[1、Java基础](#_Toc3165)

[List的实现类，以及实现类的优缺点](#_Toc4405)

[Map的实现类，以及实现类的优缺点](#_Toc23384)

[Set的实现类，以及实现类的优缺点](#_Toc20957)

[Collection](#_Toc4786)

[关键字](#_Toc14714)

[引用](#_Toc17914)

[反射机制](#_Toc10205)

[cloneable接口实现原理](#_Toc3644)

[线程](#_Toc4216)

[简介](#_Toc7530)

[生命周期](#_Toc24725)

[线程的优先级](#_Toc10813)

[创建线程](#_Toc24635)

[sleep和wait的区别](#_Toc24590)

[数组](#_Toc11041)

[2、并发](#_Toc6766)

[synchronized](#_Toc2664)

[Volatile](#_Toc12935)

[信号量Semaphore](#_Toc31984)

[实现所有线程在等待某个事件的发生才去执行](#_Toc5267)

[CAS缺陷以及如何解决](#_Toc27521)

[AQS](#_Toc17892)

[死锁、活锁](#_Toc12070)

[线程池](#_Toc2130)

[ThreadLoad原理](#_Toc869)

[LockSupport工具](#_Toc28181)

[Condition原理](#_Toc2878)

[Fork/Join框架的理解](#_Toc2373)

[分段锁的原理、锁力度的减小思考](#_Toc1814)

[八种阻塞队列以及各个阻塞队列的特性](#_Toc31906)

[3、JVM](#_Toc2804)

[运行时数据区域、JMM（JVM内存模型）](#_Toc8014)

[Minor GC、Full GC](#_Toc16319)

[各种回收算法](#_Toc13213)

[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?](#_Toc20322)

[Finalize()方法](#_Toc28971)

[4、Redis](#_Toc23286)

[各命令及底层实现](#_Toc25813)

[远程 redis 服务上执行命令:](#_Toc4969)

[字符串(String)](#_Toc32074)

[哈希(Hash)](#_Toc4158)

[列表(List)](#_Toc4428)

[集合(Set)](#_Toc7506)

[有序集合](#_Toc21726)

[发布订阅](#_Toc16336)

[穿透、缓存雪崩](#_Toc7403)

[使用redis实现分布式锁](#_Toc29621)

[并发竞争问题](#_Toc3378)

[持久化的几种方式、优缺点、怎么实现](#_Toc28315)

[缓存失败策略](#_Toc27253)

[redis集群高可用原理](#_Toc22938)

[redis缓存分片（分区）](#_Toc23357)

[redis的数据淘汰策略](#_Toc17289)

[5、MySQL](#_Toc29626)

[分页优化](#_Toc21983)

[悲观锁、乐观锁、共享锁、排他锁、表锁、行锁](#_Toc3448)

[组合索引、最左原则](#_Toc27140)

[性能优化](#_Toc12813)

[索引分类：B+,Hash 什么情况用什么索引？](#_Toc18906)

[事务的四大特性](#_Toc22508)

[事务的隔离级别](#_Toc32185)

[事务的传播行为](#_Toc27383)

[6、Netty](#_Toc23358)

[BIO、NIO和AIO](#_Toc9871)

[TCP粘包/拆包的原因及解决方法](#_Toc25499)

[TCP的三次握手](#_Toc13613)

[7、Spring](#_Toc13635)

[BeanFactory、FactoryBean](#_Toc27088)

[IoC的理解及其初始化过程](#_Toc13355)

[BeanFactory和Application](#_Toc18880)

[Spring Bean生命周期如何被管理，加载过程](#_Toc25857)

[Spring AOP怎么实现](#_Toc4465)

[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](#_Toc6903)

[Spring中使用了哪些设计模式](#_Toc30100)

[Spring MVC的工作原理](#_Toc21789)

[Spring 循环注入的原理](#_Toc10645)

[Spring AOP各个术语、怎么相互工作](#_Toc20321)

[Spring 保证controller并发的安全](#_Toc16147)

[8、24种设计模式和7大原则](#_Toc11432)

[资料地址](#_Toc19994)

[7大原则](#_Toc2722)

[单一职责](#_Toc5154)

[开闭原则](#_Toc5444)

[里氏替换原则](#_Toc26664)

[依赖倒置原则](#_Toc7951)

[接口隔离原则](#_Toc5204)

[迪米特法则（最少知道原则）](#_Toc11401)

[合成（组合/聚合）复用原则](#_Toc27146)

[24种设计模式](#_Toc5274)

[创建者模式](#_Toc19294)

[结构模式](#_Toc9383)

[9、MyBatis（源码重写）、Spring（源码重写）](#_Toc32061)

[10、数据结构算法](#_Toc28620)

[11、分布式](#_Toc22604)

# 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Collections.synchronizedMap(new HashMap<>(8));

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 Hash（key）+i(i=1、2、3…)

开放定地址法——平方探查法 Hash（key）+ i^2（i= 1、2、3……不算第一放入的数）

（开链法）

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，不然会造成死锁 |
| 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。

|  |
| --- |
| **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。

|  |
| --- |
| class TestCountDownLatch{  public static void main(String[] args){  int num = 2;  final CountDownLatch latch = new CountDownLatch(num);  for(;num> 0;num--){  new Thread(){  public void run() {  try {  System.out.println("子线程"+Thread.currentThread().getName()+"正在执行");  Thread.sleep(3000);  System.out.println("子线程"+Thread.currentThread().getName()+"执行完毕");  latch.countDown();  } catch (InterruptedException e) {  e.printStackTrace();  }  };  }.start();  }  try {  System.*out*.println("等待2个子线程执行完毕...");  latch.await();  System.*out*.println("2个子线程已经执行完毕");  System.*out*.println("继续执行主线程");  } catch (InterruptedException e) {  e.printStackTrace();  }  } } |

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。

|  |
| --- |
| class TestSemaphore{  public static void main(String[] args){  // 8位工人  int n = 8;  // 5台机器  Semaphore semaphore = new Semaphore(5);  for(int i=0;i<n;i++){  new Worker(i,semaphore).start();  }  }  static class Worker extends Thread{  private int num;  private Semaphore semaphore;  public Worker(int num,Semaphore semaphore){  this.num = num;  this.semaphore = semaphore;  }   @Override  public void run() {  try {  semaphore.acquire();  System.*out*.println("工人"+this.num+"占用一个机器在生产...");  Thread.*sleep*(2000);  System.*out*.println("工人"+this.num+"释放出机器");  semaphore.release();  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } } |

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。

|  |
| --- |
| class TestCyclicBarrier{  public static void main(String[] args) {  int N = 4;  CyclicBarrier barrier = new CyclicBarrier(N);  for(int i=0;i<N;i++){  new Writer(barrier).start();  }  }  static class Writer extends Thread{  private CyclicBarrier cyclicBarrier;  public Writer(CyclicBarrier cyclicBarrier) {  this.cyclicBarrier = cyclicBarrier;  }  @Override  public void run() {  System.*out*.println("线程"+Thread.*currentThread*().getName()+"正在写入数据...");  try {  //以睡眠来模拟写入数据操作  Thread.*sleep*(5000);  System.*out*.println("线程"+Thread.*currentThread*().getName()+"写入数据完毕，等待其他线程写入完毕");  cyclicBarrier.await();  } catch (InterruptedException e) {  e.printStackTrace();  }catch(BrokenBarrierException e){  e.printStackTrace();  }   System.*out*.println("所有线程写入完毕，继续处理其他任务...");  }  } } |

## 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

资源地址：https://blog.csdn.net/wxq544483342/article/details/53118674

### 线程池的种类

1、创建一个缓冲池，缓冲池容量大小为Integer.MAX\_VALUE

ExecutorService pool = Executors.newCachedThreadPool();

2、创建容量为1的缓冲池

ExecutorService pool = Executors.*newSingleThreadExecutor*();

3、创建固定容量大小的缓冲池

ExecutorService pool = Executors.newFixedThreadPool(int);

4、创建一个定长线程池，支持定时及周期性任务执行。

ExecutorService pool = Executors.*newScheduledThreadPool*(1);

5、创建一个拥有多个任务队列（以便减少连接数）的线程池。

ExecutorService pool = Executors.*newWorkStealingPool*();

