# Java笔记

## 容器/集合

### ArrayList内部实现

|  |
| --- |
| ArrayList内部其实为数组,占用空间连续,初始为长度为10数组,在add时对其赋值,并判断是否需要扩容,会将长度扩到之前大小的 + size/2的长度.  int newCapacity = elementData.length + (elementData.length >> 1 );  ArrayList的移除需要数组的拷贝,效率低下.  查找快  如下自定义实现一个ArrayList  public class MyArrayList<E> {  private Object[] elementData;  private static final Object[] *EMPTY\_ELEMENTDATA* = {};  private static final int *DEFALT\_CAPACITY* = 10;  private int size;  public MyArrayList(){  elementData = new Object[*DEFALT\_CAPACITY*];  }  public MyArrayList(int capacity){  if(capacity < 0){  throw new IllegalArgumentException(**"Illegal Capacity: "**+  capacity);  }else if(capacity == 0){  elementData = *EMPTY\_ELEMENTDATA*;  }  elementData = new Object[capacity];  }   public void add(E e){  if(size + 1 > elementData.length){  int newCapacity = elementData.length + (elementData.length >> 1 );  elementData = Arrays.*copyOf*(elementData,newCapacity);  }  elementData[size++] = e;  }   public boolean remove(E e){  boolean result = false;  for (int i = 0; i < size ; i++) {  if(elementData[i].equals(e)){  remove(i);  result = true;  }  }  return result;  }   public void remove(int index){  //[0,1,2,3,4,5]  checkRange(index);  int moveNum = size - index -1;  if(moveNum > 0){  System.*arraycopy*(elementData,index + 1,elementData,index,moveNum);  }  elementData[--size] = null;//把最后一个置为null,并size - 1  }     public E get(int index){  checkRange(index);  return (E)elementData[index];  }  public void set(E element,int index){  checkRange(index);  elementData[index] = element;  }   public int size(){  return this.size;  }  public void checkRange(int index){  if(index < 0 || index > size -1){  throw new IndexOutOfBoundsException(**"Index: "**+index+**", Size: "**+this.size);  }  }   **@Override** public String toString() {  StringBuilder stringBuilder = new StringBuilder();  stringBuilder.append(**"["**);  for (int i = 0;i< size;i++) {  stringBuilder.append(elementData[i] + **","**);  }  stringBuilder.setCharAt(stringBuilder.length() - 1,**']'**);  return stringBuilder.toString();  } } |

### LinkedList 原理和实现

|  |
| --- |
| 链表实现,如下图    一个Node节点有上一个元素,和下一个元素  链表查询效率很低,插入删除效率很高.因为只涉及到前后两个Node的设置  如果查找需要 从起点或结束点 一直 temp.prev或temp.next,可以判断index是否大于size/2,小于则从 first开始,否则从last开始  自定义LinkedList  如下:  public class MyLinkedList<E> {  private int size;  private Node<E> first;  private Node<E> last;  public void add(E e){  Node<E> newNode = new Node<>(e,last,null);  if(first == null){  first = newNode;  }else{  last.next = newNode;  }  last = newNode;  size++;  }   public void add(int index,E e){  if(index < 0 || index > size){  throw new IndexOutOfBoundsException(**"Index: "**+index+**", Size: "**+this.size);  }  if(index == size){  add(e);  }else{  Node<E> temp = getNode(index);  Node<E> prev = temp.prev;  Node<E> newNode = new Node<E>(e,prev,temp);  temp.prev = newNode;  if(prev == null){  first = newNode;  }else{  prev.next = newNode;  }  }  size++;  }  public void remove(int index){  checkRange(index);  Node<E> temp = getNode(index);  Node<E> prev = temp.prev;  Node<E> next = temp.next;  if(prev == null){  first = next;  }else{  prev.next = next;  temp.prev = null;  }   if (next == null){  last = prev;  }else{  next.prev = prev;  temp.next = null;  }  size--;  temp.item = null;  }    public E get(int index){  checkRange(index);  Node<E> result = getNode(index);  return result == null?null:result.item;  }   public Node<E> getNode(int index){  Node<E> result = null;  if (index < this.size >> 1){  Node<E> temp = first;  for (int i = 0; i < index; i++) {  temp = temp.next;  }  result = temp;  }else{  Node<E> temp = last;  for (int i = size - 1; i > index; i--) {  temp = temp.prev;  }  result = temp;  }  return result;  }    public void checkRange(int index){  if(index < 0 || index > size -1){  throw new IndexOutOfBoundsException(**"Index: "**+index+**", Size: "**+this.size);  }  }   **@Override** public String toString() {  StringBuilder stringBuilder = new StringBuilder(**"["**);  Node<E> temp = first;  while (temp!= null){  stringBuilder.append(temp.item + **","**);  temp = temp.next;  }  stringBuilder.setCharAt(stringBuilder.length() - 1,**']'**);  return stringBuilder.toString();  }   private static class Node<E>{  private E item;  private Node<E> prev;  private Node<E> next;   public Node(E item, Node<E> prev, Node<E> next) {  this.item = item;  this.prev = prev;  this.next = next;  }  } } |

