# 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

## 运行时数据区域、JMM（JVM内存模型）



## Minor GC、Full GC

|  |  |  |
| --- | --- | --- |
|  | Minor GC 新生代 | Full GC/Major GC 老年代 |
| -XX:+UseSerialGC | 串行GC | 串行GC |
| -XX:+UseParallelGC | PS GC | 并行MSC GC |
| -XX:+UseConcMarkSweepGC | ParNew GC | 并行GC  当出现concurrent Mode  Failure时采用串行GC |
| -XX:+UseParNewGC | 并行GC | 串行GC |
| -XX:+UseParallelOldGC | PS GC | 并行Compacting GC |
| -XX:+UseConcMarkSweepGC  -XX:+UseParNewGC | 串行GC | 并发GC  当出现Concurrent Mode  Failure或promotion failed时采用串行GC |
| 不支持的组合方式 | 1、-XX:+UseParNewGC -XX:+UseParallelOldGc  2、-XX:+UseParNewGC -XX:+UseSerialGC | |

判断对象是否“存活”或“死亡”：

1. 引用计数算法：

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器的值加1；当引用失效时，计数器的值减；当该对象的计数器的值为0时，标志该对象失效。

1. 跟搜索算法：

通过一系列的名为“GCRoots”的对象作为起始点，从这些节点开始向下搜索，搜索过的路径称为引用链，当一个对象到GCRoots没有任何引用链相连（用图论的话来说就是从GC Roots到这个对象不可达）时，则证明对象是不可用的。

## 各种回收算法

1. 标记-清除算法

首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象

1. 复制算法

将可用内存按容量划分为大小相等的两块，每次使用其中的一块，当一块的内存用完了，就将还存活的对象复制到另一块上面，然后再把已使用过的内存空间一次清空（Eden和Survivor）

1. 标记-整理算法

标记过程仍然与标记-清楚算法一样，让所以存活对象都向一端移动，然后直接清理掉边界意外的内容

1. 分代收集算法

## 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?

类的生命周期：



类加载双亲委派模型



破坏双亲委派模型：线程上下文类加载器(Thread Context ClassLoader)

## Finalize()方法

是Object的protected方法，子类可以覆盖该方法以实现资源清理工作，GC在回收对象之前调用该方法。

|  |
| --- |
| **class** TestGC{  **public static** TestGC *testGC* = **null**;  **public static void** main(String[] args) **throws** InterruptedException {  *testGC* = **new** TestGC();  *testGC* = **null**;  System.*gc*();  Thread.*sleep*(500);  **if** (**null** != *testGC*) { *//此时对象应该处于(reachable, finalized)状态* System.***out***.println(**"Yes , I am still alive"**);  } **else** {  System.***out***.println(**"No , I am dead"**);  }  *testGC* = **null**;  System.*gc*();  Thread.*sleep*(500);  **if** (**null** != *testGC*) {  System.***out***.println(**"Yes , I am still alive"**);  } **else** {  System.***out***.println(**"No , I am dead"**);  }  }  @Override  **protected void** finalize() **throws** Throwable {  **super**.finalize();  System.***out***.println(**"execute method finalize()"**);  *testGC* = **this**;  } }  C:\Users\yao\AppData\Local\Temp\1526614472(1).png |

# 4、Redis

## 各命令及底层实现

1. 远程 redis 服务上执行命令:

redis-cli -h host -p port -a password

1. 字符串(String)

字符串数据类型的相关命令用于管理 redis 字符串值

|  |  |
| --- | --- |
| SET key value | 设置指定 key 的值 |
| GET key | 获取指定 key 的值 |
| GETRANGE kay start end | 返回 key 中字符串值的子字符 |
| GETSET key value | 将给定 key 的值设为 value ，并返回 key 的旧值(old value)。 |

1. 哈希(Hash)

Redis hash 是一个string类型的field和value的映射表，hash特别适合用于存储对象。

Redis 中每个 hash 可以存储 232 - 1 键值对（40多亿）。

|  |  |
| --- | --- |
| HSET key field value | 将哈希表 key 中的字段 field 的值设为 value 。 |
| HGET key field | 获取存储在哈希表中指定字段的值。 |
| HDELETE key field1[field2] | 删除一个或多个哈希表字段 |

1. 列表(List)