ExecutorService类的方法：

|  |
| --- |
| **void** shutdown();  *关闭线程池 不可以再 submit 新的 task，已经 submit 的将继续执行*  List<Runnable> shutdownNow();  *关闭线程池 试图停止当前正在执行的 task，并返回尚未执行的 task 的 list*  **boolean** isShutdown();  是否已经关闭了线程池，当调用shutdown()或shutdownNow()方法后返回为true。  **boolean** isTerminated();  当调用shutdown()方法后，并且所有提交的任务完成后返回为true;  当调用shutdownNow()方法后，成功停止后返回为true;  **boolean** awaitTermination(**long** timeout, TimeUnit unit) **throws** InterruptedException;  <T> Future<T> submit(Callable<T> task);  <T> Future<T> submit(Runnable task, T result);  Future<?> submit(Runnable task);  <T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks)  **throws** InterruptedException;  <T> List<Future<T>> invokeAll(Collection<? **extends** Callable<T>> tasks,  **long** timeout, TimeUnit unit)  **throws** InterruptedException;  <T> T invokeAny(Collection<? **extends** Callable<T>> tasks)  **throws** InterruptedException, ExecutionException;  <T> T invokeAny(Collection<? **extends** Callable<T>> tasks,  **long** timeout, TimeUnit unit)  **throws** InterruptedException, ExecutionException, TimeoutException; |

### 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

### 工作机制及原理

线程池的两个核心队列：

1. 线程等待池，即线程队列BlockingQueue
2. 任务处理池(PoolWorker)，即正在工作的Thread列表(HashSet)

线程池的核心参数：

1. 核心池大小（corePoolSize）,即固定大小，设定好之后，线程池的稳定峰值，达到这个值之后池的线程数大小不会释放。
2. 最大处理线程池数（maximumPoolSize）,当线程池里面的线程数超过corePoolSize，小于maximumPoolSize时会动态创建与回收线池里面的线程池资源。

### ThreadPoolExecutor类

构造方法：

|  |
| --- |
| */\*\*  \*  \** ***@param corePoolSize*** *核心池大小 默认情况下，在创建好线程池之后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，  \* 当线程池中的线程数量达到corePoolSize后，就会把这些任务放到缓存队列中  \** ***@param maximumPoolSize*** *线程池最大线程数量，表示在线程池中最多能创建线程的数量；在corePoolSize和maximumPoolSize的线程数会被自动释放，而小于corePoolSize的则不会。  \** ***@param keepAliveTime*** *表示线程没有执行任务时最多保持多久时间会终止。  \* 默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会生效,直到线程池数量不大于corePoolSize，  \* 即只有当线程池数量大于corePoolSize数量，超出这个数量的线程一旦到达keepAliveTime就会终止。  \* 但是如果调用了allowCoreThreadTimeout(boolean)方法，即使线程池的线程数量不大于corePoolSize，线程也会在keepAliveTime之后就终止，知道线程池的数量为0为止。  \** ***@param unit*** *参数keepAliveTime的时间单位，一个时间单位枚举类。 Nanos/Micros/Millis/Seconds/Minutes/Hours/Days  \** ***@param workQueue*** *一个阻塞队列，用来存储等待执行任务的队列，这个参数选择也很重要，会对线程池的运行过程产生重大影响。  \* 一般来说，这里的阻塞队列就是（ArrayBlockingQueue、LinkedBlockingQueue、SynchronousQueue）。  \** ***@param threadFactory*** *线程工厂，主要用来创建线程；可以是一个自定义的线程工厂，默认就是Executors.defaultThreadFactory()。用来在线程池中创建线程。  \** ***@param handler*** *表示当拒绝处理任务时的策略，也是可以自定义的，默认是我们前面的4种取值：  \* ThreadPoolExecutor.AbortPolicy（默认的，一言不合即抛异常的）  \* ThreadPoolExecutor.DiscardPolicy（一言不合就丢弃任务）  \* ThreadPoolExecutor.DiscardOldestPolicy（一言不合就把最近的任务给抛弃，然后执行当前任务）  \* ThreadPoolExecutor.CallerRunsPolicy（由调用者所在线程来执行任务）  \*/*  public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,  BlockingQueue<Runnable> workQueue, public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,  BlockingQueue<Runnable> workQueue, ThreadFactory threadFactory,RejectedExecutionHandler handler) {  if (corePoolSize < 0 || maximumPoolSize <= 0 || maximumPoolSize < corePoolSize || keepAliveTime < 0)  throw new IllegalArgumentException();  if (workQueue == null || threadFactory == null || handler == null)  throw new NullPointerException();  this.corePoolSize = corePoolSize;  this.maximumPoolSize = maximumPoolSize;  this.workQueue = workQueue;  this.keepAliveTime = unit.toNanos(keepAliveTime);  this.threadFactory = threadFactory;  this.handler = handler;  } |

## 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

LockSupport提供了park和unpark方法实现阻塞和解除阻塞，都是基于“许可（permit）”作为关联。permit相当于一个信号量(0,1),默认是0。

使用偏移量来获取对象。是因为线程已经被阻塞了，如果不通过内存的方式，直接调用线程内的方法，线程是不会响应的。

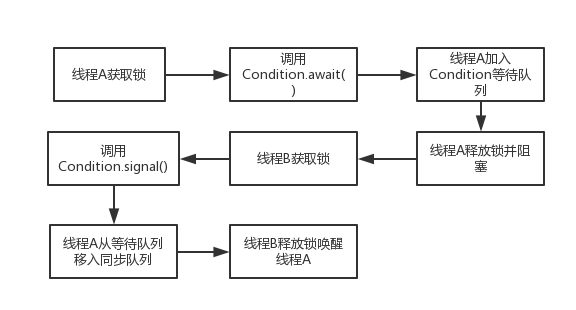
park()、unpark()和wait()、notify()的区别

1. 面向的主体不同。LockSupport是针对Thread进行阻塞，可以指定阻塞的对象，每次可以唤醒指定具体的线程。Wait()是以对象，阻塞当前的线程和唤醒单个线程或者所有线程。
2. 实现的机制不同。LockSupport可以指定monitor的object对象，但和object.wait()两者的阻塞队列并不交叉。

|  |
| --- |
| static void testLockSupport(){  // 调用native方法阻塞当前线程  LockSupport.*park*();  // 阻塞当前线程，最长不超过nanos纳秒，返回条件在park()的基础上增加了超时返回  LockSupport.*parkNanos*(1);  // 阻塞当前线程，知道deadline时间（deadline - 毫秒数）  LockSupport.*parkUntil*(1);   Object blocker = new Object();  //1) 记录当前线程等待的对象（阻塞对象）；  //2) 阻塞当前线程；  //3) 当前线程等待对象置为null。  LockSupport.*park*(blocker);  // 阻塞当前线程，最长等待时间不超过nanos毫秒，同样，在阻塞当前线程的时候做了记录当前线程等待的对象操作  LockSupport.*parkNanos*(blocker,1);  // 阻塞当前线程直到deadline时间，相同的，也做了阻塞前记录当前线程等待对象的操作  LockSupport.*parkUntil*(blocker,1);   // 唤醒处于阻塞状态的线程Thread  LockSupport.*unpark*(new Thread()); } |

## Condition原理

在经典的生产者和消费者模式中，可以使用Object.wait()和Object. Notify()阻塞和唤醒线程，但这样处理只有一个等待队列。在可重入锁ReentrantLock中，使用AQS的condition可以实现设置多个等待队列，可以使用lock.newCondition生成一个等待队列。



|  |
| --- |
| static void testCondition() throws InterruptedException {  // 需要结合lock使用  Lock lock = new ReentrantLock();  final Condition notEmpty = lock.newCondition();   // 将当前线程阻塞，调用await()之前必须先获取锁，  // 调用await()时，将线程构造成节点加入等待队列，同时释放锁，并挂起当前线程  notEmpty.await();  notEmpty.await(1, TimeUnit.*MINUTES*);  notEmpty.awaitNanos(1);  notEmpty.awaitUninterruptibly();  notEmpty.awaitUntil(new Date());   // 另一个线程将已经阻塞的线程唤醒  // 其他线程调用signal()时也必须获取锁，  // 当执行signal()方法时将等待队列的节点移入到同步队列，  // 当线程退出临界区释放锁时，唤醒同步队列的首个节点  notEmpty.signal();  notEmpty.signalAll(); } |

## Fork/Join框架的理解

并行流：把一个内容分成多个数据块，并用不同的线程分别处理每个数据块的流。（fork/join框架）

串行流：则相反

Fork/Join框架：

在必要的情况下，将一个大的任务进行拆分(fork)成若干个子任务（拆到临界值），再将一个个任务的结果进行join汇总。

Fork/Join与线程池的区别：

Fork/Join采用“工作窃取模式”，当执行新的任务时可以将其拆分成更小的任务执行，并将小任务加到线程队列中。然后再随机从一个线程中偷一个任务并把他加入到自己的队列中。