### Vector

|  |
| --- |
| Vector 与 ArrayList区别在于 是 线程安全的,但也就意味着效率低. |

### HashMap

|  |
| --- |
| HashMap实现原理为哈希表,即:数组+链表,  **当不同key经过hash算法后,得到同一值称为hash冲突/hash碰撞,这时会将发生hash冲突的值组织成单向链表.**  当链表长度大于8时,会使用红黑树代替链表,效率更高    自定义HashMap:  计算key应该被分配到哪个索引数组中时:如果算法是如下  private int myHash(K key,int length){  return key==null?0:(key.hashCode()&(length - 1)); }  则需要数组的大小为:2的幂次 如:2,4,8,16.....才能保证获得的索引在数组长度范围内  HashMap的put:需判断put的key对应数组索引位置的Node是否为空,为空则新的值为第一个;不为空,则需一直判断key是否在已经在的链表中存在,存在就覆盖其value,如果node的next == null则说明已经是最后一个,将新值赋值给temp.next = newNode  如下:  public void put(K key,V value){  int node\_hash = myHash(key.hashCode(),table.length);  Node<K,V> first = table[node\_hash];  Node<K,V> newNode = new Node<K,V>(node\_hash,key,value,null);  if(first == null){  table[node\_hash] = newNode;  size++;  }else{  Node<K,V> temp = first;  for (int i = 0;;i++) {  if(key.equals(temp.key)){  temp.value = value;  break;  }  if(temp.next == null){  temp.next = newNode;  size++;  break;  }  temp = temp.next;  }  }  }  HashMap可以存储为null的K/V,但HashMap是非线程安全的,而Hashtable是线程安全的但不能存储为null的K/V  重写toString方法:  **@Override** public String toString() {  StringBuffer stringBuffer = new StringBuffer(**"{"**);  for (int i = 0; i < table.length; i++) {  Node<K,V> temp = table[i];  while (temp != null){  stringBuffer.append(temp.key + **":"** + temp.value + **","**);  temp = temp.next;  }  }  stringBuffer.setCharAt(stringBuffer.lastIndexOf(**","**),**'}'**);  return stringBuffer.toString(); }  HashMap扩容机制:  **Size大于等于设置的阈值,0.75\*table.length 并且 发生hash碰撞/hash冲突**  Get方法:  public V get(K key){  int hash = myHash(key,table.length);  Node<K,V> temp = table[hash];  if(temp == null){  return null;  }else{  while (temp != null){  if(key == null && temp.key == null){  return temp.value;  }  if(key.equals(temp.key)){  return temp.value;  }  temp = temp.next;  }  return null;  } }  自动以其他代码:  private Node<K,V>[] table; private static final int *DEFAULT\_ARRAY\_SIZE* = 16; private int size;  public MyHashMap() {  table = (Node<K,V>[])new Node[*DEFAULT\_ARRAY\_SIZE*]; }  static class Node<K,V>{  int hash;  K key;  V value;  Node<K,V> next;   public Node(int hash, K key, V value, Node<K, V> next) {  this.hash = hash;  this.key = key;  this.value = value;  this.next = next;  } } |

### TreeMap

|  |
| --- |
| 是由红黑二叉树实现,需要对K排序时,可以使用TreeMap效率更高,  存入TreeMap中的数据,会按照 K 增量的方式排序.  TreeMap<Integer,String> treeMap = new TreeMap<>(); treeMap.put(22,**"11"**); treeMap.put(4,**"4"**); treeMap.put(55,**"33"**);  打印如下:  {4=4, 22=11, 55=33} |

### Set

|  |
| --- |
| Set和List都继承自Collection接口  Set是存储特点是:无序不可重复的  HashSet底层其实就是使用HashMap的key存储数据,所有不能重复,    如上源码,将add的数据作为key存入HashMap中,value每次为常量Object对象  TreeSet:底层为:TreeMap,把添加的数据作为Key存入TreeMap,存入的数据key是递增有序的,如下:  TreeSet<Integer> treeSet = new TreeSet<>(); treeSet.add(444); treeSet.add(44); treeSet.add(434); treeSet.add(1);  结果:  [1, 44, 434, 444] |

### Iterator遍历

|  |
| --- |
| List/Set继承自Collection,而Collection继承了Iterable,所有list/set实现类可以使用Iterator遍历.而HashMap中有返回map中<K,V>的Set<Map.Entry<K,V>>,Set也可以使用Iterator遍历  如果需要遍历时对集合中元素删除则可以使用Iterator遍历  List:  List<Integer> list = new ArrayList<>(); list.add(122); list.add(33); list.add(4444); for(Iterator<Integer> iterator = list.iterator();iterator.hasNext();){  System.*out*.println(iterator.next()); }  Set:与list类似  Set<Integer> set = new HashSet<>(); set.add(122); set.add(33); set.add(4444); for(Iterator<Integer> iterator = set.iterator();iterator.hasNext();){  System.*out*.println(iterator.next()); }  Map:  Map<Integer,String> map = new HashMap<>(); map.put(11,**"33"**); map.put(3,**"ddd"**); map.put(33,**"xxx"**); Set<Map.Entry<Integer, String>> entries = map.entrySet();  for(Iterator<Map.Entry<Integer,String>> iterator = entries.iterator();iterator.hasNext();){  Map.Entry<Integer,String> entry = iterator.next();  System.*out*.println(entry.getKey() + **":"** + entry.getValue()); } |

### hashCode()和equals()

|  |
| --- |
| Java规范规定：   * 如果equals相等（object类equals就是==，比较对象地址），那么hashcode（object类hashcode为对象地址）一定要相等。所以一般重写equals方法也会重写hashcode()方法。 * Hashcode相等，不一定 equals相等，也就是equals不等，也可能hashcode相等   这样如：set,map等集合类，由于不能重复添加元素或key,需要判断，如果使用equals判断，元素多了（一个一个比较）效率会很底下，使用hashcode经过处理得到一个hash值判断，如果hash不等，则直接认为 是不同的对象，添加元素即可。如果hash相等再去判断 equals是否相等，相等则 覆盖。所以如果 equals相等而 hashcode不等，则set,map等会将相等的对象添加进去。 |

## 自定义类对象的比较排序

|  |
| --- |
| 两种方式:   * 让类实现Comparable接口并实现CompareTo方法 * 实现一个针对该类的比较器,实现Comparator接口,重写compare方法   如果类已经存在,并且不能改变该类的源代码,则可以使用第二种方式,后者耦合性更低 |

