全的，也就是说不会发生两个线程同时修改，因为前者采用synchroinized，所以性能低
4. 对long 和double的操作不是原子操作
5. Semaphore 类可以定义 资源个数
6. semaphore.acquire 确保可用资源，资源减一

7. semaphore.release 释放该资源，可用资源加一

```
package Runtime.NO1;

    import java.util.Random;
    import java.util.concurrent.Semaphore;

    class Log{
        public static void println(String s){
            System.out.println(Thread.currentThread().getName() + ": " + s);
        }
    }
    class BoundedResource {
        private final Semaphore semaphore;
        private final int permits;
        private final static Random random = new Random(47);
        public BoundedResource(int permits){
            this.semaphore = new Semaphore(permits);
            this.permits = permits;
        }
        public void use() throws InterruptedException{
            semaphore.acquire();
            try {
                doUse();
            } finally {
                semaphore.release();
            }
        }

        private void doUse() throws InterruptedException{
            Log.println("Begin : used = " + (permits - semaphore.availablePermits()));
            Thread.sleep(random.nextInt(500));
            Log.println("END:  used = " + (permits - semaphore.availablePermits()));
        }
    }
    public class UserThread extends Thread {
    //    private final Gate gate;
    //    private final String name;
    //    private final String myaddress;
    //    public UserThread(Gate gate, String name, String myaddress){
    //        this.gate = gate;
    //        this.name = name;
    //        this.myaddress = myaddress;
    //    }
    /*    public void run(){
            System.out.println(name + " BEGIN");
            while (true){
                gate.pass(name, myaddress);
            }
        }*/
        private final static Random random = new Random(47);
        private final BoundedResource resource;
        public UserThread(BoundedResource boundedResource){
```

```
            this.resource = boundedResource;
        }
        public void run() {
            try {
                while (true) {
                    resource.use();
                    Thread.sleep(random.nextInt(3000));
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        public static void main(String[] args){
            BoundedResource boundedResource = new BoundedResource(3);
            for (int i = 0; i < 3; i++) {
                new UserThread(boundedResource).start();
            }
        }
    }
```

6. 什么时候使用 想破坏也破坏不了这个模式
    1. 实例创建之后不在发生变化，使用final声明，并且没有setter方法
    2. 实例是共享的，而且会被频繁访问
7. final只能被赋值一次
8. static final只能在定义时赋值或者在static{}代码块中赋值
9. ArrayList 类在被多个线程同时读写时会失去安全性，可用使用`Collections.synachroizedList(new ArrayList<>())`，同时在读的时候加上

```
synchroized
            package Runtime.NO2;
```

```
        import java.util.ArrayList;
        import java.util.Collections;
        import java.util.List;
        class readrThread extends Thread {
        private List<Integer> list;
        public readrThread(List list){
            this.list = list;
        }

        @Override
        public void run() {
            while (true) {
                    synchronized (list) {
                        for (int n : list){
                            System.out.println(n);
                        }
                    }
            }
        }
```

```
    }

    public class WriterThread  extends Thread{
        private final List<Integer> list;
        public WriterThread(List list){
            this.list = list;
        }

        @Override
        public void run() {
            for (int i = 0; true; i++) {
                list.add(i);
                list.remove(0);
                //System.out.println(list.get(i));
            }
        }
        public static void main(String[] args){
           final List<Integer> list = Collections.synchronizedList( new ArrayList<>()
);
            WriterThread writerThread = new WriterThread(list);
            readrThread readrThread = new readrThread(list);
            writerThread.start();
            readrThread.start();
        }
    }
```

10. 上面的也可以使用CopyOnWriteArrayList()，适用于读次数多，写较少的情况
11. wait()使进程进入等待序列，notify()激活进程
12. 用while循环，不能用if

- 存在循环
- 存在条件检查
- 因为某种原因等待

13. LinkedBlockingQueue.take() 取队列首元素，如果为空就wait() .put()将元素放置到队尾take和put互斥
14. 什么时候使用Blaking(不用不执行模式)

- 有的进程不需要进行
- 不需要等待守护成立条件
- 守护条件仅在第一次成立

15. synchronized 中没有超时，也不能中断。
16. java.util.concurrent中超时可以通过异常通知超时，也可以通过返回值通知超时
17. 怎么写一个超时判断

```
    public synchronized void execute() throws   InterruptedException, TimeoutException
{
        long start = System.currentTimeMillis();
        while (!ready) {
            long now = System.currentTimeMillis();
            long rest = timeout - (now - start);
            if(rest <= 0) {
                throw new TimeoutException("now - start = " + (now - start) + ",
timeout = " + timeout);
            }
            wait(rest);
        }
        doExecute();
    }
```