Redis列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）

一个列表最多可以包含 232 - 1 个元素 (4294967295, 每个列表超过40亿个元素)。

|  |  |
| --- | --- |
| LPUSH key value1[value2] | 将一个或多个值插入到列表头部 |
| LPUSHX key value | 将一个值插入到已存在的列表头部 |
| LPOP key | 移出并获取列表的第一个元素 |
| LLEN key | 获取列表长度 |

1. 集合(Set)

Redis 的 Set 是 String 类型的无序集合。集合成员是唯一的，这就意味着集合中不能出现重复的数据。

Redis 中集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是 O(1)。

集合中最大的成员数为 232 - 1 (4294967295, 每个集合可存储40多亿个成员)。

|  |  |
| --- | --- |
| SADD key member1 [member2] | 向集合添加一个或多个成员 |
| SMEMBERS key | 返回集合中的所有成员 |
| SISMEMBER key member | 判断 member 元素是否是集合 key 的成员 |
| SPOP key | 移除并返回集合中的一个随机元素 |

## 穿透、缓存雪崩

## 使用redis实现分布式锁

## 并发竞争问题

## 持久化的几种方式、优缺点、怎么实现

## 缓存失败策略

## redis集群高可用原理

## redis缓存分片

## redis的数据淘汰策略

# 5、MySQL

## 分页优化

## 悲观锁、乐观锁

## 组合索引、最左原则

## 表锁、行锁

## 性能优化

## 索引分类：B+,Hash 什么情况用什么索引？

## 事务的四大特性

## 事务的隔离级别

# 6、Netty

## BIO、NIO和AIO

## TCP粘包/拆包的原因及解决方法

## TCP的三次握手

# 7、Spring

## BeanFactory、FactoryBean

1. FactoryBean其实就是一个简单工厂,实现其方法覆写getObject方法可以直接简易的实现工厂模式.
2. BeanFactory是Spring的核心接口,说白了其实也是采用了工厂模式,根据传入的不同bean名字,之后调用容器(如DefaultListableBeanFactory)返回具体的bean实例.我们常用的ClassPathXmlApplicationContext以及FileSystemXmlApplicationContext等都实现了此接口.

## IoC的理解及其初始化过程

## BeanFactory和Application

## Spring Bean生命周期如何被管理，加载过程

## Spring AOP怎么实现

## 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

## Spring中使用了哪些设计模式

1.工厂模式，这个很明显，在各种BeanFactory以及ApplicationContext创建中都用到了；

2.模版模式，这个也很明显，在各种BeanFactory以及ApplicationContext实现中也都用到了；

3.代理模式，在AOP实现中用到了JDK的动态代理；

4.单例模式，这个比如在创建bean的时候。

5.Tomcat中有很多场景都使用到了外观模式，因为Tomcat中有很多不同的组件，每个组件需要相互通信，但又不能将自己内部数据过多地暴露给其他组件。用外观模式隔离数据是个很好的方法。

6.策略模式在Java中的应用，这个太明显了，因为Comparator这个接口简直就是为策略模式而生的。[Comparable和Comparator的区别](http://www.cnblogs.com/baizhanshi/p/6201875.html" \t "_blank)一文中，详细讲了Comparator的使用。比方说Collections里面有一个sort方法，因为集合里面的元素有可能是复合对象，复合对象并不像基本数据类型，可以根据大小排序，复合对象怎么排序呢？基于这个问题考虑，Java要求如果定义的复合对象要有排序的功能，就自行实现Comparable接口或Comparator接口.

7.原型模式：使用原型模式创建对象比直接new一个对象在性能上好得多，因为Object类的clone()方法是一个native方法，它直接操作内存中的二进制流，特别是复制大对象时，性能的差别非常明显。

8.迭代器模式：Iterable接口和Iterator接口这两个都是迭代相关的接口，可以这么认为，实现了Iterable接口，则表示某个对象是可被迭代的；Iterator接口相当于是一个迭代器，实现了Iterator接口，等于具体定义了这个可被迭代的对象时如何进行迭代的

