# 1、Java基础

## List的实现类，以及实现类的优缺点

   a、ArrayList：实现是基于动态数组的数据结构

   b、LinkedList：实现是基于链表的数据结构

   c、Vector：同ArrayList的数据结构，但它是线程安全的

 优缺点：

   a、对于随机访问get，ArrayList优于LinkedList，因为LinkedList要移动指针

   b、对于新增和删除操作add和remove，LinkedList优于ArrayList，为ArrayList需要移动数据

   c、Vector是同步的,ArrayList/LinkedList是不同步的(在多线程的情况下，有时候就不得不使用Vector了)

   d、扩容机制

ArrayList:

|  |
| --- |
| **int** newCapacity = oldCapacity + (oldCapacity >> 1); |

Vector:

|  |
| --- |
| **int** newCapacity = oldCapacity + ((**capacityIncrement** > 0) ?  **capacityIncrement** : oldCapacity); |

## Map的实现类，以及实现类的优缺点

1. HashMap：基于Hash表实现、线程不安全、key允许为null。默认
2. 底层实现

HashMap是散列表，存储内容为（key-value）映射键值对

HashMap继承于AbstractMap，实现Map、Cloneable、java.io.Serializable接口

HashMap不是线程安全的，但可以用Collections的SynchronizedMap方法

HashMap的映射不是有序的

HashMap有两个参数影响其性能：初始容量默认为16、加载因子为0.75（减少了空间开销，增加了查询成本）

|  |
| --- |
| */\*\*  \* The default initial capacity - MUST be a power of two.  \*/* **static final int *DEFAULT\_INITIAL\_CAPACITY*** = 1 << 4; *// aka 16*  */\*\*  \* The maximum capacity, used if a higher value is implicitly specified  \* by either of the constructors with arguments.  \* MUST be a power of two <= 1<<30.  \*/* **static final int *MAXIMUM\_CAPACITY*** = 1 << 30;  */\*\*  \* The load factor used when none specified in constructor.  \*/* **static final float *DEFAULT\_LOAD\_FACTOR*** = 0.75f;  **int** **threshold = *MAXIMUM\_CAPACITY\* DEFAULT\_LOAD\_FACTOR;*** |

确定哈希桶数组索引位置，Hash算法本质是三步：取key的hashcode值、高位运算、取模运算

|  |
| --- |
| **static final int** hash(Object key) {  **int** h;  **return** (key == **null**) ? 0 : (h = key.hashCode()) ^ (h >>> 16); } |

HashMap是数组+链表+红黑树（1.8增加的），当链表太长时（默认超过8），链表转换为红黑树

如图：



分析put(K key,V value)方法：



1. 处理哈希冲突的方法

开放定地址法——线性探测法

开放定地址法——平方探查法

链表解决——可以用红黑树提高查找效率

  b、LinkedHashMap：LinkedHashMap是HashMap的一个子类，它保留插入的顺序

  c、TreeMap：基于红黑树、线程不安全、可自定义排序器

  d、HashTable：基于单向链的二维数组，有序存储、key值不允许为null、线程安全的，synchronized是针对整张Hash表的，即每次锁住整张表让线程独占

  e、ConcurrentHashMap：ConcurrentHashMap和HashTable主要区别就是围绕着锁的粒度以及如何锁

 优缺点：

  a、ConcurrentHashMap的三点：

     ConcurrentHashMap的锁分段技术（锁桶或段）

     ConcurrentHashMap的锁是否需要加锁，为什么？（否，完全并发）

     ConcurrentHashMap的迭代器是弱一致性

  b、HashTable与ConcurrentHashMap（线程安全）

      HashTable和ConcurrentHashMap主要区别就是围绕着锁的粒度以及如何锁



         如图所示：左边是HashTable的实现方式——锁整个Hash表；右边是ConcurrentHashMap的实现方式——锁桶（或段），ConcurrentHashMap将Hash表分为16个桶（默认值，如get、put、remove等常用操作只锁当前用到的桶），原来只能一个线程进入，现在同时16个写线程进入（写线程需要锁定，读线程几乎不受限制），并发性提高。