### 类实现Comparable接口

|  |
| --- |
| 想让对象作排序,让该类有排序的功能,则需实现Comparable接口,重写CompareTo方法  This.xx> o.xx 返回 正数  This.xx< o.xx 返回 负数  This.xx == o.xx 返回 0  public class Person implements Comparable<Person>{  private int id;  private String name;  private int age;   public Person(int id, String name, int age) {  this.id = id;  this.name = name;  this.age = age;  }   **@Override** public int compareTo(Person p) {  if(this.age > p.age){  return 1;  }else if(this.age < p.age){  return -1;  }else{  if(this.id > p.id){  return 1;  }else if(this.id < p.id){  return -1;  }else {  return 0;  }  }   }  } |

### 给类去实现一个新的比较器,实现Comparator

|  |
| --- |
| 不用动 要排序的类,实现一个比较器就行.  public class PersonComparator implements Comparator<Person> {   **@Override** public int compare(Person o1, Person o2) {   if(o1.getId() > o2.getId()){  return 1;  }else if(o1.getId() < o2.getId()){  return -1;  }else{  return 0;  }   } }  ArrayList<Person> list = new ArrayList<>(); list.add(new Person(1,**"11a"**,77)); list.add(new Person(5,**"11b"**,2)); list.add(new Person(7,**"11c"**,7));  Collections.*sort*(list,new PersonComparator());  Collections.sort(list,new PersonComparator());  Collections可以指定一个比较器. |

## 算法和数据结构

|  |
| --- |
| **数据结构**:数据的存储结构  **算法**:解决某个问题的方法/操作数据的一组方法  算法一般基于数据结构之上的,相对于数据结果来说的.  **复杂度分析**:效率和资源消耗的度衡量.(时间复杂度,空间复杂度)  大O标记法:将数据量假设往大加/循环次数加多,看算法的时间复杂度是怎么变化的,如果是成倍数增大则为:O(n)  成指数增大则为:O(n²)  **最常用的方法是只关注循环执行次数最多的那一段代码的时间复杂度**  **如果整个代码有多个量级，如有：**O(n)和O(n²)则取最大量级为：O(n²)   * 只关注循环执行次数最多的一段代码 * 加法法则:总复杂度等于量级最大的那段代码的复杂度 * 乘法法则:嵌套代码的复杂度等于嵌套内外代码复杂度的乘积   非多项式量级只有两个:O(2 n )和O(n!)，效率比较低，随着数据的增大复杂度急剧增大。   1. O(1) ：常量级时间复杂度的一种表示方法 2. O(logn)、O(nlogn)：对数阶时间复杂度 3. O(m+n)、O(m\*n)：有两个数据规模决定复杂度   **空间复杂度：**  常见的空间复杂度就是O(1)、O(n)、O(n 2 )  从低阶到高阶有:O(1)、O(logn)、O(n)、O(nlogn)、O(n² ) |

### 数组

|  |
| --- |
| 一种线性表数据结构，用一段连续的内存空间存储一组相同数据类型的数据。  **线性表**：数组，链表，队列，栈    **非线性表**,比如二叉树、堆、图等    数组为了保持内存数据的连续性，导致插入和删除效率低下。 |

### 链表

|  |
| --- |
| 一种不连续的内存结构，前一个指向下一个内存    单向链表：    链表的插入和删除 时间复杂度为：O(1)  循环链表：成环形连接，头尾相连    双向链表：方向有两个，有上一个和下一个，（LinkedList就是）    数组和链表： |

### 栈

|  |
| --- |
| 栈结构：先进后出，后进先出    从栈的操作特性上来看,栈是一种“操作受限”的线性表,只允许在一端插入和删除数据  栈的实现可以用数组也可以用链表：  如下数组实现：支持动态扩展  public class MyArrayStack<E> {  private int size;  private Object[] stackData;  private static final int *DEFAULT\_CAPACITY* = 10;  public MyArrayStack() {  stackData = new Object[*DEFAULT\_CAPACITY*];  }  public MyArrayStack(int capacity) {  stackData = new Object[capacity];  }  public boolean push(E e){  if(size + 1 > stackData.length){  int newCapacity = stackData.length + (stackData.length >> 1 );  stackData = Arrays.*copyOf*(stackData,newCapacity);  }  stackData[size++] = e;  return true;  }  public E pop(){  return (E) stackData[--size];  } }   * **栈的实际应用：每个线程会分配一个结构为：栈的内存，用来存储函数的临时变量，没进入一个函数就会将临时变量压入栈中，当被调用函数执行完成,返回之后,将这个函数对应的栈帧出栈。** * **表达式的求值：从左向右，将数字压入栈中，将操作符压入另一个栈中，当操作符优先级大于栈顶的操作符，则压入操作符栈中，当优先级小于或等于则将数字栈顶中前两个数取出使用操作符栈顶的运算符运算，将结果放入数字栈顶，继续进行。。。。**      * 使用栈实现页面的前进后退功能：   我们使用两个栈, X 和 Y ,我们把首次浏览的页面依次压入栈 X ,当点击后退按钮时,再依次从栈 X 中出栈,并将出栈的数据依次放入栈 Y 。当我们点击前进按钮  时,我们依次从栈 Y 中取出数据,放入栈 X 中。当栈 X 中没有数据时,那就说明没有页面可以继续后退浏览了。当栈 Y 中没有数据,那就说明没有页面可以点击前  进按钮浏览了。 |