## Spring MVC的工作原理

## Spring 循环注入的原理

## Spring AOP各个术语、怎么相互工作

## Spring 保证controller并发的安全

# 8、24种设计模式和7大原则

## 资料地址

<https://www.cnblogs.com/lspz/p/6859991.html>

https://www.cnblogs.com/maowang1991/archive/2013/04/15/3023236.html

## 7大原则

### 单一职责

一个类一个职责

### 开闭原则

一个软件实体如类、模块和函数对扩展开放，对修改关闭。

### 里氏替换原则

继承和派生类（尽量不重写父类已实现的方法，可以用接口等其他方法绕过）

1. 子类可以实现父类的抽象方法，但不能覆盖父类的非抽象方法
2. 子类中可以增加自己特有的方法
3. 当子类的方法重载父类的方法时，方法的前置条件（即方法的形参）要比父类方法的输入参数更宽松。
4. 当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。

### 依赖倒置原则

以开闭原则为基础，对接口编程，不要针对实现编程。

高层模块不依赖于底层模块，二者都应依赖其抽象细节；细节应该依赖于抽象不依赖于具体。

### 接口隔离原则

建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。降低类之间的耦合度，降低依赖。便于维护和升级。

### 迪米特法则（最少知道原则）

一个类应尽量少的与其他类之间发生相互作用，使得系统功能模块相对独立。就是一个类对自己依赖的类知道的越少越好。也就是说，对于被依赖的类来说，无论逻辑多么复杂，都尽量地将逻辑封装在内部，对外除了提供public方法，不对外泄露任何信息。

### 合成（组合/聚合）复用原则

尽量使用合成/聚合的方式，而不是使用继承的关系来达到复用的原则。

## 24种设计模式

### 创建者模式

#### 单例模式（Singleton）

确保一个类只有一个实例，并提供全局访问点。

使用synchronized关键字锁定对象，一定要恰当使用。

使用类的静态方法实现单例模式效果相同，但两者有区别。

|  |
| --- |
| /\*\*  \* 单例模式  \* 单例对象（Singleton）是一种常用的设计模式。在Java应用中，单例对象能保证在一个JVM中，  \* 该对象只有一个实例存在。这样的模式有几个好处：  1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。  2、省去了new操作符，降低了系统内存的使用频率，减轻GC压力。  3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。  （比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程。  \*/  public class SingletonModel {  /\* 持有私有静态实例，防止被引用，此处赋值为null，目的是实现延迟加载 \*/  private static SingletonModel model = null;  /\* 私有构造方法，防止被实例化 \*/  private SingletonModel(){}  public static SingletonModel getInstance(){  if (model == null) syncInit();  return model;  }  /\* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 \*/  public Object readResolve() {  return model;  }  private synchronized static void syncInit(){  if (model == null) model = new SingletonModel();  }  public void run(){  System.out.println("This is a singleton.");  }  public static void main(String[] args) {  SingletonModel model = SingletonModel.getInstance();  model.run();  }  } |

#### 工厂模式（Factory）

定义一个创建对象的接口，由子类决定要实例化的类是哪一个。工厂方法让类实例化推迟到子类。

普通工厂：建立一个工厂类，对实现了同一接口的一些类进行实例的创建。

多工厂：对普通工厂方法模式进行改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象。

静态工厂：将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

#### 抽象工厂模式（Abstract Factory）

提供一个接口，用于创建相关或依赖对象的家族，而不需要指定具体类。

创建者模式（Builder）

将各种产品集中起进行管理，用来创建复合对象（某个类具有不同的属性）

即使用创建者模式封装一个产品的构造过程，并允许按步骤构造。将一个复杂对象的构建与它表示分离，使得同样的构建过程可以创建不同的表示。

|  |
| --- |
| /\*\*  \* 工厂方法模式  \*/  public class FactoryMethod {  public static void main(String[] args) {  /\*\*  \* 普通工厂模式  \* 建立一个工厂类，对实现了同一接口的一些类进行实例的创建  \* 如：发送信息，可以短信/邮件/qq/微信等  \*/  SenderFactory factory = new SenderFactory();  factory.produce("sms").send();  /\*\*  \* 多工厂模式  \* 对普通工厂方法模式的改进，在普通工厂方法模式中，  \* 如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，  \* 分别创建对象  \*/  new MultiSenderFactory().produceQQ().send();  /\*\*  \* 静态工厂方法模式  \* 多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用  \*/  StaticSenderFactory.produceQQ().send();  /\*\*  \* 抽象工厂模式  \* 工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则。  \* 所以，从设计角度考虑，有一定的问题，如何解决？  \* 就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，  \* 不需要修改之前的代码。  \*/  Sender sender = new QQProduce().produce();  sender.send();  }  }  interface Sender{  void send();  }  class SmsSender implements Sender{  public void send() {  System.out.println("this is sms sender");  }  }  class QQSender implements Sender{  public void send() {  System.out.println("this is QQ sender");  }  }  class SenderFactory{  public Sender produce(String type){  if ("qq".equals(type)) return new QQSender();  else if ("sms".equals(type)) return new SmsSender();  return null;  }  }  class MultiSenderFactory{  public Sender produceQQ(){  return new QQSender();  }  public Sender produceSms(){  return new SmsSender();  }  }  class StaticSenderFactory{  public static Sender produceQQ(){  return new QQSender();  }  public static Sender produceSms(){  return new SmsSender();  }  }  interface Provider{  Sender produce();  }  class QQProduce implements Provider{  public Sender produce() {  return new QQSender();  }  }  class SmsProduce implements Provider{  public Sender produce() {  return new SmsSender();  }  } |