       ConcurrentHashMap的读取并发，因为在读取时大多数时候并没有用到锁，所以读取操作几乎完全的并发操作，而写操作锁定的粒度又非常细，比之前又更加快速（在桶更多时表现得更明显），只是在求size等操作时才需要锁定整个表。

      在迭代时，ConcurrentHashMap使用了弱一致迭代器。在迭代中，当iterator被创建后集合再发生改变就不再抛出ConcurrentModificationException,取而代之的是改变时new新的数据从而不影响原来的数据,iterator完成后再将头指针替换为新的数据。这样iterator线程可以使用原来老的数据，而写线程也可以并发完成改变。更重要的是，保证了多线程并发执行的连续性和扩展性，是性能提升的关键字。

ConcurrentHashMap中主要三个实现类：

  ConcurrentHashMap（整个Hash表）

  Segment（桶）

  HashEntry（节点）

## Set的实现类，以及实现类的优缺点

   HashSet：使用map来存储，因此值不可重复 初始化new HashMap()

LinkedHashSet：初始化 new LinkedHashMap();

## Collection

   Arrays.sort()原理：基本类型数据使用快速排序法，对象数组使用归并排序。

   Collections.sort()原理：合并排序

## 关键字

final: 1、修饰变量 2、修饰方法 3、修饰类，不能被继承

|  |
| --- |
| public class FinalTest {  public final int i = 1; //修饰变量    public final void test() { //修饰方法    }  static final class InnerClass{ //修饰类，不能被继承    }  } |

    finally: 为异常处理的一部分，它只能用在try/catch语句中，并且附带一个语句块，表示这段语句最终一定会被执行（不管有没有抛出异常），经常被用在需要释放资源的情况下。

static：static关键字：1、修饰静态包 2、修饰变量 3、修饰静态块 4、修饰内部类

|  |
| --- |
| import static java.lang.System.out; //静态导包  public class StaticTest {  public static int count; //修饰变量  static { //修饰静态块  count ++;  }    static class InnerClass{//修饰内部类    }  } |

transient关键字：修饰变量

描述：

    1、当对象序列化时，transient阻止实例中那些用此关键字的声明的变量  
    2、当反序列化时，这样的实例变量值不会被持久化和恢复  
    例如，当反序列化对象——数据流（例如，文件）可能不存在时，原因是你的对象中存在类型为java.io.InputStream的变量，序列化时这些变量引用的输入流无法被打开。

|  |
| --- |
| public class TransientTest implements Serializable{       private static final long serialVersionUID = 1L;      private transient InputStream is; //修饰变量      private int majorVer;     private int minorVer;     TransientTest(InputStream is) throws IOException{         System.out.println("TransientTest(InputStream) called");         this.is = is;         DataInputStream dis;         if (is instanceof DataInputStream) {             dis = (DataInputStream) is;         }else {             dis = new DataInputStream(is);         }         if (dis.readInt() != 0xcafebabe) {             throw new IOException("not a .class file");         }         majorVer = dis.readShort();         minorVer = dis.readShort();     }     public int getMajorVer() {         return majorVer;     }     public int getMinorVer() {         return minorVer;     }      void showIS(){         System.out.println(is);     } } |

## 引用

|  |  |  |  |
| --- | --- | --- | --- |
|  | GC回收时间 | 用途 | 生存时间 |
| 强引用 | Never | 对象的一般状态 | JVM停止运行时 |
| 软引用 | 内存不足时 | 对象缓存 | 内存不足时终止 |
| 弱引用 | GC时 | 对象缓存 | GC后终止 |
| 虚引用 | unknow | unknow | unknow |

|  |
| --- |
| public void reference(){  TestJvm jvm = new TestJvm();  // 强引用 是指创建一个对象并把这个对象赋给一个引用变量。  Object object = new Object();  Object[] objArr = new Object[1000];  // 软引用  SoftReference softReference = new SoftReference(jvm);  // 弱引用  WeakReference weakReference = new WeakReference(jvm);  // 虚引用  ReferenceQueue<String> queue = new ReferenceQueue<String>();  PhantomReference<String> pr = new PhantomReference<String>(new String("hello"), queue);  System.out.println(pr.get());  } |