### 队列

|  |
| --- |
| 先进先出，     * 顺序队列：数组实现，当入队时发现数组长度不够时，将队头到队尾的数移到从数组下标0开始的位置。      * 链式队列：链表实现      * 循环队列     数组实现：  public class CircularQueue {  // 数组:items,数组大小:n  private String[] items;  private int n = 0;  // head表示队头下标,tail表示队尾下标  private int head = 0;  private int tail = 0;  // 申请一个大小为capacity的数组  public CircularQueue(int capacity) {  items = new String[capacity];  n = capacity;  }  // 入队  public boolean enqueue(String item) {  // 队列满了  if ((tail + 1) % n == head) return false;  items[tail] = item;  tail = (tail + 1) % n;  return true;  }  // 出队  public String dequeue() {  // 如果head == tail 表示队列为空  if (head == tail) return null;  String ret = items[head];  head = (head + 1) % n;  return ret;  }  }   * 阻塞队列：当队列为空时，从队头取数据会被阻塞，直到队列中有数据才返回；如果队列满了，入队尾时会阻塞，知道队列有空位置再插入数据返回。     基于阻塞队列实现的 “ 生产者 - 消费者模型 ” ,可以有效地协调生产和消费的速度。当 “ 生产者 ” 生产数据的速度过快, “ 消费者 ” 来不及消费时,存储数据的队列  很快就会满了。这个时候,生产者就阻塞等待,直到 “ 消费者 ” 消费了数据, “ 生产者 ” 才会被唤醒继续 “ 生产 ”    **线程安全的队列我们叫作并发队列**  **队列应用场景：线程池请求排队的场景；数据库连接池** |

### 递归

|  |
| --- |
| 递归需要满足三个条件：   * 一个问题的解可以分解为几个子问题的解 * 这个问题与分解之后的子问题,除了数据规模不同,求解思路完全一样 * 存在递归终止条件   如果递归的深度过多，因为函数调用都会将临时变量压入栈中，层次过深导致栈空间不走会内存溢出。  避免重复计算：  f(5) = f(4) + f(3)  f(4) = f(3) + f(2)  其中f(3)被计算了多次，避免重复计算可以将f(k)结果存入数据结构中如（散列表：map）等中，计算前先检查是否存在，如下：  private static Map<Integer,Integer> *hashSolvedList* = new HashMap<>(); public static int f(int n){  int result;  if(n == 1){  return 1;  }  if(n == 2){  return 2;  }  if(*hashSolvedList*.containsKey(n)){  return *hashSolvedList*.get(n);  }  result = *f*(n - 1) + *f*(n - 2);  *hashSolvedList*.put(n,result);  return result; }  递归的利弊：  利：代码的表达力很强,写起来非常简洁;  弊：就是空间复杂度高、有堆栈溢出的风险、存在重复计算、过多的函数调用会耗  时较多等问题 |

### 排序

|  |
| --- |
| 排序算法的稳定性：是指相同的值，经排序后前后位置不变，如：15,1,4,4,7  排序后两个相同的4前后位置不变  实际应用如：订单按照金额小到大排序，金额相同的订单按照时间先后顺序  实现思路：先按照时间先后排序，再使用稳定性的排序算法按照金额排序     * 冒泡排序：互换相邻两个位置的数据进行排序，空间复杂度为：O(1)是原地排序算法，由于相邻数据有差异才去互换，所以冒泡排序是稳定的排序算法   当一轮比较后没有发生交换，则说明已经排好序了，可以提前退出（优化）  **时间复杂度就是O(n 2 )**  如：  int[] arrays = {4,11,7,3,6}; boolean flag = false; for (int i = 0; i < arrays.length - 1; i++) {  for (int j = 0; j < arrays.length - i - 1 ; j++) {  if(arrays[j] > arrays[j+1]){  int temp = arrays[j];  arrays[j] = arrays[j+1];  arrays[j+1] = temp;  flag = true;  }  }  if(!flag){  break;  } }   * 插入排序 |
|  |

### 跳表

|  |
| --- |
| Redis用到了跳表。跳表是一种，高效动态插入，删除和查询的一种数据结构。  它的实现：有序链表+ 索引      每两个结点提取一个结点到上一级，如果只是每两个提取一个，空间复杂度（On）太高，可以每多个节点提取到上一级（类似b+树）。  跳表的查询/插入/删除的时间复杂度（Ologn）,在插入删除时需要动态更新索引，将没有在索引节点的数据 添加一个索引，实现方式：随机一个索引高度k，将数据加入到1-k的所有高度中去。如下如果不更新跳表索引，跳表将**退化成单链表**：    更新索引： |

### 散列表

|  |
| --- |
| Hash table:由数组演变而来。利用数组下标查找的时间复杂度为（O1）高效特性。  将key利用散列函数（hash函数）得到 数组下标index,从而查找。    散列函数：将key转换为对应数组下标的函数。散列函数的要求：  1. 散列函数计算得到的散列值是一个非负整数;  2. 如果key1 = key2,那hash(key1) == hash(key2);  3. 如果key1 ≠ key2,那hash(key1) ≠ hash(key2)。（**很难达到，不现实**）  **散列冲突**：不同key通过散列函数得到同样的hash值。散列冲突解决方法有两类：   * **开放寻址法( open addressing )**   如果出现散列冲突则重新探测一个空闲位置（如：插入或查询一个数据，如果发生冲突了，如果数据存在或是要查找的数据，则插入或返回数据。否则就从冲突的位置往下一个找）    如下如果是删除则需要先将数据标记为delete    **散列表的装载因子**=填入表中的元素个数/散列表的长度  装载因子越大,说明空闲位置越少,冲突越多,散列表的性能会下降   * **链表法( chaining )**     **散列表扩容：**动态的散列表，需要扩容，扩容需要重新计算hash值等等。。。    当装载因子大于某个阈值时进行扩容，当装载因子小于某个阈值时进行缩容。  **一次性扩容**：如当装载因子大于某个阈值，进行二倍扩容，这样如果数据过多,可能会消耗大量时间，这样的扩容机制就不太合适。  **可以分批完成**：当达到阈值时则，申请新的散列表，但不去扩容。当插入新值时，放入新的散列表，并拿出老的散列表中一个放入新的散列表，如此重复，数据就分批被转移到新散列表中。    查找操作：先从新的散列表中查找，如果没有再从老的散列表中查找。  解决冲突的两种方式的优缺点：   * 开放寻址法：如：ThreadLocalMap，   优点：查找速度快，序列化实现比较简单。  缺点：浪费内存。  当数据量比较小、装载因子小的时候,适合采用开放寻址法   * 链表法：LinkedHashMap,对大装载因子的容忍度更高     基于链表的散列冲突处理方法比较适合存储大对象、大数据量的散列表,而且,比起开放寻址法,它更加灵活,支持更多的优化策略,比如**用红黑树代替链表**。 |