例如：CPU上有两个线程A/B，A已经执行完了，B还有任务未执行。这时A将B队尾的任务偷过来，加入到自己的队列中。相对于传统的线程池，Fork/Join更有效的使用了CPU资源。

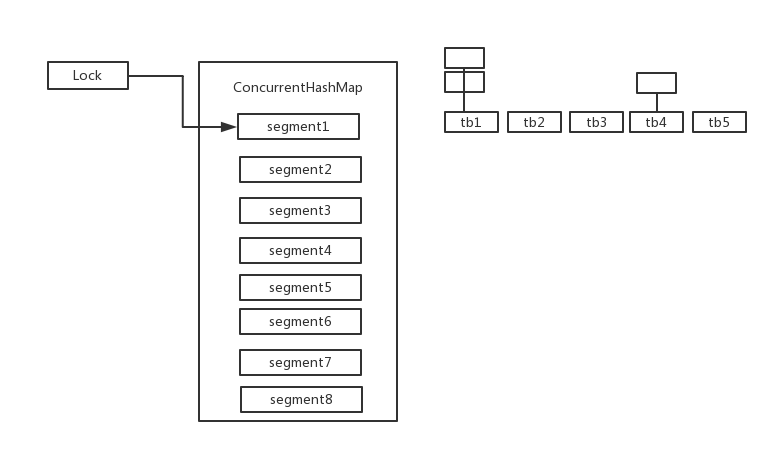
|  |
| --- |
| // ForkJoin这个框架的实现需要继承RecursiveTask 或者 RecursiveAction， // RecursiveTask是有返回值的，相反RecursiveAction则没有 @Data public class TestForkJoinWork extends RecursiveTask<Long> {  private Long start;  private Long end;  public static final Long *CRITICAL* = 10000L;   public TestForkJoinWork(Long start , Long end){  this.start = start;  this.end = end;  }   @Override  protected Long compute() {  Long length = end - start;  if (length <= *CRITICAL*){  Long sum = 0L;  for (Long i = start;i<=end;i++){  sum += i;  }  return sum;  }else{  Long middle = (start+end) / 2;  TestForkJoinWork right = new TestForkJoinWork(start,middle);  right.fork(); // 拆分 压入线程队列   TestForkJoinWork left = new TestForkJoinWork(middle+1,end);  left.fork(); // 拆分 压入线程队列   return right.join() + left.join();  }  } }  class TestForkJoinWorkDemo{   void test1(){  //Fork/Join实现  long l = System.*currentTimeMillis*();  ForkJoinPool forkJoinPool = new ForkJoinPool();//实现ForkJoin 就必须有ForkJoinPool的支持  TestForkJoinWork task = new TestForkJoinWork(0L,10000000000L);//参数为起始值与结束值  Long invoke = forkJoinPool.invoke(task);  long l1 = System.*currentTimeMillis*();  System.*out*.println("invoke = " + invoke+" time: " + (l1-l));  }   void test2(){  // 普通线程完成  Long x = 0L;  Long y = 10000000000L;  long l = System.*currentTimeMillis*();  for (Long i = 0L; i <= y; i++) {  x += i;  }  long l1 = System.*currentTimeMillis*();  System.*out*.println("invoke = " + x+" time: " + (l1-l));  }   void test3(){  // 并行流  long l = System.*currentTimeMillis*();  //parallel()与sequential()在并行流与串行流中随意切换  long reduce = LongStream.*rangeClosed*(0, 10000000000L).parallel().reduce(0, Long::*sum*);  long l1 = System.*currentTimeMillis*();  System.*out*.println("invoke = " + reduce+" time: " + (l1-l));  }   public static void main(String[] args){  TestForkJoinWorkDemo demo = new TestForkJoinWorkDemo();  demo.test1();  demo.test2();  demo.test3();  */\*\*  \* invoke = -5340232216128654848 time: 40213  \* invoke = -5340232216128654848 time: 52510  \* invoke = -5340232216128654848 time: 963  \*/* } } |

## 分段锁（JUC）的原理、锁力度的减小思考

容器中有多把锁，每一把锁用于锁容器中的一部分数据。当多线程时访问容器中的不同数据段的数据时，线程间就不会发生锁竞争了。因而有效的提高了高并发的访问效率。例如ConcurrentHashMap使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据一把锁，当一个线程占用锁访问其中一个段数据时，其他段的数据也可被其他线程访问。ConcurrentHashMap中使用了一个包含16个锁的数组，每个锁保护所在的散列桶的1/16，其中第N个散列桶由第（N mod 16）个锁来保护。合理的使用散列算法来使关键字均匀分布，那么可使对锁的请求减少到原来的1/16。这样可使ConcurrentHashMap的并发达到16个线程。

独占锁：只有一把锁，每次只能一个线程能访问。例如HashTable。采用串行方式，会降低其伸缩性。一般有三种方式降低锁的竞争：

1. 减少锁的持有时间
2. 降低锁的请求频率
3. 使用带有协调机制的独占锁，这些机制允许更高的并发性。



## 八种阻塞队列以及各个阻塞队列的特性

|  |
| --- |
| public class TestQueue {  public static void main(String[] args) throws InterruptedException {  Object obj = new Object();  boolean flag;  Object val;  // 由数组结构组成的有界阻塞队列  BlockingQueue arrayBlockingQueue = new ArrayBlockingQueue(10);  // 队未满时，返回true，队满时则抛出异常IllegalStateException  flag = arrayBlockingQueue.add(obj);  // 队未满时，返回true，队满时返回false。非阻塞立即返回  arrayBlockingQueue.offer(obj);  // 队未满时直接插入无返回值，队满时阻塞等待，一直等到队列未满时再插入  arrayBlockingQueue.put(obj);   // 队列不为空时，返回队首值并移除。队为空时抛出异常NoSuchElementException  val = arrayBlockingQueue.remove();  // 队列不为空时，返回队首值并移除。队为空时返回null。非阻塞立即返回  val = arrayBlockingQueue.poll();  // 设定等待的时间，如果在指定时间内队列为空则返回null，不为空则返回队首值  val = arrayBlockingQueue.poll(1,TimeUnit.*MINUTES*);  // 队列不为空返回队首值并移除；当队列为空时会阻塞等待，一直等到队列不为空时再返回队首值。  val = arrayBlockingQueue.take();   flag = arrayBlockingQueue.contains(obj);   // 由链表结构组成的有界阻塞队列  BlockingQueue linkedBlockingQueue = new LinkedBlockingQueue(10);  // 支持优先级别排序的无界阻塞队列 默认大小为11  BlockingQueue priorityBlockingQueue = new PriorityBlockingQueue();  // 使用有限级队列实现的无界阻塞队列 PriorityQueue  BlockingQueue delayQueue = new DelayQueue();  // 不存储元素的阻塞队列  BlockingQueue synchronousQueue = new SynchronousQueue();  // 由链表结构组成的无界阻塞队列  BlockingQueue linkedTransferQueue = new LinkedTransferQueue();  // 由链表结构组成的双阻塞队列  BlockingQueue linkedBlockingDeque = new LinkedBlockingDeque();  } |

# 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-清除算法一样，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

### 远程 redis 服务上执行命令:

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

### 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)。 |

### 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 |

### 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 |

### 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 |

### 有序集合

Redis 有序集合和集合一样也是string类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

有序集合的成员是唯一的,但分数(score)却可以重复。

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)。

|  |  |
| --- | --- |
| ZADD key score1 member1 [score2 member2] | 向有序集合添加一个或多个成员，或者更新已存在成员的分数 |
| ZCARD key | 获取有序集合的成员数 |
| ZCOUNT key min max | 计算在有序集合中指定区间分数的成员数 |

### 发布订阅

Redis 发布订阅(pub/sub)是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。

Redis 客户端可以订阅任意数量的频道。

频道Channel1，以及订阅这三个频道的客户端—— client2 、 client5 和 client1 之间的关系：



当有新消息通过 PUBLISH 命令发送给频道 channel1 时， 这个消息就会被发送给订阅它的三个客户端：



|  |  |
| --- | --- |
| SUBSCRIBE channel [channel ...] | 订阅给定的一个或多个频道的信息。 |
| PSUBSCRIBE pattern [pattern ...] | 订阅一个或多个符合给定模式的频道。 |
| UNSUBSCRIBE [channel [channel ...]] | 退订给定的频道。 |
| PUNSUBSCRIBE [pattern [pattern ...]] | 退订所有给定模式的频道。 |
| PUBLISH channel message | 将信息发送到指定的频道。 |
| PUBSUB subcommand [argument [argument ...]] | 查看订阅与发布系统状态。 |