## 反射机制

|  |
| --- |
| @Data  public class TestReflect implements TestReflectImpl{  private String nameVal;  String ageVal;  public TestReflect(String nameVal,String ageVal){  this.nameVal = nameVal;  this.ageVal = ageVal;  }  public void sayChina() {  System.out.println("hello ,china");  }  public void sayHello(String name, int age) {  System.out.println(name+" "+age);  }  }  interface TestReflectImpl{  public static final String name="name";  public static int age=20;  public void sayChina();  public void sayHello(String name, int age);  }  class TestReflectMain{  public static void main(String[] args){  try{  Class clazz = Class.forName("com.yaozou.jdk.TestReflect");  // 获得此类的构造方法  Constructor<?>[] constructors = clazz.getConstructors();  System.out.println("constructors:"+constructors.length);  Object[] params = null;  Class[] paramsClazzs = null;  if ((paramsClazzs=constructors[0].getParameterTypes()).length > 0){  params = new Object[paramsClazzs.length];  int i = 0;  for (Class paramsClazz:paramsClazzs) {  params[i] = paramsClazz.newInstance();  i++;  }  }  //实例化  Object obj = constructors[0].newInstance(params);  //获得Field  Field[] fields = clazz.getDeclaredFields();  System.out.println("fields:"+fields.length);  for (Field field:fields) {  field.setAccessible(true);  System.out.println("field:"+field.getName());  field.set(obj,"aaaaa");  }  //获得所有方法  Method[] methods = clazz.getMethods();  System.out.println("methods:"+methods.length);  for (Method method:methods) {  System.out.println("metthod:"+method.getName());  if (method.getName().equals("sayHello")) {  Object[] paramMethods = null;  Type[] typeParams;  if ((typeParams = method.getParameterTypes()).length > 0){  int i = 0;  paramMethods = new Object[typeParams.length];  for (Type typeParam:typeParams ) {  System.out.println("type:"+typeParam.toString());  Object object = null;  if (typeParam instanceof Class){  object = "ueee";  }else if (typeParam.equals("int")) object = 1;  paramMethods [i] = object;  i++;  }  }  System.out.println("paramMethod:"+paramMethods.length);  method.invoke(obj,paramMethods);  }  }  }catch(Exception e){  e.printStackTrace();  }  }  } |

## cloneable接口实现原理

clone: 允许在堆中克隆出一块和原对象一样的对象，并将这个对象的地址赋予新的引用。

Java中所有类都默认继承java.lang.Object类，在java.lang.Object类中有一个方法clone()，这个方法将返回Object对象的一个拷贝。

特点：

1. 拷贝对象返回的是一个对象，而不是一个引用
2. 拷贝对象与用new操作符返回的新对象的区别是拷贝的对象已经包含了一些原来对象的信息，而不是对象的初始信息。
3. 重载了clone()方法，调用super.clone()就是直接或间接调用了java.lang.Object类的clone()方法。

|  |
| --- |
| **public class** TestCloneable **implements** Cloneable{  @Override  **protected** Object clone() **throws** CloneNotSupportedException {  **return super**.clone();  } } |

## 线程

### 简介

线程是程序执行流的最小单元。是程序中一个单一的顺序控制流程。是进程中的一个实体，是被系统独立调度和分派的基本单位。进程内有一个相对独立的、可调度的执行单位，是系统独立调度和分派CPU的基本单位指令运行时的程序的调度单位。在单个程序中同时运行多个线程完成不同的工作，称为多线程。