### 树

|  |
| --- |
| 结构像一颗树。没有父节点的节点为**根节点，**没有子节点的节点**为叶子节点** |

### 二叉树

|  |
| --- |
| 每个节点**最多有两个叉**。如下：    如上二图：叶子节点都在最底层，除了叶子节点外，每个节点都有左右两个节点，这种称之为**满二叉树**  **完全二叉树：**叶子节点都在最后两层，最后一层的叶子节点**都靠左（由左向右排列）**，除最后一层，其他节点的个数都达到**最大化**。    二叉树的存储方式：   * 链式存储（节点有三个字段（数据，指向左右节点的指针））      * 基于数组的顺序存储（将左节点存储在2\*i的偶数index数组中，右节点存储在2\*i+1   的奇数index的数组位上i为当前节点存储的数组下标。）    如上为完全二叉树数组存储方式，为了方便计算只浪费了index为0的位置，如果是非完全二叉树则如下：比较浪费存储空间，这也是为什么完全二叉树要求最后一层的子节点**要靠左排列**。如果是完全二叉树使用数组结构比链式结构节省空间（不用存储指针/引用）。    二叉树遍历：（相对于节点自己分为：前中后）递归的过程,时间复杂度：O(n)  前序遍历：先打印树节点本身，再打印它的左子树，最后打印它的右子树。  中序遍历：先打印它的左子树，再打印节点自己，最后打印它的右子树。  后序遍历：先打印它的左子树，再打印右子树，最后打印节点自己。 |

### 二叉查找树（二叉排序树）

|  |
| --- |
| 二叉查找树最大的特点就是：支持动态数据集合的快速插入、删除、查找操作。  定义：树中任意一个节点，其左子树的每个节点值都小于该节点的值，其右子树的每个节点值都大于该节点的值。  **使用中序遍历可以输出有序的数据序列**。    查找：从根节点开始如果等于返回，小于则递归左子树，如果大于则递归右子树。  public Node find(int data){  Node node = tree;  while(node != null){  if(data < node.data ){  node = node.left;  }else if(data > node.data){  node = node.right;  }else{  return node;  }  }  return null;  }  插入：如果插入数据比节点数据小并且左叶子节点为空则把该数据放在该处，比节点大同理。    删除：  **第一种情况**是,如果要删除的节点没有子节点,我们只需要直接将父节点中,指向要删除节点的指针置为 null 。比如图中的删除节点 55 。  **第二种情况**是,如果要删除的节点只有一个子节点(只有左子节点或者右子节点),我们只需要更新父节点中,指向要删除节点的指针,让它指向要删除节点的  子节点就可以了。比如图中的删除节点 13 。  **第三种情况**是,如果要删除的节点有两个子节点,这就比较复杂了。我们需要找到这个节点的右子树中的最小节点,把它替换到要删除的节点上。然后再删除掉  这个最小节点,因为最小节点肯定没有左子节点(如果有左子结点,那就不是最小节点了),所以,我们可以应用上面两条规则来删除这个最小节点。比如图中  的删除节点 18 。  public void delete(int data){  Node node = tree;  Node father = null;  while (node != null && data != node.data){  father = node;  if(data < node.data){  node = node.left;  }else if(data > node.data){  node = node.right;  }  }  if(node == null){  return;  }  //没有子节点或有一个子节点  if(node.left == null || node.right == null){  if(father.left == node){  father.left = (node.left==null?node.right:node.left);  }  if(father.right == node){  father.right = (node.right == null?node.left:node.right);  }  }  //有两个子节点  if(node.left != null && node.right != null){  Node temp = node.right;  Node minFather = node;  while (temp.left != null){  minFather = temp;  temp = temp.left;  }  Node minNode = temp;  node.data = minNode.data;  minFather.left = minNode.right;  }  }  完整代码github上 |

### 红黑树

红黑树是一种：平衡二叉查找树

|  |
| --- |
| 如果二叉查找树退化成一条线，效率很相当于一个链表。为了提高效率设计出：**平衡二叉查找树：二叉树中任意一个节点的左右子树的高度相差不能大于 1。（只是这么定义，但也不一定如：红黑树可能不满足）**  发明平衡二叉查找树这类数据结构的初衷是,解决普通二叉查找树在频繁的插入、删除等动态更新的情况下,出现时间复杂度退化的问题  那么：满二叉树和完全二叉树就是一种平衡二叉树。    红黑树的定义：   * 根节点是黑色的，叶子节点都是黑色的并且不存储数据 * 任何相邻的父子节点不能同时为红色（需要黑色隔开） * 从一个节点到达 它可到达的叶子节点的所有路径，都包含相同数目的黑色节点。   下图省略了黑色的叶子节点： |