## redis和memcache的区别

memcache是分布式高速缓存系统。

共同点：都是内存数据库

区别：

memcache:

1. 可以利用多核优势，单实例吞吐量极高，可以达到几十万QPS，适用于大数据量
2. 只支持简单的key/value数据结构，redis支持丰富的数据类型
3. 无法进行持久化，数据不能备份，只能用于缓存，重启后数据全部丢失

redis：

1. 支持多种数据结构，如string、list、set、zset、dict等
2. 单线程请求，所有命令串行执行，并发情况下不需要考虑数据一致性
3. 支持持久化，可以是使用AOF及RDB数据持久化到磁盘，从而进行数据的备份或数据恢复等操作，防止数据的丢失。
4. 支持通过Replication进行数据复制，通过master-slave机制，可以实现实时进行数据的同步复制，支持多级复制和增量复制。
5. 支持pub/sub消息订阅机制，可以用来进行消息订阅与通知
6. 支持简单的事务，使用场景很少，还不成熟

## 穿透、缓存雪崩

|  |  |
| --- | --- |
| 穿透 | 描述：  指在redis中查询一个一定不存在的数据时，需要从数据库中查询。查询到数据不存在时则不写入缓存，这将导致每次在redis中获取不存在的数据时都会去数据库中查询，这将导致缓存穿透。 |
| 解决方法：   1. 使用布隆过滤，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据就会被拦截，这样就可以避免对数据库的查询压力。 2. 如果在数据库查询的数据为空，不管何种原因造成的，仍然将空结果进行缓存，只是设置的过期时间很短，最长不超过5分钟。 |
| 缓存雪崩 | 描述：  Redis的缓存集中在一段时间内失效，发生大量的穿透，将所有的查询落到数据库中，造成了缓存雪崩。 |
| 解决方法：  1、缓存失效后，通过加锁或队列来控制数据库写缓存的线程数量。如一个key只允许一个线程查询数据和写缓存，其他线程等待。  2、可以通过缓存reload机制，预先去更新缓存，再即将发生大并发访问前手动触发加载缓存。  3、不同的key设置不同的过期失效时间，让缓存失效时间点尽量均匀。  4、做好二级缓存或者双缓存策略。A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期。 |

## 使用redis实现分布式锁

分布式锁一般有三种实现方式：1. 数据库乐观锁；2. 基于Redis的分布式锁；3. 基于ZooKeeper的分布式锁。

1、SETNX是set if not exist的缩写，如果不存在就返回保存value并返回1，如果存在就返回0。将key设置值为value，如果key不存在，这种情况下等同[SET](http://redis.cn/commands/set.html)命令。 当key存在时，什么也不做。

2、GETSET其实就是两个指令GET和SET，首先会GET到当前key的值并返回，然后在设置当前Key为要设置Value。

|  |
| --- |
| **public class** RedisLock {  **private static final** String ***LOCK\_SUCCESS*** = **"OK"**;  **private static final** String ***SET\_IF\_NOT\_EXIST*** = **"NX"**;  **private static final** String ***SET\_WITH\_EXPIRE\_TIME*** = **"PX"**;  **private final** JedisPool **jedisPool**;   **public** RedisLock(JedisPool jedisPool) {  **this**.**jedisPool** = jedisPool;  }   */\*\*  \* 尝试获取分布式锁  \** ***@param jedis*** *Redis客户端  \** ***@param lockKey*** *锁  \** ***@param requestId*** *请求标识  \** ***@param expireTime*** *超期时间  \** ***@return*** *是否获取成功  \*/* **public static boolean** tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, **int** expireTime) {   String result = jedis.set(lockKey, requestId, ***SET\_IF\_NOT\_EXIST***, ***SET\_WITH\_EXPIRE\_TIME***, expireTime);   **if** (***LOCK\_SUCCESS***.equals(result)) {  **return true**;  }  **return false**;   }   */\*\*  \* 加锁  \** ***@param locaName*** *锁的key  \** ***@param acquireTimeout*** *获取超时时间  \** ***@param timeout*** *锁的超时时间  \** ***@return*** *锁标识  \*/* **public** String lockWithTimeout(String locaName,  **long** acquireTimeout, **long** timeout) {  Jedis conn = **null**;  String retIdentifier = **null**;  **try** {  *// 获取连接* conn = **jedisPool**.getResource();  *// 随机生成一个value* String identifier = UUID.*randomUUID*().toString();  *// 锁名，即key值* String lockKey = **"lock:"** + locaName;  *// 超时时间，上锁后超过此时间则自动释放锁* **int** lockExpire = (**int**)(timeout / 1000);   *// 获取锁的超时时间，超过这个时间则放弃获取锁* **long** end = System.*currentTimeMillis*() + acquireTimeout;  **while** (System.*currentTimeMillis*() < end) {  **if** (conn.setnx(lockKey, identifier) == 1) {  conn.expire(lockKey, lockExpire);  *// 返回value值，用于释放锁时间确认* retIdentifier = identifier;  **return** retIdentifier;  }  *// 返回-1代表key没有设置超时时间，为key设置一个超时时间* **if** (conn.ttl(lockKey) == -1) {  conn.expire(lockKey, lockExpire);  }   **try** {  Thread.*sleep*(10);  } **catch** (InterruptedException e) {  Thread.*currentThread*().interrupt();  }  }  } **catch** (JedisException e) {  e.printStackTrace();  } **finally** {  **if** (conn != **null**) {  conn.close();  }  }  **return** retIdentifier;  }   */\*\*  \* 释放锁  \** ***@param lockName*** *锁的key  \** ***@param identifier*** *释放锁的标识  \** ***@return*** *\*/* **public boolean** releaseLock(String lockName, String identifier) {  Jedis conn = **null**;  String lockKey = **"lock:"** + lockName;  **boolean** retFlag = **false**;  **try** {  conn = **jedisPool**.getResource();  **while** (**true**) {  *// 监视lock，准备开始事务* conn.watch(lockKey);  *// 通过前面返回的value值判断是不是该锁，若是该锁，则删除，释放锁* **if** (identifier.equals(conn.get(lockKey))) {  Transaction transaction = conn.multi();  transaction.del(lockKey);  List<Object> results = transaction.exec();  **if** (results == **null**) {  **continue**;  }  retFlag = **true**;  }  conn.unwatch();  **break**;  }  } **catch** (JedisException e) {  e.printStackTrace();  } **finally** {  **if** (conn != **null**) {  conn.close();  }  }  **return** retFlag;  } } |

## 并发竞争问题

Redis的并发竞争问题，主要是发生在并发写竞争。

1. 可以使用独占锁的方式，类似操作系统的mutex机制。
2. 使用乐观锁的方式进行解决（成本较低，非阻塞，性能较高）。

|  |
| --- |
| watch stock  get stock $stock  $stock = $stock - 10  multi  set stock $stock  exec |

watch这里表示监控该key值，后面的事务是有条件的执行，如果从watch的exec语句执行时，watch的key对应的value值被修改了，则事务不会执行。

|  |  |
| --- | --- |
| WATCH key [key ...] | 监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。 |
| UNWATCH | 取消 WATCH 命令对所有 key 的监视。 |
| MULTI | 标记一个事务块的开始。 |
| EXEC | 执行所有事务块内的命令。 |
| DISCARD | 取消事务，放弃执行事务块内的所有命令。 |

## 持久化的几种方式、优缺点、怎么实现

1. RDB持久化：将Reids在内存中的数据库记录定时dump到磁盘上的RDB持久化。

配置：

通过配置文件来修改Redis服务器dump快照的频率，在打开6379.conf文件之后，搜索save

|  |
| --- |
| save 900 1  #在900秒(15分钟)之后，如果至少有1个key发生变化，则dump内存快照。  save 300 10 #在300秒(5分钟)之后，如果至少有10个key发生变化，则dump内存快照。  save 60 10000 #在60秒(1分钟)之后，如果至少有10000个key发生变化，则dump内存快照。 |

1. AOF（append only file）持久化：原理是将Reids的操作日志以追加的方式写入文件。

配置：

在Redis的配置文件中存在三种同步方式，它们分别是：

|  |
| --- |
| appendfsync always  #每次有数据修改发生时都会写入AOF文件。  appendfsync everysec  #每秒钟同步一次，该策略为AOF的缺省策略。  appendfsync no     #从不同步。高效但是数据不会被持久化。 |