18. interrupt 方法可以中断sleep方法，抛出InterruptedException也不必获得要中断进程的锁
19. interrupt 方法中断wait方法，会在被中断进程获得锁后才抛出异常
20. 执行wait(), notify 和 notifyall 的进程必须要获取实例的锁，执行interrupt不需要获取被中断进程的锁
21. isInterruped方法，检查是否处于中断状态
22. interruped，检查并清除中断状态
23. Java的线程机制是抢占式的
24. java.util.concurrent中的Executor更常用，管理Thread()对象。shutdown()防止新任务被提交给Executor()
25. SingleThreadExecutor用于希望线程长期运行的任务
26. Runnabke不返回值，Callable返回值
27. 一个线程是后台线程，那么由这个线程创建的都会被自动设置成后台线程
28. 当非后台线程结束时，后台进程立即被终止，所有finally执行不了
29. 在一个线程进行中join，就是让join的线程运行，等待join的运行完后，本线程重新获得CPU运行
30. setDaemon()设置是否是后台运行线程
31. 异常不容易捕获，需要特殊处理
32. 两种线程的异常处理方式

```
class ExceptionThread implements Runnable {


    @Override
    public void run() {
        Thread thread = Thread.currentThread();
        System.out.println(" run by " + thread);
        System.out.println("eh =+ " + thread.getUncaughtExceptionHandler());
        throw new RuntimeException();
    }
}
class MyUncaughtExceptionHandler implements Thread.UncaughtExceptionHandler {

    @Override
    public void uncaughtException(Thread t, Throwable e) {
        System.out.println("caught " + e);
    }
}
class HandlerThreadFactory implements ThreadFactory {
```

```
    @Override
    public Thread newThread(Runnable r) {
        System.out.println(this + " creating new Thread");
        Thread thread = new Thread(r);
        System.out.println("created " + thread);
        thread.setUncaughtExceptionHandler(new MyUncaughtExceptionHandler());
        System.out.println(
                "eh = " + thread.getUncaughtExceptionHandler()
        );
        return thread;
    }
}
public class CaptureUncaughtException {
    public static void main(String[] args) {
        ExecutorService service = Executors.newCachedThreadPool(
                new HandlerThreadFactory());
        service.execute(new ExceptionThread());
    }
}
```

第二种

```
public class SettingDefaultHandler {
    public static void main(String[] args) {
        Thread.setDefaultUncaughtExceptionHandler(
                new MyUncaughtExceptionHandler());
        ExecutorService service = Executors.newCachedThreadPool();
        service.execute(new ExceptionThread());
    }
}
```

1. 在使用并发时，将对象的访问域设置为private 非常重要
2. 大部分都用synchronized，只有在一些需要特殊处理的时候采用lock 和unlock
3. ReentrantLock lock = new ReentrantLock()
4. 除了long 和double都被保证操作的原子性
5. 使用lock unlock时， 下面是一个例子

```
    public int next() {
            lock.lock();
        try {
            ++currentValue;
            ++currentValue;
            return currentValue;
        } finally {
            lock.unlock();
        }
    }
```

return语句一定要放在try中，确保unlock不会太早发生。

6. 尝试获取锁，可能会产生异常，或者获取一段时间锁，需要显示的Lock
7. 显示的Lock给你更细粒度的控制力，可以用在遍历链表的节点的传递加锁。

8. 使用volatile唯一安全的情况是类中只有一个可变域，其他都不是很安全
9. volatile变量会立刻同步到内存，保证了变量的可见性，实际上synchronized也保证了可见性。
10. volatile 并不能使操作变成原子操作。
11. 模板方法模式：把变化封装在代码里，就是把不变的放在抽象类，变得定义新的子类
12. 整个方法被sychronized 访问效率不如 方法内同步控制块 访问的快，因为整个方法被加锁，而同步控制块 一部分被加锁，后者加锁的时间短。
13. 如果一个线程获取了对象的synchronized锁，那该对象其他的synchronized方法不能被调用了。
14. 一个对象如果有两种不同的同步锁，可以被两个不同的线程同时访问不同的被同步对象
15. TreadLocal 通常当作静态域存储 为使用相同变量的每个不同的线程都创建不同的储存

```
package Runtime.Thread;

import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

class Accessor implements Runnable{

    private final int id;

    Accessor(int id) {
        this.id = id;
    }

    @Override
    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            ThreadLocalVariableHolder.increment();
            System.out.println(this);
            Thread.yield();
        }
    }

    @Override
    public String toString() {
        return "#" + id + ": " + ThreadLocalVariableHolder.get();
    }
}
public class ThreadLocalVariableHolder {
    private static ThreadLocal<Integer> value = new ThreadLocal<Integer>() {
        private Random rand = new Random(47);
        protected synchronized Integer initialValue() {
            return rand.nextInt(10000);
        }
    };
    public static void increment() {
        value.set(value.get() + 1);
    }
    public static int get() { return value.get();}

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