### B+树

|  |
| --- |
| B+ 树的特点:   * 每个节点中子节点的个数不能超过m,也不能小于m/2; * 根节点的子节点个数可以不超过m/2,这是一个例外; * m叉树只存储索引,并不真正存储数据,这个有点儿类似跳表; * 通过链表将叶子节点串联在一起,这样可以方便按区间查找; * 一般情况,根节点会被存储在内存中,其他节点存储在磁盘中。   使用二叉查找树，由于数据节点太多不能都存储在内存中，而放在磁盘上访问效率会很低，所以需要减少访问磁盘的次数，让二叉查找树 分支 更多，就会更少的访问磁盘。所以B+树是多叉树，**其数据都存储在叶子节点上，叶子节点用链表串起来的有序链表**。    但也不是 m叉树，m越大就越好，操作系统都是按页(一页大小通常是 4KB ,这个值可以通过 getconfig PAGE\_SIZE 命令查看)来读取的,一次会读一页的数据。如果要读取的数据量超过一页的大小,就会触发多次 IO 操作。所以尽量让一个节点存储的数据等于一页的大小。 |

## 设计模式

### 静态代理

|  |
| --- |
| 常用作日志记录，监控等，代理类和被代理对象 需要实现 同一接口 如下：  乘客：真是角色  火车票：接口  火车票代理点：代理类/代理角色（有真是角色 成员属性）  代理类：  public class TicketProxy implements Ticket {  private Ticket ticket ;    public TicketProxy(Ticket ticket){  this.ticket = ticket ;  }  @Override  public void getTicket() {  //准备工作  ticket .getTicket();  //结束工作  }  }  代理对象：  public class People implements Ticket {  @Override  public void getTicket() {  System.out.println("获得火车票");  }  }  **接口：**  public interface Ticket {  public void getTicket();  }  调用：  new TicketProxy(new People()).getTicket(); |

### 动态代理

|  |
| --- |
| 静态代理只能 去 代理 实现的 接口类型的 对象，如：火车代理只能代理火车票，不能代理其他的如：电影票，其他所有的  动态代理 代理类 不是我们自己写的。动态生成的代理对像  Jdk动态代理如下：  Tank tank = new Tank();  Moveable movePorxy = (Moveable)Proxy.newProxyInstance(Tank.class.getClassLoader()  ,new Class[]{Moveable.class}, new InvocationHandler() {  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  startLog();  method.invoke(tank,args);  endLog();  return null;  }  public void startLog(){  System.out.println("log start");  }  public void endLog(){  System.out.println("log end");  }  });  movePorxy.move();//执行的是 代理类对象的move方法  生成代理对象 需要 传入：   * 和被代理对象同一个 类加载器 * 被代理对象实现的 接口 Class<?>[] interfaces,因为 动态生成的代理类需要去实现这些接口   如下设置 jdk动态代理 生成文件  System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");    这个动态代理类 中move方法 中会调用 传入的InvocationHandler的实现类对象   * 传入 InvocationHandler对象 ，此实现类 中invoke方法 会被 $Proxy0.class代理对象 move 方法调用 如下：     **我们只需要在 invoke方法中 实现自己的 代理行为 就行。**  JDK动态代理必须 被代理类 必须实现一个接口  Cglib实现动态代理：  Enhancer enhancer = new Enhancer();  enhancer.setSuperclass(Tank.class);//设置动态代理类的父类  enhancer.setCallback(new MethodInterceptor() {//回调  public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {  System.out.println("before");  Object result = methodProxy.invokeSuper(o, objects);//o就是生成的动态代理类对象  System.out.println("end");  return result;  }  });  Tank tank = (Tank) enhancer.create();  tank.move();  **//cglib被代理类 不能是 final的，因为 代理类必须继承 它**  Spring的aop就是动态代理实现，如下：    动态 给 一个方法 之前或之后 去动态添加 一个方法。去执行。  或者注解： |

### 工厂

任何能够产生对象的类方法都可算是工厂

|  |
| --- |
| * 简单工厂   简单工厂其实**不算是设计模式，算是 编程习惯**  将new 实例的事情 在 工厂中去做，如下：所有类型披萨都能生产  SimplePizzaFactory：披萨工厂  public static Pizza createPizza(String pizzaType){  if("CheesePizza".equals(pizzaType)){  return new CheesePizza();  }else if("ClamPizza".equals(pizzaType)){  return new ClamPizza();  }  return null;  }  PizzaStore：披萨店  public Pizza orderPizza(String pizzaType){  Pizza pizza = SimplePizzaFactory.createPizza(pizzaType);  pizza.prepare();//准备  pizza.bake();//  pizza.cut();//切块  pizza.box();//打包  return pizza;  }   * 工厂方法：不同类型由不同类型工厂生产,单个产品扩展容易   所有工厂实现一个工厂的接口，不同工厂去生产不同产品。       * 抽象工厂：任意定义产品一族，工厂生产一系列产品，具体哪一系列由具体系列工厂实现，只需要实现系列工厂，和 继承/创建产品       名词用抽象类（因为比较具体属于类），形容词用接口（不具体的属于接口） |