#### 原型模式（Prototype）

当创建给定类的实例过程很复杂时，就使用原形模式。

与工厂模式无关，将一个对象作为原型对其进行复制、克隆，产生一个和原对象类似的新对象。

|  |
| --- |
| /\*\*  \* 原型模式  \* 将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的新对象  \* 1、浅复制：将一个对象复制后，基本数据类型的变量都会重新创建，而引用类型，指向的还是原对象所指向的。  2、深复制：将一个对象复制后，不论是基本数据类型还有引用类型，都是重新创建的。简单来说，  就是深复制进行了完全彻底的复制，而浅复制不彻底。（  要实现深复制，需要采用流的形式读入当前对象的二进制输入，再写出二进制数据对应的对象。）  \*/  @Data  public class PrototypeMethod implements Cloneable,Serializable {  private String key;  private String value;  /\*\*浅复制\*/  @Override  protected Object clone() throws CloneNotSupportedException {  return super.clone();  }  public Object deepClone() throws ClassNotFoundException,IOException{  /\*写入当前对象的二进制流\*/  ByteArrayOutputStream bos = new ByteArrayOutputStream();  ObjectOutputStream oos = new ObjectOutputStream(bos);  oos.writeObject(this);  /\*读出二进制流产生新对象\*/  ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());  ObjectInputStream ois = new ObjectInputStream(bis);  return ois.readObject();  }  } |

#### 多例模式（Multitude）

在一个解决方案中结合两个或多个模式，以解决一般或重复发生的问题

定义：作为对象的创建模式，多例模式中的多例类可以有多个实例，而且多例类必须自己创建、管理自己的实例，并向外界提供自己的实例。

特点：

1、多例类可有多个实例

2、多例类必须自己创建、管理自己的实例，并向外界提供自己的实例

3、根据是否有实例上限分为：有上限多例类和无上限多例类。

|  |
| --- |
| /\*\*  \* 多例模式  \* 在一个解决方案中结合两个或多个模式，以解决一般或重复发生的问题  \* @author yaozou  \* @create 2018-04-26 21:36  \*\*/  public class MultitudeMethod {  private static int num = 3; //实例个数  private static Map<Integer,MultitudeMethod> multitude = new HashMap<Integer, MultitudeMethod>(); //存放实例  private static int index; //当前实例所在位置  static{ //当类加载时创建指定个数的实例  for(int i = 0;i < num;i++){  multitude.put(i,new MultitudeMethod());  }  }  private MultitudeMethod(){}  public static MultitudeMethod getInstance(){  Random random = new Random();  index = random.nextInt(num); //随机获取实例  return multitude.get(index);  }  public int getIndex() {  return index;  }  public static void main(String[] args) {  System.out.println("当前实例是第"+MultitudeMethod.getInstance().getIndex());  System.out.println("当前实例是第"+MultitudeMethod.getInstance().getIndex());  System.out.println("当前实例是第"+MultitudeMethod.getInstance().getIndex());  }  } |