## 缓存失败策略

穿透和缓存雪崩

## redis集群高可用原理

两种方式：

1. 主备方式

一台主机、一台或多台备机，在正常情况下主机对外提供服务，并把数据同步到备机，当主机宕机后，备机立刻开始服务。

优点：对客户端无影响

缺点：绝大数情况下备机没有使用。一直空闲浪费。

1. 主从方式

主从之间进行数据同步。 当Master宕机后，通过选举算法(Paxos、Raft)从slave中选举出新Master继续对外提供服务，主机恢复后以slave的身份重新加入。

优点：另一个目的是进行读写分离，这是当单机读写压力过高的一种通用型解决方案。 其主机的角色只提供写操作或少量的读，把多余读请求通过负载均衡算法分流到单个或多个slave服务器上。

缺点：主机宕机后，Slave虽然被选举成新Master了，但对外提供的IP服务地址却发生变化了，意味着会影响到客户端。 解决这种情况需要一些额外的工作，在当主机地址发生变化后及时通知到客户端，客户端收到新地址后，使用新地址继续发送新请求

数据同步方式：

1、异步

2、同步

## redis缓存分片（分区）

分区是分割数据到多个Redis实例的处理过程，因此每个实例只保存key的一个子集。

|  |  |
| --- | --- |
| 优点 | 缺点 |
| 1. 利用多台计算机内存的和值，允许我们构造更大的数据库。 2. 通过多核和多台计算机，允许我们扩展计算能力；通过多台计算机和网络适配器，允许我们扩展网络带宽。 | 1. 涉及多个key的操作通常是不被支持的。举例来说，当两个set映射到不同的redis实例上时，你就不能对这两个set执行交集操作。 2. 涉及多个key的redis事务不能使用。 3. 当使用分区时，数据处理较为复杂，比如你需要处理多个rdb/aof文件，并且从多个实例和主机备份持久化文件。 4. 增加或删除容量也比较复杂。redis集群大多数支持在运行时增加、删除节点的透明数据平衡的能力，但是类似于客户端分区、代理等其他系统则不支持这项特性。然而，一种叫做presharding的技术对此是有帮助的。 |

分区类型：

1. 范围分区

最简单的分区方式是按范围分区，映射一定范围的对象到特定的Redis实例

比如，ID从0到10000的用户会保存到实例R0，ID从10001到 20000的用户会保存到R1，以此类推。

这种方式是可行的，并且在实际中使用，不足就是要有一个区间范围到实例的映射表。这个表要被管理，同时还需要各 种对象的映射表，通常对Redis来说并非是好的方法。

1. 哈希分区

用一个hash函数将key转换为一个数字，比如使用crc32 hash函数。对key foobar执行crc32(foobar)会输出类似93024922的整数。

对这个整数取模，将其转化为0-3之间的数字，就可以将这个整数映射到4个Redis实例中的一个了。93024922 % 4 = 2，就是说key foobar应该被存到R2实例中。注意：取模操作是取除的余数，通常在多种编程语言中用%操作符实现。

## redis的数据淘汰策略

redis内存的数据集大小到一定大小时，会实施数据淘汰策略。Redis有6种数据淘汰策略：

1. volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰。
2. volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
3. volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰。
4. allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰。
5. allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰。
6. no-enviction（驱逐）：禁止驱逐数据。

# 5、MySQL

## 分页优化

mysql大数据量使用limit分页，随着页码的增大，查询效率越低下。

1. 直接使用limit

语句：SELECT \* FROM 表名 LIMIT M,N;

场景：适用于数据量较少的情况(元组百/千级)

缺点：全表扫描,速度会很慢且有的数据库结果集返回不稳定

1. 建立主索引或唯一索引（假设每页10条）

语句：SELECT \* FROM 表名称 WHERE id\_pk > (pageNum\*10) LIMIT M;

场景：适用于数据量多的情况(元组数上万)

缺点：索引扫描,速度会很快。可能查询的数据不是按id\_pk排序的，因此可能有漏掉数据的情况。只能使用方法3

1. 基于索引再排序

语句：SELECT \* FROM 表名称 WHERE id\_pk > (pageNum\*10) ORDER BY id\_pk ASC LIMIT M;

场景：适用于数据量多的情况(元组数上万). 最好ORDER BY后的列对象是主键或唯一所以,使得ORDERBY操作能利用索引被消除但结果集是稳定的。

优点：索引扫描,速度会很快。

1. 基于索引使用prepare

语句：PREPARE stmt\_name FROM SELECT \* FROM 表名称 WHERE id\_pk > (？\* ？) ORDER BY id\_pk ASC LIMIT M；

场景：大数据量

优点：索引扫描,速度会很快. prepare语句又比一般的查询语句快一点。

1. 利用MySQL支持ORDER操作可以利用索引快速定位部分元组,避免全表扫描

语句：SELECT \* FROM your\_table WHERE pk>=1000 ORDER BY pk ASC LIMIT 0,20；（读第1000到1019行数据）

场景：大数据量

1. 利用"子查询/连接+索引"快速定位元组的位置,然后再读取元组. 道理同方法5

语句：SELECT \* FROM your\_table WHERE id <=

(SELECT id FROM your\_table ORDER BY id desc LIMIT ($page-1)\*$pagesize ORDER BY id desc LIMIT $pagesize;

场景：大数据量

## 悲观锁、乐观锁、共享锁、排他锁、表锁、行锁

|  |  |
| --- | --- |
| 悲观锁 | 悲观锁就是在操作数据时，认为此操作会出现数据冲突，所以在进行每次操作时都要通过获取锁才能进行对相同数据的操作，这点跟java中的synchronized很相似，所以悲观锁需要耗费较多的时间。悲观锁是由数据库自己实现了的，要用的时候，我们直接调用数据库的相关语句就可以了。  共享锁和排它锁是悲观锁的不同的实现，它俩都属于悲观锁的范畴。 |
| 乐观锁 | 乐观锁是指操作数据库时(更新操作)，想法很乐观，认为这次的操作不会导致冲突，在操作数据时，并不进行任何其他的特殊处理（也就是不加锁），而在进行更新后，再去判断是否有冲突了。  例如：下单操作包括3步骤：  1.查询出商品信息  select status,status,version from t\_goods where id=#{id}  2.根据商品信息生成订单  3.修改商品status为2  update t\_goods set status=2,version=version+1  where id=#{id} and version=#{version}; |

|  |  |
| --- | --- |
| 共享锁 | 对于多个不同的事务，对同一个资源共享同一个锁。通过在执行语句后面加上lock in share mode就代表对某些资源加上共享锁了。 |
| 排他锁 | 对于update,insert,delete语句会自动加排它锁。指对于多个不同的事务，对同一个资源只能有一把锁。在需要执行的语句后面加上for update |

|  |  |
| --- | --- |
| 行锁 | 给某一行加上锁，也就是一条记录加上锁。  例如：  SELECT \* from city where id = "1" lock in share mode;  由于对于city表中,id字段为主键，就也相当于索引。执行加锁时，会将id这个索引为1的记录加上锁，那么这个锁就是行锁。 |
| 表锁 | 给这个表加上锁。 |

## 组合索引、最左原则

索引：单列索引、组合索引

主键索引：

主键索引是唯一索引的特定类型。

唯一索引：

唯一索引是不允许其中任何两行具有相同索引值的索引。可防止添加将在表中创建重复键值的新数据。

聚集索引：

在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。

索引列：

基于数据库表中的单列或多列创建索引。多列索引可以区分其中一列可能有相同值的行。

CREATE INDEX indexName ON mytable(username(length));

ALTER table tableName ADD INDEX indexName(columnName);

CREATE TABLE mytable(

ID INT NOT NULL,

username VARCHAR(16) NOT NULL,

INDEX [indexName] (username(length))

);

DROP INDEX [indexName] ON mytable;

唯一索引

CREATE UNIQUE INDEX indexName ON mytable(username(length));

显示索引：

SHOW INDEX FROM table\_name;

最左原则：例如组合索引是orderId,creatTime

语句1：Select \* from order\_info where order\_id=001;

语句2：Select \* from order\_info where order\_id=001 and creatTime=2018-08-01;

语句3：Select \* from order\_info where creatTime=2018-08-01;