```
        ExecutorService service = Executors.newCachedThreadPool();
        for (int i = 0; i < 5; i++)
            service.execute(new Accessor(i));
        TimeUnit.SECONDS.sleep(3);
        service.shutdown();
    }
}
```

16. 能中断调用sleep()的线程，但是不能中断试图获取synchroized锁或者试图执行IO的线程,可以通过关闭底层资源的形式，让其产生中断
17.

```
        Future<?> f = executor.submit(runnable);//submit提交可以有返回值
        TimeUnit.MILLISECONDS.sleep(100);
        f.cancel(true);//cancel可以提交中断进程
```

18. ReentrantLock 可以设置 lock.interruptibly() 运行被中断
19. synchronized(class) synchronized(this) - > 线程各自获取monitor，不会有等待． synchronized(this) synchronized(this) - > 如果不同线程监视同一个实例对象，就会等待，如果不同的实例，不会等待． synchronized(class) synchronized(class) - > 如果不同线程监视同一个实例或者不同的实例对象，都会等待．
20. 把run方法放到一个大的try语句块里，可以保证run结束
21. PipedReader 与普通的IO的区别在于 它可以被中断
22. 发生死锁的四个条件
    1. 互斥条件
    2. 拥有一个资源并且等待另一个资源
    3. 不可抢占
    4. 循环等待
23. CountDownLatch用来同步一个或多个任务，强制它们等待其他任务执行完成的一组操作
24. CountDownLatch 的用法， 等待任务执行await() 等待 被等待任务 完成， 被等待的任务 完成一次调用一次countDown()，让CountDownLatch的值减一，等到为0的时候，开始执行await()的线程。
25. CyclicBarrier 的用法， 设置 个数，每次await()计数减一，当容量为0时，并发执行，在此之前都不执行。
26. 区别在于CountDownLatch 只能用一次， CyclicBarrier可以用多次。当计数值为0时，自动开始所有等待的任务。 package Runtime.Thread; import java.util.ArrayList; import java.util.List; import java.util.Random; import java.util.concurrent.*; class Horse implements Runnable{

```
    private static int counter = 0;
    private final int id = counter++;
    private int strides = 0;
    private static Random random = new Random(47);
    private static CyclicBarrier barrier;
    public Horse(CyclicBarrier b) { barrier = b;}
    public synchronized int getStrides() {return  strides;}
    @Override
    public void run() {
        try{
            while (!Thread.interrupted()) {
```

```
                synchronized (this) {
                    strides += random.nextInt(3);
                }
                barrier.await();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
    public String toString() { return "Horse " + id + " ";}
    public String tracks() {
        StringBuilder builder = new StringBuilder();
        for (int i = 0; i < getStrides(); i++)
            builder.append("*");
        builder.append(id);
        return builder.toString();
    }
}
public class HorseRace {
    static final int FINISH_LINE = 75;
    private List<Horse> horses = new ArrayList<Horse>();
    private ExecutorService exec = Executors.newCachedThreadPool();
    private CyclicBarrier barrier;
    public HorseRace(int hourses, final int pause) {
        barrier = new CyclicBarrier(hourses, new Runnable() {
            @Override
            public void run() {
                StringBuilder s = new StringBuilder();
                for(int i = 0; i < FINISH_LINE; i++)
                    s.append("=");
                System.out.println(s);
                for (Horse horse : horses)
                    System.out.println(horse.tracks());
                for (Horse horse : horses)
                    if(horse.getStrides() >= FINISH_LINE) {
                        System.out.println(horse + " won!");
                        exec.shutdownNow();
                        return;
                    }
                    try{
                        TimeUnit.MILLISECONDS.sleep(pause);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
            }
        });
        for(int i = 0; i < hourses; i++) {
            Horse horse = new Horse(barrier);
            horses.add(horse);
            exec.execute(horse);
        }
```