### 结构模式

#### 适配器模式（Adapter）

将一个类的接口，转换成客户期望的另一个接口。适配器让原来不兼容的类可以合作无间。

类的适配器：多重继承

对象的适配器：组合

接口的适配器

|  |
| --- |
| /\*\*  \* 适配器模式  \* 要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。  \*/  public class AdapterMethod {  public static void main(String[] args){  //类的适配器模式  ClassAdapter classAdapter = new ClassAdapter();  classAdapter.run();  //对象的适配器模式  ObjectAdapter objectAdapter = new ObjectAdapter();  objectAdapter.run();  //接口的适配器模式  InterfaceAdapter interfaceAdapter = new InterfaceAdapter();  interfaceAdapter.run();  }  }  interface Targetable{  void methodFirst();  void methodSecond();  }  class Source{  public void methodFirst(){  System.out.println("This a first method.");  }  }  /\*\*  \* 类的适配器模式  \* 核心思想就是：有一个Source类，拥有一个方法，待适配，目标接口时Targetable，通过Adapter类，将Source的功能扩展到Targetable里  \* 应用场景：将一个接口转换成满足另一个新接口的类时，可以使用类的适配器模式，创新一个新类，继承原有的类，实现新的接口。  \*/  class ClassAdapter{  class Adapter extends Source implements Targetable{  public void methodSecond() {  System.out.println("This a second method.");  }  }  public void run(){  Targetable targetable = new Adapter();  targetable.methodFirst();  targetable.methodSecond();  }  }  /\*\*  \* 对象的适配器模式  \* 核心思想：基本思路和类的适配器模式相同，只是将Adapter类作修改，这次不继承Source类，而是持有Source类的实例，以达到解决兼容性的问题。  \* 应用场景：将一个对象转换成满足另一个新接口的对象时，可以创建一个Adapter类，持有原类的一个实例，在Adapter类的方法中，调用实例的方法  \*/  class ObjectAdapter{  @Data  class Adapter implements Targetable{  private Source source;  public Adapter(Source source){  super();  this.source = source;  }  public void methodFirst() {  source.methodFirst();  }  public void methodSecond() {  System.out.println("This a second method.");  }  }  public void run(){  Targetable targetable = new Adapter(new Source());  targetable.methodFirst();  targetable.methodSecond();  }  }  /\*\*  \* 接口的适配器模式  \* 核心思想：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，  \* 因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，  \* 该抽象类实现了该接口，实现了所有的方法，  \* 而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。  \* 应用场景：实现一个接口中所有的方法时，可以创建一个抽象类AbstractClass，实现所有方法，写别类时继承抽象类即可  \*/  class InterfaceAdapter{  abstract class AbstractClass implements Targetable{  public void methodFirst() { }  public void methodSecond(){ }  }  class SourceSubFirst extends AbstractClass{  @Override  public void methodFirst() {  System.out.println("This a first method.");  }  }  class SourceSubSecond extends AbstractClass{  @Override  public void methodSecond() {  System.out.println("This a second method.");  }  }  public void run(){  Targetable targetable1 = new SourceSubFirst();  Targetable targetable2 = new SourceSubSecond();  targetable1.methodFirst();  targetable1.methodSecond();  targetable2.methodFirst();  targetable2.methodSecond();  }  } |

#### 装饰模式（Decorator）

对象的适配器

动态地将责任附加到对象上，若要扩展功能，装饰者提供了比继承更有弹性的替代方案。

|  |
| --- |
| /\*\*  \* 装饰模式  \* @author yaozou  \* @create 2018-04-20 14:30  \*\*/  public class DecoratorMethod {  public static void main(String[] args) {  // 基本思想：装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例  // Source类是被装饰类，Decorator类是一个装饰类，可以为Source类动态的添加一些功能  // 应用场景：1、需要扩展一个类的功能。  // 2、动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）  // 缺点：产生过多相似的对象，不易排错！  DecoratorMethod method = new DecoratorMethod();  method.run();  }  interface Sourceable{  void method();  }  class Source implements Sourceable{  public void method() {  System.out.println("This is a method");  }  }  class Decorator implements Sourceable{  private Sourceable source;  public Decorator(Sourceable source){  this.source = source;  }  public void method() {  System.out.println("Before Decorator......");  source.method();  System.out.println("After Decorator.......");  }  }  public void run(){  Sourceable source = new Source();  Decorator decorator = new Decorator(source);  decorator.method();  }  } |