性能：语句1=语句2>语句3

也就是索引字段的数据必须是有序的，才能实现这种类型的查找，才能利用到索引。

## 性能优化

对于sql的优化、尽量减少对数据库的链接

## 索引分类：B+,Hash 什么情况用什么索引？

索引分为B+索引和哈希索引

B+索引比哈希索引的检索效率高，但哈希索引有很多的限制和弊端：

1. Hash 索引仅仅能满足”=”,”IN”和”<=>”查询，不能使用范围查询。
2. Hash 索引无法被用来避免数据的排序操作。
3. Hash 索引不能利用部分索引键查询。
4. Hash 索引在任何时候都不能避免表扫描。
5. Hash 索引遇到大量Hash值相等的情况后性能并不一定就会比B-Tree索引高。
6. B+完全基于key的比较，和二叉树相同，相当于建个排序后的数据集，使用二分法查找算法，实际上也非常快，而且受数据量增长影响非常小。

## 事务的四大特性（ACID）

原子性（Atomicity）：要么都执行，要么都不执行。

一致性（Comisteny）：当事务完成时，数据必须处于一致状态。

隔离性（Isolation）：对数据进行修改的所有并发事务是彼此隔离的，这表明事务必须是独立的它不应以任何方式依赖于或影响其他事务。

持久性（Durability）：事务完成后，它对数据库的修改被永久保存，事务日志能保持事务的永久性。

## 事务的隔离级别

1. Read uncommitted

一个事务可以读取另一个未提交事务的数据

1. Read committed

一个事务要等另一个事务提交后才能读取数据

1. Repeatable read

在开始读取数据（事务开启）时，不再允许修改操作

1. Serializable

事务串行化顺序执行，可以避免脏读，不可重复度与幻读。这种事务隔离级别效率低下，比较耗数据库性能，一般不使用。

## 事务的传播行为

1. propagation\_required

如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务。该设置是最常用的设置

1. Propagation\_supports

支持当前事务，如果当前存在事务，就加入事务。如果当前不存在事务，就抛出异常。

1. Propagation\_mandatory

支持当前事务，如果当前存在事务，就加入事务。如果当前不存在事务，就抛出异常。

1. Propagation\_required\_new

创建新事务，无论当前存不存在事务都创建事务

1. Propagation\_not\_supported

以非事务方式执行，如果当前存在事务，就把当前事务挂起。

1. Propagation\_never

以非事务方式执行，如果当前存在事务，则抛出异常。

1. Propagation\_nested
2. 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与propagation\_required类似的操作。

# 6、Netty

## BIO、NIO和AIO

参考地址：https://www.cnblogs.com/ygj0930/p/6543960.html

BIO：同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

NIO：同步非阻塞，NIO方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。NIO还提供了两个新概念：Buffer和Channel

AIO：异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理，适用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持

## TCP粘包/拆包的原因及解决方法

问题产生

一个完整的业务可能会被TCP拆分成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送，这个就是TCP的拆包和封包问题。

下面可以看一张图，是客户端向服务端发送包：



1. 第一种情况，Data1和Data2都分开发送到了Server端，没有产生粘包和拆包的情况。

2. 第二种情况，Data1和Data2数据粘在了一起，打成了一个大的包发送到Server端，这个情况就是粘包。

3. 第三种情况，Data2被分离成Data2\_1和Data2\_2，并且Data2\_1在Data1之前到达了服务端，这种情况就产生了拆包。

由于网络的复杂性，可能数据会被分离成N多个复杂的拆包/粘包的情况，所以在做TCP服务器的时候就需要首先解决拆包/粘包的问题。

TCP粘包和拆包产生的原因

1. 应用程序写入数据的字节大小大于套接字发送缓冲区的大小

2. 进行MSS大小的TCP分段。MSS是最大报文段长度的缩写。MSS是TCP报文段中的数据字段的最大长度。数据字段加上TCP首部才等于整个的TCP报文段。所以MSS并不是TCP报文段的最大长度，而是：MSS=TCP报文段长度-TCP首部长度

3. 以太网的payload大于MTU进行IP分片。MTU指：一种通信协议的某一层上面所能通过的最大数据包大小。如果IP层有一个数据包要传，而且数据的长度比链路层的MTU大，那么IP层就会进行分片，把数据包分成若干片，让每一片都不超过MTU。注意，IP分片可以发生在原始发送端主机上，也可以发生在中间路由器上。

TCP粘包和拆包的解决策略

1. 消息定长。例如100字节。

2. 在包尾部增加回车或者空格符等特殊字符进行分割，典型的如FTP协议

3. 将消息分为消息头和消息尾。

4. 其它复杂的协议，如RTMP协议等。

## TCP的三次握手

tcp/ip的3次握手, 简单来说就是第一次我连接你给你一个标识SYN,你给我返回SYN并给一个新的ACK标记我,然后我再把ACK给你，这样证明我们之前传东西是可靠的然后就正式传数据了。

tcp/ip的4次挥手断开,相当于,你给我一个ACK我给你一个FIN,然后再次彼此交换确认,OK就可以结束通信了



# 7、Spring

## Spring架构图



## BeanFactory、FactoryBean

1. FactoryBean是一个Bean但又不是一个简单的Bean，而是一个能产生或者修饰对象生成的工厂Bean，他的实现与设计模式中的工厂模式和修饰器模式类似。
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。在Spring中所有的Bean都是由BeanFactory（IoC容器）来进行管理的。

## IoC的理解及其初始化过程

IoC的理解：<http://jinnianshilongnian.iteye.com/blog/1413846>

BeanDefinition的Resource定位，载入和注册三个过程

第一个过程是Resource定位：

这个Resource的定位是指BeanDefinition的资源定位，它是由ResourceLoader通过统一的Resource接口来完成，这个Resource对各种形式的BeanDefinition的使用都提供统一接口。这个定位过程类似于容器寻找数据的过程，就像水桶装水先把水找到一样。

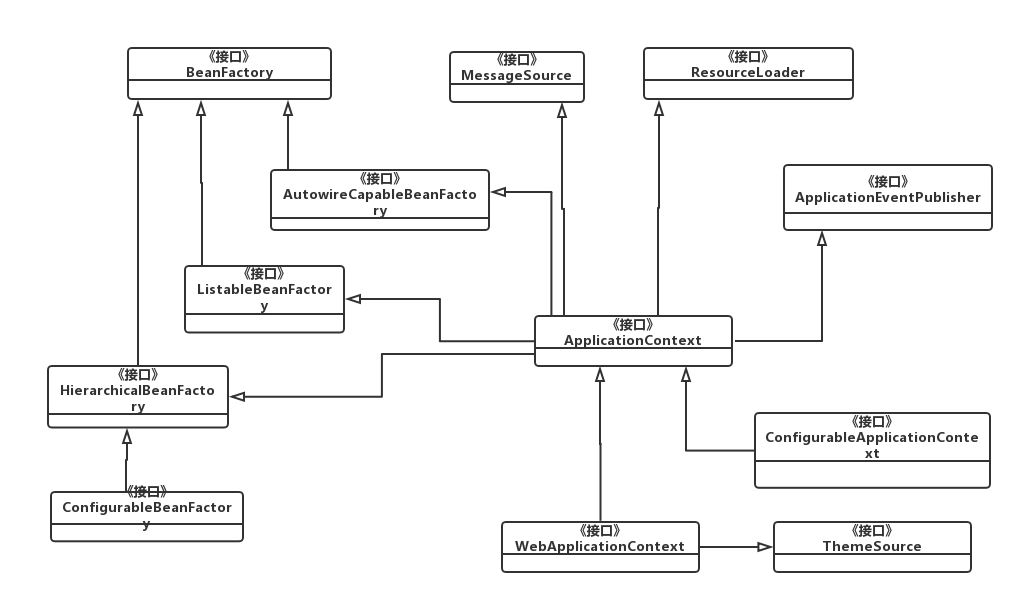
第二个过程是BeanDefinition的载入：

把用户定义好的Bean表示成IoC容器内部的数据结构，而这个容器内部的数据结构就是BeanDefinition。具体来说，这个BeanDefinition实际上就是POJO对象在IoC容器中的抽象，通过这个BeanDefinition定义的数据结构，使IoC容器能够方便地对POJO对象也就是Bean进行管理。

第三个过程是向IoC容器注册这些BeanDefinition的过程：

这个过程通过调用BeanDefinitionRegistry接口的实现来完成的。这个注册过程把载入过程中解析得到的BeanDefinition向IoC容器进行注册。在IoC容器内部将BeanDefinition注入到一个HashMap中去，IoC容器通过这个HashMap来持有这些BeanDefinition数据的。