```
    }

    public static void main(String[] args) {
        int nH = 7;
        int pause = 200;
        if(args.length > 0) {
            int n = new Integer(args[0]);
            nH = n > 0 ? n : nH;
        }
        if(args.length > 1) {
            int p = new Integer(args[1]);
            pause = p > -1 ? p : pause;
        }
        new HorseRace(nH, pause);
    }
}
```

```
package Runtime.Thread;

import java.util.Random;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

class TaskPortion implements Runnable {

    private static int counter = 0;
    private final int id = counter++;
    private static Random random = new Random(47);
    private final CountDownLatch latch;

    TaskPortion(CountDownLatch latch) {
        this.latch = latch;
    }

    @Override
    public void run() {
        try {
            doWork();
            latch.countDown();
        } catch (InterruptedException e) {
            System.out.println(this + "interrupted");
        }
    }

    public void doWork() throws InterruptedException {
        TimeUnit.MILLISECONDS.sleep(random.nextInt(2000));
        System.out.println(this + " completed");
    }
```

```
        @Override
        public String toString() {
            return String.format("%1$-3d", id);
        }
    }
     class WaitingTask implements Runnable {

        private static int counter = 0;
        private final int id = counter++;
        private final CountDownLatch latch;

        WaitingTask(CountDownLatch latch) {
            this.latch = latch;
        }

        @Override
        public void run() {
            try{
                latch.await();
                System.out.println("lanch barrier passed for " + this);
            } catch (InterruptedException e) {
                System.out.println(this + "interrupted");
            }
        }

        @Override
        public String toString() {
            return String.format("WaitingTask %1$-3d", id);
        }
    }

    public class CountDownLatchDemo {
        static final int size =100;

        public static void main(String[] args) {
            ExecutorService exec = Executors.newCachedThreadPool();
            CountDownLatch latch = new CountDownLatch(size);
            for(int i = 0; i < 10; i++)
                exec.execute(new WaitingTask(latch));
            for(int i = 0; i < size; i++)
                exec.execute(new TaskPortion(latch));
            System.out.println("Launched all tasks");
            exec.shutdownNow();
        }
    }
```

1. DelayedQueue 是一个无界的BlockingQueue， 是一个优先级队列，队列里面的元素 也应是任务，继承Delayed， 实现getDelay()和comparedTo()
2. `trigger = System.nanoTime() + NANOSECONDS.convert(delta, MILLISECONDS);` 将int转化为纳秒
3. PriorityBlokingQueue,按照优先级 先后执行
4. scheduledExecutor .schedule()执行一次任务 .scheduleAtFiedRate() 按照固定频率执行任务；

5. Semaphore锁同时允许多个对象访问有限的对象 package Runtime.Thread; import java.util.ArrayList; import java.util.List; import java.util.concurrent.*; public class Pool { private int size; private List items = new ArrayList<>(); private volatile boolean[] checkedOut; private Semaphore aviailable; public Pool(Class tClass, int size) { this.size = size; checkedOut = new boolean[size]; aviailable = new Semaphore(size, true);//初始话Semaphore for (int i = 0; i < size; i++) try { items.add(tClass.newInstance()); } catch (IllegalAccessException e) { e.printStackTrace(); } catch (InstantiationException e) { e.printStackTrace(); } } public T checkOut() throws InterruptedException { aviailable.acquire();//占用一个资源 return getItem(); } public void checkIn(T x) { if(releaseItem(x)) aviailable.release();//释放一个资源 } public synchronized T getItem() { for(int i = 0; i < size; ++i) { if(!checkedOut[i]) { checkedOut[i] = true; return items.get(i); } } return null; } public synchronized boolean releaseItem(T item) { int index = items.indexOf(item); if(index == -1) return false; if(checkedOut[index]){ checkedOut[index] = false; return true; } return false; }

```
}
    class Fat {
        private volatile double d;
        private static int counter = 0;
        private final int id = counter++;
        public Fat() {
            for(int i = 1; i < 10000; i++) {
                d += (Math.PI + Math.E) / (double) i;
             }
        }
        public void operation() {
            System.out.println(this);
        }

        @Override
        public String toString() {
            return "Fat: id: " + id;
        }
    }
    class CheckoutTask<T> implements Runnable{

        private static int counter = 0;
        private final int id = counter++;
        private Pool<T> pool;

        public CheckoutTask(Pool<T> pool) {
            this.pool = pool;
        }

        @Override
        public void run() {
            try {
                T item = pool.checkOut();
                System.out.println(this + " check out " + item);
                TimeUnit.SECONDS.sleep(1);
                System.out.println(this + "checking in" + item);
                pool.checkIn(item);
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
            }
        }