### 单例

|  |
| --- |
| Singleton:只需要一个实例   * 饿汉式：最实用，不存在线程安全问题   private static final Singleton1 *INSTANT* = new Singleton1();  private Singleton1(){  } public static Singleton1 getInstant(){  return *INSTANT*; }   * 懒汉式：如下写法不完美，存在并发问题   private static Singleton2 *instant*; private Singleton2(){ } public static Singleton2 getInstant(){  if(*instant* == null){  //多线程情况下如果在此处切换线程，会导致非单例  *instant* = new Singleton2();  }  return *instant*; }   * 加同步锁的懒汉式：效率太低，锁的是XX.class粒度太大   private static Singleton3 instant;  private Singleton3(){  }  public static synchronized Singleton3 getInstant(){  if(instant == null){  instant = new Singleton3();  }  return instant;  }   * 优化同步锁，双重检查：   private static Singleton2 instant;  private Singleton2(){  }  public static Singleton2 getInstant(){  if(instant == null){//效率  synchronized (Singleton2.class) {  if (instant == null) {  //多线程情况下如果在此处切换线程，会导致非单例  instant = new Singleton2();  }  }  }  return instant;  }   * 静态内部类：   private Singleton4(){}  public static class SingleHolder{  private static final Singleton4 INSTANT = new Singleton4();  }  public static Singleton4 getInstant(){  return SingleHolder.INSTANT;//构造方法只调用一次，所以不存在并发问题  }   * 枚举实现：比较完美，单例类也不会被序列化   public enum Singleton5 {  INSTANT;  }  序列化和反序列化 会 是不同的对象。 |

### 策略模式（strategy）

|  |
| --- |
| 开闭原则：尽量不要修改原来的代码,对扩展开放。  Comparator就是策略模式    比较的方式/策略有不同方式。 |

### 原型/克隆模式

|  |
| --- |
| 需要克隆一个对象。Java自带克隆模式，如果属性多又需要再用一个  被克隆的对象需要 实现 Cloneable接口，并重写 clone方法，如果不实现接口只重写方法，运行时会报错。如下：只是对对象进行克隆，对象中的引用对象没有处理，会导致修改其中一个属性对象的属性，而使克隆后的对象属性也发生改变 为**浅克隆**  **浅克隆 对象的属性 如果是 对象则 只拷贝 了地址，两个对象指向同一个对象。**  **但 String 特殊，因为其值在常量池中。**  public class People implements Cloneable{  public int age;  public String name;  public Location location = new Location(1,2) ;    public People(int age,String name){  this.age = age;  this.name = name;  }  @Override  protected Object clone() throws CloneNotSupportedException {  return super.clone();  }  }  如下修改 location的属性 x 会将 克隆后对象中location对象的x也改变，  People p1 = new People(1,new String("space"));  People p2 = (People) p1.clone();  System.out.println("p1.age:" + p1.age + ",name:" + p1.name);  System.out.println("p2.age:" + p2.age + ",name:" + p2.name);  p1.location.x = 3;  System.out.println(p2.location.x);  **深克隆：属性对象也需要实现 Cloneable,和重写 Clone方法**  public class People implements Cloneable{  public int age;  public String name;  public Location location = new Location(1,2) ;  public People(int age,String name){  this.age = age;  this.name = name;  }  @Override  protected Object clone() throws CloneNotSupportedException {  People p = (People) super.clone();  p.location = (Location)location.clone();  return p;  }  }  public class Location implements Cloneable{  public int x;  public int y;  public Location(int x,int y){  this.x = x;  this.y = y;  }  @Override  protected Object clone() throws CloneNotSupportedException {  return super.clone();  }  } |

## 多线程

|  |
| --- |
| 多条线路同时进行，真正多线程是多核cpu才能实现。 |
|  |

### 线程的创建

|  |
| --- |
| 线程有三种方式创建：   * **继承Thread类**   public class MyThread extends Thread {  @Override  public void run() {  System.out.println(Thread.currentThread().getName());  }  }  MyThread myThread = new MyThread();  myThread.start();   * **实现Runnable接口:常用解决不能多继承问题**   public class MyRunnable implements Runnable {  @Override  public void run() {  System.out.println(Thread.currentThread().getName());  }  }  new Thread(new MyRunnable(),"t2").start();  使用Thread启动   * **现实Callable接口** |

### 线程状态

|  |
| --- |
| 线程创建即为：新生状态  Start()后 就等待cpu的某时刻的调用，此时处于：就绪状态  进入就绪状态的情况：   * Start()方法 * yield() 让出Cpu 重新进入就绪状态   当cpu执行线程时：运行状态  运行结束即为：死亡状态  运行状态进入 阻塞状态：   * Sleep :不释放锁 。Timed Waiting 类的阻塞 * Wait ：释放锁 :WAITING * Join：WAITING * Io的read/write操作   从阻塞到就绪状态：   * Sleep时间到 * Notify/notifyAll :让当前锁对象上的线程 解除阻塞状态进入就绪状态，但不释放锁 |

### 停止线程

|  |
| --- |
| 不要使用 线程中的 stop()/destroy()方法，使用标识字段控制线程停止。如下：  public class TestThreadStop implements Runnable{   private volatile boolean flag = true;  **@Override** public void run() {  while(flag){  System.*out*.println(Thread.*currentThread*().getName() + **""** + System.*currentTimeMillis*());  }  }   public void setStop(){  flag = false;  }   public static void main(String[] args) {  TestThreadStop tts = new TestThreadStop();  new Thread(tts).start();  try {  Thread.*sleep*(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  tts.setStop();  } } |

### 线程的优先级

|  |
| --- |
| 线程的优先级，优先级高只是表示 被cpu执行的**概率高**，并不能保证一定优先执行。  默认所有线程的优先级均为5  Thread.MAX\_PRIORITY：**10**  Thread.NORM\_PRIORITY:**5**  Thread.MIN\_PRIORITY:**1**  如下：  Thread t2 = new Thread(() -> {  for (int i = 0; i < 100; i++) {  System.out.println(Thread.currentThread().getName() + ":" + i);  }  },"t2");  System.out.println(t2.getPriority());//获取优先级  t2.setPriority(Thread.MAX\_PRIORITY);//设置优先级  t2.start(); |