## BeanFactory和Application



IoC容器的接口设计图

1. 从BeanFactory到HierarchicalBeanFactory，再到ConfigurableBeanFactory，是一条主要的BeanFactory设计路径。通过这些接口设计的叠加，定义了BeanFactory就是简单IoC的基本功能。

BeanFactory接口定义了基本的IoC容器的规范。这个接口定义中，包括了getBean()通过这个方法可以从容器中取得Bean。

HierachicalBeanFactory接口在继承了BeanFactory的基本接口之后，增加了getParentBeanFactory()的接口功能，使BeanFactory具备了双亲IoC容器的管理功能。

ConfigurableBeanFactory接口中，主要定义了一些对BeanFactory的配置功能。比如通过setBeanPostProcessor()配置Bean后置处理器，通过setParentBeanFactory()设置双亲IoC容器的管理功能。

1. 以ApplicationContext应用上下文接口为核心的接口设计，从BeanFactory到ListableBeanFactory，再到ApplicationContext，再到WebApplicationContext或者ConfigurableApplicationContext接口。

这个接口体现在HierarchicalBeanFactory和ListableBeanFactory连接BeanFactory接口定义和ApplicationContext应用上下文的接口定义。ListableBeanFactory接口细化了很多BeanFactory的接口功能，比如定义了getBeanDefinitionNames()。

ApplicationContext接口，它通过继承MessageResource、ResourceLoader、ApplicationEventPublisher接口，在BeanFactory简单IoC容器的基础上添加了许多对高级容器的特性的支持。

## 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

### 如何管理事务

Spring并不直接管理事务，而是提供了多种事务管理器，他们将事务管理的职责委托给Hibernate或者JTA等持久化机制所提供的相关平台框架的事务来实现。

Spring事务管理器的接口是org.springframework.transaction.PlatformTransactionManager，通过这个接口，Spring为各个平台如JDBC、Hibernate等都提供了对应的事务管理器，但是具体的实现就是各个平台执行。

1. JDBC事务

如果应用程序中直接使用JDBC来进行持久化，DataSourceTransactionManager会为你处理事务边界。为了使用DataSourceTransactionManager，你需要使用如下的XML将其装配到应用程序的上下文定义中：

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource" />

</bean>

实际上，DataSourceTransactionManager是通过调用java.sql.Connection来管理事务，而后者是通过DataSource获取到的。通过调用连接的commit()方法来提交事务，同样，事务失败则通过调用rollback()方法进行回滚。

1. Hibernate事务

如果应用程序的持久化是通过Hibernate实习的，那么你需要使用HibernateTransactionManager。对于Hibernate3，需要在Spring上下文定义中添加如下的<bean>声明：

<bean id="transactionManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">

<property name="sessionFactory" ref="sessionFactory" />

</bean>

sessionFactory属性需要装配一个Hibernate的session工厂，HibernateTransactionManager的实现细节是它将事务管理的职责委托给org.hibernate.Transaction对象，而后者是从Hibernate Session中获取到的。当事务成功完成时，HibernateTransactionManager将会调用Transaction对象的commit()方法，反之，将会调用rollback()方法。

1. Java持久化API事务（JPA）

Hibernate多年来一直是事实上的Java持久化标准，但是现在Java持久化API作为真正的Java持久化标准进入大家的视野。如果你计划使用JPA的话，那你需要使用Spring的JpaTransactionManager来处理事务。你需要在Spring中这样配置JpaTransactionManager：

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">

<property name="sessionFactory" ref="sessionFactory" />

</bean>

JpaTransactionManager只需要装配一个JPA实体管理工厂（javax.persistence.EntityManagerFactory接口的任意实现）。JpaTransactionManager将与由工厂所产生的JPA EntityManager合作来构建事务。

1. Java原生API事务

如果你没有使用以上所述的事务管理，或者是跨越了多个事务管理源（比如两个或者是多个不同的数据源），你就需要使用JtaTransactionManager：

<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManager">

<property name="transactionManagerName" value="java:/TransactionManager" />

</bean>

JtaTransactionManager将事务管理的责任委托给javax.transaction.UserTransaction和javax.transaction.TransactionManager对象，其中事务成功完成通过UserTransaction.commit()方法提交，事务失败通过UserTransaction.rollback()方法回滚。

### 不同传播行为有哪些，有何用

1、propagation\_required

如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务。该设置是最常用的设置

2、Propagation\_supports

支持当前事务，如果当前存在事务，就加入事务。如果当前不存在事务，就抛出异常。

3、Propagation\_mandatory

支持当前事务，如果当前存在事务，就加入事务。如果当前不存在事务，就抛出异常。

4、Propagation\_required\_new

创建新事务，无论当前存不存在事务都创建事务

5、Propagation\_not\_supported

以非事务方式执行，如果当前存在事务，就把当前事务挂起。

6、Propagation\_never

以非事务方式执行，如果当前存在事务，则抛出异常。

7、Propagation\_nested

8、如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与propagation\_required类似的操作。

### 隔离级别有哪些，有何用

\* Isolation.READ\_UNCOMMITTED 一个事务可以读取别一个未提交事务的数据

\* Isolation.READ\_COMMITTED 一个事务要等别一个事务提交后台可能读取数据 （这就是读提交，若有事务对数据进行更新（UPDATE）操作时，读操作事务要等待这个更新操作事务提交后才能读取数据，可以解决脏读问题。但不可解决不可重复读。）

\* Isolation.REPEATABLE\_READ 在开始读取数据（事务开启）时，不再允许修改操作 （可以解决不可重复读问题，不可重复读对应的是修改，即UPDATE操作。但是可能还会有幻读问题。因为幻读问题对应的是插入INSERT操作，而不是UPDATE操作。）

\* Isolation.SERIALIZABLE 事务串行化顺序执行，可以避免脏读，不可重复读与幻读

## 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)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

## 绪论

### 逻辑结构类型

集合

线性结构（一对一）：除开始结点和终端结点以外，其余结点都有且仅有一个直接前驱。有且仅有一个直接后继。顺序表是一个典型的线性结构

树形结构（一对多）：每个结点最多只有一个直接前驱，但可以有多个直接后继，可以有多个终端结点。二叉树就是一个典型的树形结构。

图形结构（多对多）：每个结点的直接前驱和直接后继的个数都可以是任意的。

树形结构和图形结构都是非线性结构

### 存储结构类型

顺序存储结构：把逻辑上相邻的结点存储在物理位置上相邻的存储单元里。主要优点：节省存储空间、实现对结点的随机存取。主要缺点：不便于修改，对结点的插入、删除运算时，可能要移动一系列的结点。

链式存储结构：不要求逻辑上相邻的结点在物理位置上也相邻，结点间的逻辑关系是由附加的指针字段表示的。主要优点：便于修改，在进行插入、删除运算时，仅需修改相应结点的指针域，不必移动结点。主要缺点：存储空间的利用率较低、不能对结点进行随机存储。

索引存储结构：在存储结点信息的同时还建立附加的索引表。一般形式是：(关键字，地址)，关键字唯一标识一个结点，地址作为指向结点的指针。优点：可以大大提高数据的查找的速度、可以对结点进行随机访问、在进行插入、删除运算时，只需移动存储在索引表中对应结点的存储地址。缺点：增加了索引表，降低了存储空间的利用率。

哈希（或散列）存储结构：根据结点的关键字通过哈希函数直接计算一个值，并将这个值作为该结点的存储地址。优点：查找速度快。哈希存储方法一般只适合要求对数据能够进行快速查找和插入的场合。

### 算法效率分析

时间复杂度

算法的执行时间大致为基本运算所需的时间与其运算次数（一个语句的运行次数称为语句频度）的乘积。

算法中基本运算次数T(n)是问题规模n的某个函数f(n)，记做：

T(n)=O(f(n))

O(1)<O(log2n)< O(n)< O(nlog2n)< O(n2)< O(2n)< O(n!)