### 生命周期

    1、新建状态

使用new关键字和Thread类或其子类建立一个线程对象后，该线程对象就处于新建状态。它保持这个状态直到程序start()这个线程。

1. 就绪状态

当线程对象调用了start()方法之后，该线程就进入就绪状态。就绪状态的线程处于就绪队列中，要等待JVM里线程调度器的调度。

1. 运行状态

如果就绪状态的线程获取CPU资源，就可以执行run()，此时线程便处于运行状态。处于运行状态的线程最为复杂，它可以变为阻塞状态、就绪状态和死亡状态。

1. 阻塞状态

如果一个线程执行了sleep（睡眠）、suspend（挂起）等方法，失去所占用资源之后，该线程就从运行状态进入阻塞状态。在睡眠时间已到或获得设备资源后可以进入就绪状态。可以分为三种：

等待阻塞：运行状态中的线程执行wait()方法，使线程进入到等待阻塞状态。

同步阻塞：线程在获取synchronized同步锁失败（因为同步锁被其他线程占用）

其他阻塞：通过调用线程的sleep()或join()发出了I/O请求时，线程就会进入阻塞状态。当sleep()状态超时，join()等待线程终止或超时，或者I/O处理完毕，线程重新转入就绪状态。

1. 死亡状态

一个运行的线程完成任务或者其他终止条件发生时，该线程就切换到终止状态。

### 线程的优先级

其取值范围1(Thread.MIN\_PRIORITY)-10(Thread.MAX\_PRIORITY)

默认情况下，每一个线程都会分配一个优先级 NORM\_PRIORITY（5）。

具有较高优先级的线程对程序更重要，并且应该在低优先级的线程之前分配处理器资源。但是，线程优先级不能保证线程执行的顺序，而且非常依赖于平台。

### 创建线程

1. 实现Runnable接口
2. 继承Thread类本身
3. Callable和Future创建线程

### sleep和wait的区别

1. sleep()方法属于Thread类，wait()方法属于Object类。
2. sleep()方法使线程暂停执行指定的时间，让出cpu给其他线程，但是它的监控状态依然保持，当指定时间到了自动恢复运行状态。
3. sleep()方法调用过程中，线程不会释放对象锁
4. wait()方法调起时，线程会放弃对象锁。线程进入等待此对象的等待锁定池。只有针对此对象调用的notify()方法后线程才会进入对象锁定池准备，获得了对象锁进入运行状态。

## 数组

数组中内存如何分配

|  |
| --- |
| **public static void** main(String[] args){  **int**[] arr=**new int**[3];  arr[0]=10;  arr[1]=20;  arr[2]=70;  System.***out***.println(arr);   **short**[] arr2=**new short**[2];  arr2[0]=30;  arr2[1]=40;  System.***out***.println(arr2);    **int**[] arr3=arr;  arr3[0]=100;  arr3[1]=200;  arr3[2]=300;  System.***out***.println(arr); } |



# 2、并发

## synchronized

|  |
| --- |
| **class** TestSynchronized{  Object **lock** = **null**;  **public void** method1(){  **lock** = **new** Object();  **synchronized** (**lock**){  System.***out***.println(**"----method1----"**);  }  }  **public void** method2(){  **synchronized** (**this**){  System.***out***.println(**"----method2----"**);  }  }  **public synchronized void** method3(){  System.***out***.println(**"----method3----"**);  }  **public static synchronized void** method4(){  System.***out***.println(**"----method4----"**);  }  } |

|  |  |  |
| --- | --- | --- |
| 类别 | Synchronized | Lock |
| 存在形式 | JAVA关键字，JVM | 是一个类 |
| 锁的释放 | 1. 获取锁的线程执行完成同步中的代码 2. 线程发生异常 | 在finally中必须释放锁，不然惠造成死锁 |
| 获取锁 | 如果A线程获取了锁，那么B线程等待。如果A线程阻塞，那么B线程会一直等待 | Lock有多种锁的获取方式，都可以尝试获得锁，线程可以不用一直等待 |
| 锁状态 | 无法判断 | 可判断 |
| 锁类型 | 悲观锁（可重入 不可中断 非公平） | 乐观锁（可重入 可判断 可公平（两者皆可）） |
| 性能 | 少量同步 | 大量同步 |
|  | 偏向锁-》轻量级锁-》重量级锁（1.6后） | 互斥锁ReentrantLock |