        @Override
        public String toString() {
            return "CheckoutTask id : " + id ;
        }
    }
    class Demo{
        final static int SIZE = 25;
        public static void main(String[] args) throws Exception {
            final Pool<Fat> pool = new Pool<>(Fat.class, SIZE);
            ExecutorService exe = Executors.newCachedThreadPool();
            for(int i = 0; i < SIZE; i++) {
                exe.execute(new CheckoutTask<Fat>(pool));
            }
            System.out.println("All CheckoutTasks created");
            List<Fat> list = new ArrayList<>();
            for(int i = 0; i < SIZE; i++) {
                Fat f = pool.checkOut();
                System.out.println(i + ": main() thread checked out ");
                f.operation();
                list.add(f);
            }
            Future<?> blocked = exe.submit(new Runnable() {
                @Override
                public void run() {
                    try{
                        pool.checkOut();
                    } catch (InterruptedException e) {
                        System.out.println("checkout()  Interrupted" );
                    }
                }
            });
            TimeUnit.SECONDS.sleep(2);
            blocked.cancel(true);
            System.out.println("Checking in objects in " + list);
            for(Fat f : list)
                pool.checkIn(f);
            for (Fat f : list)
                pool.checkIn(f);
            exe.shutdownNow();
        }
    }
```

6. CopyOnWriteArraylist可以保证两个线程同时修改不会出错
7. 有一个 holder List Exchanger<List> exchanger. exchange(holder）传入这个方法重新赋值给holder， 另一个线程便可以同时进行holder.remove;
8. 如果有对各Atomic对象，可能会被强制要求放弃这种用法，转而使用更加常规的互斥

枚举学习

1. 如果自定义自己的方法，必须在enum实例序列的最后一个添加分号；
2. toString()方法可以调用name()方法来获取实例的名字。
3. values()可以获取所有的实例
4. 枚举里面定义的都是 menu的实例
5. Enum并没有values这个方法，是由编译器添加的static方法，而且还添加了valueOf方法
6. enum不能被继承，因为被声明为final类
7. Class中有个getEnumConstants()，可以获取所有enum实例
8. enum可以实现一个或多个接口
9. 利用接口可以实现enum的扩展，也可以把一个枚举嵌套到另一个枚举内进行扩展
10. EnumSet非常快速高效，不用担心性能
11. EnumSet allOf(Class elementType): 创建一个包含指定枚举类里所有枚举值的EnumSet集合。 EnumSet complementOf(EnumSet e): 创建一个其元素类型与指定EnumSet里元素类型相同的EnumSet集合，新EnumSet集合包含原EnumSet集合所不包含的、此类枚举类剩下的枚举值（即新EnumSet集合和原EnumSet集合的集合元素加起来是该枚举类的所有枚举值）。

EnumSet copyOf(Collection c): 使用一个普通集合来创建EnumSet集合。 EnumSet copyOf(EnumSet e): 创建一个指定EnumSet具有相同元素类型、相同集合元素的EnumSet集合。 EnumSet noneOf(Class elementType): 创建一个元素类型为指定枚举类型的空EnumSet。 EnumSet of(E first,E...rest): 创建一个包含一个或多个枚举值的EnumSet集合，传入的多个枚举值必须属于同一个枚举类。 EnumSet range(E from,E to): 创建一个包含从from枚举值到to枚举值范围内所有枚举值的EnumSet集合

12. EnumSet 的基础是long， EnumSet最多不超过64个，如果超过，可能会扩充
13. EnumMap 要求键必须来自一个enum， EnumSet元素必须全部来自一个enum
14. 可以给enum实例编写方法，给每个实例赋予不同的行为，方法是定义一个抽象方法，也可以覆盖方法
15. Java 只支持单路分发， 如果执行的操作包含了不止一个类型未知的对象，java的动态绑定机制只能处理其中一个， java多路分发的处理

注解

1、元注解

元注解是指注解的注解。包括 @Retention @Target @Document @Inherited四种。

2. 默认值 不能为null 元素必须要么具有默认值，要么在使用注解时提供元素的值。
3. 注解不支持继承
4. 注解处理工具apt, 操作java源文件，在处理后进行编译
5. 1.8 移除了apt 不学了
6.