例1：下面程序段的时间复杂度是—O(n\*m)—

|  |
| --- |
| for(i=0;i<n;i++){  for(j=0;j<m;j++){  A[i][j]=0;  }  } |

例2：下面程序段的时间复杂度是—O()—

|  |
| --- |
| i=s=0;  while(s < n){  i++;  s += i;  } |

例3：下面程序段的时间复杂度是—O(log3n)—

|  |
| --- |
| i= 1;  while(i <= n) i = i \*3; |

### 算法存储空间分析

空间复杂度

对一个算法在运行过程中临时占用的存储空间大小的量度。记做：

S(n) = O(g(n))

## 线性表

### 简介

线性表具有相同特性的数据元素的一个有限序列。

### 顺序存储结构

顺序表：把线性表中的所有元素按照其逻辑顺序依次存储在计算机存储器中指定存储位置开始的一块连续的存储空间中。

### 链式存储结构

链表

每个存储结点不仅包含有元素本身的信息（数据域），而且包含有元素之间逻辑模式关系的信息，即一个结点中包含直接后继结点的地址信息，这称为指针域。

单链表

只设置一个指针域，用以指向其直接后继结点，这个构成的链接表称为线性单向链接表。

插入结点：

s->next=p->next; p->next=s;



删除结点

p->next=p->next->next;



双链表

在每个结点中除包含有数值域外，设置两个指针域，分别用于指向其直接前驱结点和后继结点，这样构成的链表称之为线性双向链表。



循环链表

单循环链表

为空表是的判定p->next==L

统计其data值域为x的结点个数

|  |
| --- |
| int count(LinkList \*L,ElemType x){  int n =0;  LinkList \*p = L->next;  While(p != L){  If(p->data == x) n++;  p = p->next;  }  return(n);  } |

双循序链表

有一个带头结点的循环双链表L，设计一个算法删除第一个data域值为x的结点

|  |
| --- |
| Int deletem(DlinkList \*&L,ElemType x){  DlinkList \*p = L->next;  While(p!=L&& p->data != x){  P = p->next;  }  If(p!= L){  P->next->prior=p->prior;  p->prior->next=p->next;  free(p);  return(1);  }else return(0)  } |

有序表

所有的元素以递增或递减方式排列，并规定有序表中不存在元素值相同的元素。

## 栈和队列

### 栈

定义

一种只能在一端进行插入或删除操作的线性表。栈顶允许进行插入、删除操作。栈顶的当前位置是动态的，栈顶的当前位置由一个称为栈顶指针的位置指示器指示。表的另一端称为栈底。主要特点是“后进先出”。

栈的顺序存储结构

1、采用顺序存储的栈称为顺序栈

空栈 s->top = -1;

销毁栈 free(s);

栈的长度 s->top += 1;

栈满 s->top = MaxSize -1;

进栈 s->top +=1;s->data[s-top]=e;

出栈 e=s->data[s-top];s->top--;

2、采用链式存储的栈称为链栈

优点：不存在栈满溢出

空栈：s->next = NULL

销毁栈：

|  |
| --- |
| Void clearStack(LiStack \*&s){  LiStack \*p=s;\*q=s->next;  While(q != NULL){  free(q);  p=q;  q=p->next;  }  free(p);  } |

栈的长度

|  |
| --- |
| int StackLength(LiStack \*&s){  Int n = 0;  LiStack \*p;  p = s->next;  while(p != NULL){  n++;  p=p->next;  }  return n;  } |

进栈

|  |
| --- |
| void push(LiStack \*&s,ElemType e){  LiStack \*p;  p = (LiStack \*)malloc(sizeof(LiStack);  P->data=e;  P->next = s->next;  S->next = p;  } |

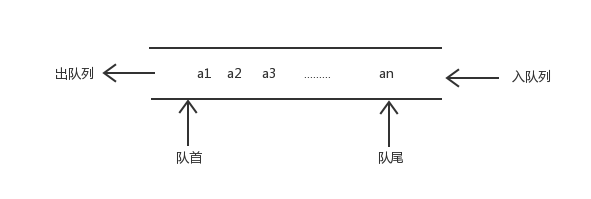
出栈

|  |
| --- |
| void pop(LiStack \*&s,ElemType e){  LiStack \*p;  If(s->next == NULL)  return 0;  p = s->next;  e = p->next;  s->next = p->next;  free(p);  return 1;  } |

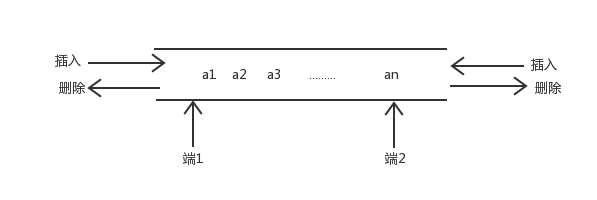
### 队列

定义

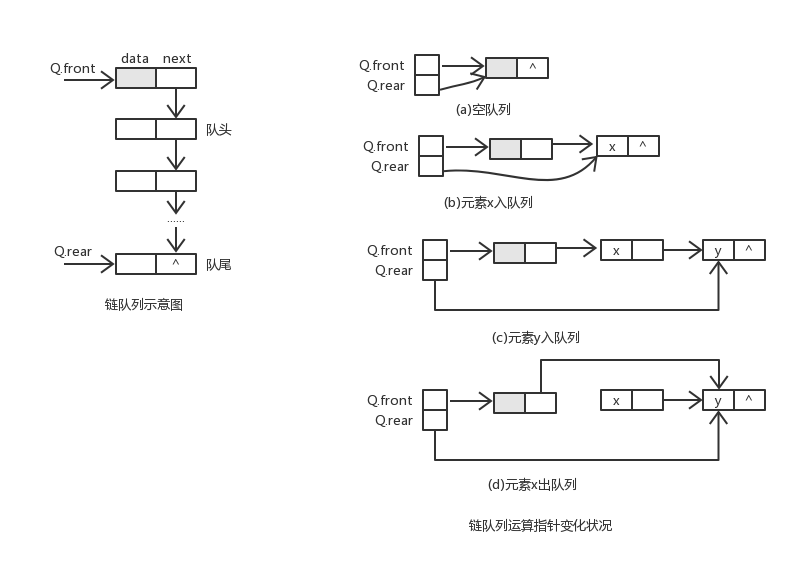
队列是一种先进先出（FIFO）的线性表。是一种操作受限的线性表，其限制为仅允许在表的一端进行插入，而在表的另一端进行删除。 在队列中允许插入的一端叫做队尾，允许删除的一端则称为对头。



双端队列：限定插入和删除操作在表的两端进行的线性表。



链队列：使用链表表示的队列。一个链队列需要队头和队尾（头指针，尾指针）的指针才能唯一确定。空链队列的判决条件为头指针和尾指针均指向头结点。



循环队列

## 串

### 基本概念

### 存储结构

### 模式匹配

## 数组和广义表

### 数组

### 稀疏矩阵

### 广义表

## 树和二叉树

### 树的基本概念

### 二叉树的概念和性质

### 二叉树存储结构

### 二叉树的基本运算及其实现

### 二叉树的遍历

### 二叉树的构造

### 线索二叉树

### 哈夫曼树

## 查找

### 线性表的查找

顺序表查找

二分查找

索引存储结构和分块查找

### 树表的查找

二叉树排序树

平衡二叉树

B-树

B+树

### 哈希表的查找

哈希表的基本概念

哈希函数构造方法

哈希冲突解决方法

哈希表上的运算

## 内排序

### 插入排序

直接插入排序

二分插入排序

希尔排序

### 交换排序

冒泡排序

快速排序

### 选择排序

直接选择排序

堆排序

### 归并排序

### 基数排序

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 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 |  |

# 分布式

## Dubbo

简介：

Dubbo是一款分布式服务框架，高性能和透明化的RPC实现服务的输出和输入功能。

主要核心部件：

Remoting：网络通信架构，实现sync-over-async和request-response消息机制

RPC：一个远程过程调用的抽象，支持负载均衡、容灾和集群功能

Registry：服务目录架构用于服务的注册和服务事件发布和订阅

工作原理：



Provider:暴露服务的服务提供者

Consumer：调用远程服务的服务消费者

Registry：服务注册与发现的中心目录服务的服务注册中心

Monitor：统计服务的调用次数和调用时间的日志服务的服务监控中心

调用流程：

1. 服务容器负责启动、加载、运行服务提供者
2. 服务提供者在启动时，向注册中心注册自己提供的服务
3. 服务消费者在启动时，向注册中心订阅自己所需的服务
4. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者
5. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。
6. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

## Spring Cloud

## Spring Boot

## Hadoop、Storm、Spark

## Zookeeper

### zookeeper是什么

### zookeeper哪里用到

### zookeeper的选主过程

### zookeeper集群之间如何通讯

### zookeeper的节点加密

### 分布式锁的实现过程

## MongDB

## HBase

# Linux命令

## linux常用的命令有哪些

## 如何获取java进程的pid

## 如何获取某个进程的网络端口号

## 如何实时打印日志

## 如何统计某个字符串行数