## Volatile

volatile：用来修饰被不同线程访问和修改的变量

volatile关键字与JVM内存模型相关

  Java语言是支持多线程的，为了解决线程并发的问题，在语言内部引入了同步块和volatile关键字机制

volatile关键字机制：

 synchronized修饰方法和代码块，以实现同步

 用volatile修饰的变量，线程在每次使用变量的时候，都会读取变量修改后的值。volatile经常被误用进行原子性的操作。但是这些操作并不是真正的原子性。在读取加载之后，如果变量发生了变化，线程工作内存中的值由于已加载，不会产生对应的变法。对于volatile修改的变量，JVM只是保证从内存加载到线程工作内存的值是最新的。

             交互图：



|  |
| --- |
| public class VolatileTest {     //public static int count = 0; //实际运算每次结果都不一样     public static volatile int count = 0; //     public static void inc(){          //这里延迟1毫秒，使得结果明显         try {             Thread.sleep(1);         } catch (Exception e) {             // TODO: handle exception         }         count ++;     }          public static void main(String[] args) {         //同时启动1000个线程，去进行i++运算，看看实际结果         for (int i = 0; i < 1000; i++) {             new Thread(new Runnable() {                  @Override                 public void run() {                     VolatileTest.inc();                 }             }).start();         }                  System.out.println("运行结果:"+count);     }  } |

## 信号量Semaphore

在多线程中，线程间传递信号的一种方式。

与互斥量的区别

1. 互斥量用于线程的互斥，信号量用于线程的同步
2. 互斥量值只能为0/1，信号量值可以为非负整数。信号量可以实现多个同类资源的多线程互斥和同步。
3. 互斥量的加锁和解锁必须由同一线程分别对应使用，信号量可以由一个线程释放，另一个线程得到。

|  |
| --- |
| 1. **class** TestSemaphore{  Semaphore **semaphore** = **new** Semaphore(10);*//控制10个共享资源的使用* **public void** use(){  **try** {  **semaphore**.acquire();  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  **public void** release(){  **semaphore**.release();  } } |

## 实现所有线程在等待某个事件的发生才去执行

1.闭锁CountDownLatch

闭锁是典型的等待事件发生的同步工具类，将闭锁的初始值设置1，所有线程调用await方法等待，当事件发生时调用countDown将闭锁值减为0，则所有await等待闭锁的线程得以继续执行。

2.阻塞队列BlockingQueue

所有等待事件的线程尝试从空的阻塞队列获取元素，将阻塞，当事件发生时，向阻塞队列中同时放入N个元素(N的值与等待的线程数相同)，则所有等待的线程从阻塞队列中取出元素后得以继续执行。

3.信号量Semaphore

设置信号量的初始值为等待的线程数N，一开始将信号量申请完，让剩余的信号量为0，待事件发生时，同时释放N个占用的信号量，则等待信号量的所有线程将获取信号量得以继续执行。

4.栅栏CyclicBarrier

设置栅栏的初始值为1，当事件发生时，调用barrier.wait()冲破设置的栅栏，将调用指定的Runable线程执行，在该线程中启动N个新的子线程执行。这个方法并不是让执行中的线程全部等待在某个点，待某一事件发生后继续执行。

## CAS缺陷以及如何解决

参考地址：https://blog.csdn.net/jeffleo/article/details/75269618

CAS：Compare And Swap，即比较并交换。（AtomicInteger）

实现了Java多线程的并发操作。整个AQS同步组件、Atomic原子类操作等都是以CAS实现的，甚至ConcurrentHashMap在1.8的版本中也调整为了CAS+Synchronized。可以说CAS是整个JUC的基石。