#### 代理模式（Proxy）

对象的适配器

为另一个对象提供一个替身或占位符以控制对这个对象的访问。

|  |
| --- |
| /\*\*  \* 代理模式  \* @author yaozou  \* @create 2018-04-20 15:40  \*\*/  public class ProxyMethod {  public static void main(String[] args) {  // 多一个代理类出来，替原对象进行一些操作。比如：有的时候打官司，我们需要请律师，因为律师在法律方面有专长，可以替我们进行操作，表达我们的想法。  /\*\*  \* 应用场景：  \* 如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：  \* 1、修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则(开闭原则)。  \* 2、就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。  \* 使用代理模式，可以将功能划分的更加清晰，有助于后期维护！  \*/  ProxyMethod method = new ProxyMethod();  method.run();  }  interface Sourceable{  void method();  }  class Source implements Sourceable{  public void method() {  System.out.println("this is original method");  }  }  class Proxy implements Sourceable{  private Source source;  public Proxy(){  super();  this.source = new Source();  }  public void method() {  System.out.println("after proxy!");  source.method();  System.out.println("before proxy!");  }  }  public void run(){  Proxy proxy = new Proxy();  proxy.method();  }  } |

#### 外观模式（Facade）

对象的适配器

提供一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用。

|  |
| --- |
| /\*\*  \* 外观模式  \* @author yaozou  \* @create 2018-04-20 16:12  \*\*/  public class FacadeMethod {  public static void main(String[] args) {  /\*\*  \* 为了解决类与类之家的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，  \* 而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度，该模式中没有涉及到接口  \*/  // 例如打开电脑  FacadeMethod method = new FacadeMethod();  method.open();  method.close();  }  class CPUClass{  public void startup(){  System.out.println("CPU startup......");  }  public void stopdown(){  System.out.println("CPU stopdown......");  }  }  class MemoryClass{  public void startup(){  System.out.println("Memory startup......");  }  public void stopdown(){  System.out.println("Memory stopdown......");  }  }  class DiskClass{  public void startup(){  System.out.println("Disk startup......");  }  public void stopdown(){  System.out.println("Disk stopdown......");  }  }  class Computer{  private CPUClass cpu;  private MemoryClass memory;  private DiskClass disk;  public Computer(){  this.cpu = new CPUClass();  this.memory = new MemoryClass();  this.disk = new DiskClass();  }  public void startup(){  cpu.startup();  memory.startup();  disk.startup();  }  public void stopdown(){  cpu.stopdown();  memory.stopdown();  disk.stopdown();  }  }  public void open(){  Computer computer = new Computer();  computer.startup();  }  public void close(){  Computer computer = new Computer();  computer.stopdown();  }  } |

#### 桥接模式（Bridge）

使用桥接模式通过将实现和抽象放在两个不同的类层次中而使它们可以独立改变。

|  |
| --- |
| /\*\*  \* 桥接模式  \*/  public class BridgeMethod {  public static void main(String[] args) {  // 基本思路：桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化。  // 桥接的用意是：将抽象化与实现化解耦，使得二者可以独立变化，  /\*\*  \* 例如：常用的JDBC桥DriverManager一样，JDBC进行连接数据库的时候，在各个数据库之间进行切换，基本不需要动太多的代码，甚至丝毫不用动，  \* 原因就是JDBC提供统一接口，每个数据库提供各自的实现，用一个叫做数据库驱动的程序来桥接就行了  \*/  BridgeMethod method = new BridgeMethod();  method.run();  }  interface Driver{  void connection();  }  class MysqlDriver implements Driver{  public void connection() {  System.out.println("mysql is connecting......");  }  }  class OracleDriver implements Driver{  public void connection() {  System.out.println("oracle is connecting.......");  }  }  @Data  abstract class DriverManager{  private Driver driver;  public void connection(){  driver.connection();  }  }  class Client extends DriverManager{  public void connection(){  getDriver().connection();  }  }  public void run(){  Client client = new Client();  /\*调用第一个对象\*/  Driver mysqlDriver = new MysqlDriver();  client.setDriver(mysqlDriver);  client.connection();  /\*调用第二个对象\*/  Driver oracleDriver = new OracleDriver();  client.setDriver(oracleDriver);  client.connection();  }  } |

#### 组合模式（Composite）

允许你将对象组合成树形结构来表现"整体/部分"层次结构. 组合能让客户以一致的方式处理个别对象以及对象组合.

|  |
| --- |
| /\*\*  \* 组合模式  \* 部分-整体模式在处理类似树形结构的问题时比较方便  \*/  public class CompositeMethod {  public static void main(String[] args) {  }  @Data  class TreeNode{  private String name;  private TreeNode parent;  private Vector<TreeNode> children = new Vector<TreeNode>();  public TreeNode(String name){  this.name = name;  }  /\*添加孩子节点\*/  public void add(TreeNode node){  children.add(node);  }  /\*删除孩子节点\*/  public void remove(TreeNode node){  children.remove(node);  }  /\*取得孩子节点\*/  public Enumeration<TreeNode> getChildren(){  return children.elements();  }  }  class Tree{  TreeNode root = null;  public Tree(String name){  root = new TreeNode(name);  }  }  public void run(){  Tree tree = new Tree("A");  TreeNode nodeB = new TreeNode("B");  TreeNode nodeC = new TreeNode("C");  nodeB.add(nodeC);  tree.root.add(nodeB);  System.out.println("build the tree finished!");  }  } |

#### 享元模式（Flyweight）

如想让某个类的一个实例能用来提供许多的“虚拟实例”，就使用蝇量模式。

行为模式

|  |
| --- |
| /\*\*  \* 享元模式  \*/  public class FlyweightMethod {  public static void main(String[] args) {  /\*\*  \* 主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。  \* FlyWeightFactory负责创建和管理享元单元，当一个客户端请求时，工厂需要检查当前对象池中是否有符合条件的对象，  \* 如果有，就返回已经存在的对象，如果没有，则创建一个新对象，FlyWeight是超类。  \* 一提到共享池，我们很容易联想到Java里面的JDBC连接池，想想每个连接的特点，我们不难总结出：  \* 适用于作共享的一些个对象，他们有一些共有的属性，就拿数据库连接池来说，url、driverClassName、username、password及dbname，  \* 这些属性对于每个连接来说都是一样的，所以就适合用享元模式来处理，建一个工厂类，将上述类似属性作为内部数据，  \* 其它的作为外部数据，在方法调用时，当做参数传进来，这样就节省了空间，减少了实例的数量。  \*/  //通过连接池的管理，实现了数据库连接的共享，不需要每一次都重新创建连接，节省了数据库重新创建的开销，提升了系统的性能  ConnectionPool connectionPool = ConnectionPool.getInstance();  connectionPool.getConnection();  connectionPool.release();  }  }  class ConnectionPool{  private Vector<Connection> pool;  /\*公有属性\*/  private String url = "jdbc:mysql://localhost:3306/test";  private String username = "root";  private String password = "123456";  private String driverClassName = "com.mysql.jdbc.Driver";  private int poolSize = 100;  private static ConnectionPool instance = null;  Connection conn = null;  private ConnectionPool(){  pool = new Vector<Connection>(poolSize);  for (int i=0;i<poolSize;i++) {  try{  Class.forName(driverClassName);  conn = DriverManager.getConnection(url,username,password);  pool.add(conn);  } catch (ClassNotFoundException e) {  e.printStackTrace();  }catch (SQLException e){  e.printStackTrace();  }  }  }  private static synchronized ConnectionPool createPool(){  if (instance == null) instance = new ConnectionPool();  return instance;  }  public static ConnectionPool getInstance(){  if (instance == null) return createPool();  return instance;  }  /\*返回连接到连接池\*/  public synchronized void release(){  pool.add(conn);  }  /\* 返回连接池中的一个数据库连接 \*/  public synchronized Connection getConnection(){  if (pool.size() > 0){  Connection connection = pool.get(0);  pool.remove(connection);  return connection;  }else return null;  }  } |

# 9、MyBatis（源码重写）、Spring（源码重写）

# 10、数据结构算法

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 排序方法 | 平均情况 | 最好情况 | 最坏情况 | 辅助空间 | 稳定性 | 基本思想 |
| 插入排序 | 插入排序 | O(n2) | O(n) | O(n2) | O(1) | 稳定 |  |
| 二分排序 |  |  |  | O(1) | 稳定 |  |
| 希尔排序 |  |  |  | O(1) | 不稳定 |  |
| 交换排序 | 冒泡排序 |  |  |  | O(1) | 稳定 |  |
| 快速排序 |  |  |  | O(1) | 稳定 |  |
| 选择排序 | 直接选择排序 |  |  |  | O(1) | 稳定 |  |
| 堆排序 |  |  |  | O(1) | 不稳定 |  |
|  | 归并排序 |  |  |  | O(n) | 稳定 |  |
|  | 堆排序 |  |  |  | O(logn)~O(n) | 稳定 |  |

# 11、分布式