缺陷：

主要表现在三个方法：

循环时间太长

只能保证一个共享变量原子操作

ABA问题：解决方案则是版本号

## AQS

参考资料：https://www.cnblogs.com/daydaynobug/p/6752837.html

AQS:AbstractQueuedSynchronized, 抽象的队列式的同步器，AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch

AQS里的state只有两种状态：0表示未锁定，1表示锁定

## 死锁、活锁

死锁：指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。4个产生条件：

1. 互斥条件
2. 请求和保持条件
3. 不剥夺条件
4. 环路等待条件

活锁：指的是任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试—失败—尝试—失败的过程。处于活锁的实体是在不断的改变状态，活锁有可能自行解开。

## 线程池

1、创建一个缓冲池，缓冲池容量大小为Integer.MAX\_VALUE

Executors.newCachedThreadPool();

2、创建容量为1的缓冲池

Executors.newSingleThreadExecutor();

3、创建固定容量大小的缓冲池

Executors.newFixedThreadPool(int);

配置线程池大小：

1. CPU密集型任务，就需要尽量压榨CPU，参考值可以设为 *N*CPU+1
2. IO密集型任务，参考值可以设置为2\**N*CPU

## ThreadLoad原理

|  |
| --- |
| **class** TestThreadLocal{  **private static** TestThreadLocal *testThreadLocal* = **null**;  **private static** ThreadLocal<TestThreadLocal> *map* = **new** ThreadLocal<TestThreadLocal>();  **private** TestThreadLocal(){}  **public static synchronized** TestThreadLocal getInstance1(){  **if**(*testThreadLocal*==**null**){  *testThreadLocal* = **new** TestThreadLocal();  }  **return** *testThreadLocal*;  }  **public static** TestThreadLocal getInstance2(){ *//效率大于getInstance1()  testThreadLocal* = *map*.get();  **if**(*testThreadLocal*==**null**){  *testThreadLocal* = **new** TestThreadLocal();  *map*.set(*testThreadLocal*);  }  **return** *testThreadLocal*;  } } |

## LockSupport工具

## Condition原理

## Fork/Join框架的理解

## 分段锁的原理、锁力度的减小思考

## 八种阻塞队列以及各个阻塞队列的特性

3、JVM

1. 运行时数据区域
2. JMM（JVM内存模型）
3. Minor GC、Full GC
4. 各种回收算法
5. 类的实例化顺序
6. 类加载，为何使用双亲委派模式？是否有场景可以打破这种模式?
7. Finalize()方法

4、Redis

1. 各命令及底层实现
2. 穿透、缓存雪崩
3. 使用redis实现分布式锁
4. 并发竞争问题
5. 持久化的几种方式、优缺点、怎么实现
6. 缓存失败策略
7. redis集群高可用原理
8. redis缓存分片
9. redis的数据淘汰策略

5、MySQL

1. 分页优化
2. 悲观锁、乐观锁
3. 组合索引、最左原则
4. 表锁、行锁
5. 性能优化
6. 索引分类：B+,Hash 什么情况用什么索引？
7. 事务的四大特性
8. 事务的隔离级别

6、Netty

1. BIO、NIO和AIO
2. TCP粘包/拆包的原因及解决方法
3. TCP的三次握手

7、Spring

1. BeanFactory、FactoryBean
2. IoC的理解及其初始化过程
3. BeanFactory和Application
4. Spring Bean生命周期如何被管理，加载过程
5. Spring AOP怎么实现
6. Spring事务管理机制：1、如何管理事务 2、不同传播行为有哪些，有何用
7. Spring中使用了哪些设计模式
8. Spring MVC的工作原理
9. Spring 循环注入的原理
10. Spring AOP各个术语、怎么相互工作
11. Spring 保证controller并发的安全

8、MyBatis（源码重写）、Spring（源码重写）

9、数据结构算法

10、分布式