### 线程分类

|  |
| --- |
| 线程分为：**用户线程** 和 **守护线程（daemon）**  我们创建的一般线程均为 用户线程  守护线程是为 用户线程 服务的 jvm停止 不用等待守护线程执行完毕，只需要用户线程执行完就行。如下：  new Thread(() -> {  for (int i = 1; i < 366; i++) {  System.out.println(Thread.currentThread().getName() + ":" + i);  }  },"t1").start();  Thread daemon = new Thread(() -> {  while (true){  System.out.println(Thread.currentThread().getName() + ",daemon");  }  },"daemon");  **daemon.setDaemon(true);**  daemon.start();  使用**.setDaemon(true);设置为守护线程，jvm不会等守护线程执行完才停止，只要用户线程执行完就停止。** |

### 线程的生命周期

|  |
| --- |
| **Runnable ---->blocked**:如：加了synchronized的代码块或方法，只允许一个线程执行，其他线程执行时就从runnable转化为blocked状态。当获得锁就会从blocked---》runnable.  **Runnable--->waiting:**   * 调用wait() * 调用A.join()的线程会转换为waiting状态，当A执行完后又转换为Runnable状态。 * 调用LockSupport.park()方法。并发包中的方法。   **RUNNABLE与TIMED\_WAITING：**   1. 调用**带超时参数**的Thread.sleep(long millis)方法； 2. 获得synchronized隐式锁的线程，调用**带超时参数**的Object.wait(long timeout)方法； 3. 调用**带超时参数**的Thread.join(long millis)方法； 4. 调用**带超时参数**的LockSupport.parkNanos(Object blocker, long deadline)方法； 5. 调用**带超时参数**的LockSupport.parkUntil(long deadline)方法。   **New--->runnable：调用线程的start()方法。**  **Runnable--->terminated（结束）:**  一般run方法执行完就会转换为terminated状态，但有时需要主动停止线程，则需要：  Stop()(**已经废弃**)，推荐使用interrupt()  Stop方法会强制杀掉线程，有的线程还没有释放锁，stop后其他线程就永远无法获取到锁。  interrupt()方法会通知线程 |

### 常用方法

|  |
| --- |
| Thread.currentThread()//当前线程  t1.setName("t1");//设置线程name  t1.isAlive() //是否还存活 |

## 并发编程



### 多线程同步

|  |
| --- |
| 操作系统的多任务即为：多进程。如：wps,chrome,等等。。。  线程为进程的子任务  Synchronized线程同步，如果修饰对象：则为给这个对象加了一把锁，其他线程访问需先要获取这把锁，锁标记是在该对象存储的**堆**上存储标记的  private int count; private Object o = new Object();  public void m(){  synchronized(o){//或者this  count--;  System.*out*.println(Thread.*currentThread*().getName() + **" count = "** + count);  } }  当synchronized中代码执行结束时，则锁释放，这种只要有一个线程拿到锁，其他线程无法获取 为**互斥锁**  **Synchronized锁定的是对象，不是代码块,加了Synchronized的代码需要去申请对象锁**  public void m(){  synchronized(this){//锁定自身对象  count--;  System.out.println(Thread.currentThread().getName() + " count = " + count);  }  }  也可以一下简写：  public synchronized void m(){  count--;  System.out.println(Thread.currentThread().getName() + " count = " + count);  }  **当Synchronized修饰static方法时**，则表加锁的对象为 该类的class对象，因此，其他任意线程调用该类时都需要先获取 class对象锁。  private static int *count*;  public synchronized static void m(){  *count*--; }  相当于如下：  public static void m2(){  synchronized (Test2.class){  count--;  }  }  **Synchronized 的代码块是原子操作，其他线程不可打断，如下为线程重入**  public class Test3 implements Runnable {  private int count = 10;  @Override  public void run() {  count--;  System.out.println(Thread.currentThread().getName() + " count = " + count);  }  public static void main(String[] args) {  Test3 t3 = new Test3();  for (int i = 0; i < 5; i++) {  new Thread(t3,"Thread" + i).start();  }  }  }  由于几个线程对同一个对象 t3的属性 count进行操作，可能在第一个线程count--后还没有打印，其他线程又来执行，这时再去执行第一个线程打印时count的值可能被--了多次，加synchronized  public synchronized void run() {  count--;  System.out.println(Thread.currentThread().getName() + " count = " + count);  }  在同步方法执行期间，非同步方法是可以调用的，因为不需要去申请锁  public class Test4{   public synchronized void m1(){  System.*out*.println(Thread.*currentThread*().getName() + **" m1 start"**);  try {  Thread.*sleep*(10000);   } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(Thread.*currentThread*().getName() + **" m1 end"**);  }  public void m2(){  try {  Thread.*sleep*(5000);  System.*out*.println(Thread.*currentThread*().getName() + **"m2"**);  } catch (InterruptedException e) {  e.printStackTrace();  }  }   public static void main(String[] args) {  Test4 t4 = new Test4();  new Thread(t4::m1,**"t1"**).start();  new Thread(t4::m2,**"t2"**).start();  } }  new Thread(t4::m2,"t2").start();为Lambda表达式  相当于 new Runnable并在run方法中调用 t4.m2()方法  new Runnable(){  @Override  public void run() {  t4.m2();  }  } |

### 脏读

|  |
| --- |
| 由于给只给写操作加锁，没给读操作加锁而导致读取到写操作中没有完成的数据或（数据的差异），在写操作执行完后才允许读操作执行。**需看实际业务中是否允许脏读**  如下产生脏读：  public class Test5 {   private String name;  private double balance;   public synchronized void set(String name,double balance){  this.name = name;  try {  Thread.*sleep*(5000);  } catch (InterruptedException e) {  e.printStackTrace();  }  this.balance = balance;  }   public double get(String name){  return this.balance;  }   public static void main(String[] args) {  Test5 t5 = new Test5();  new Thread(()->t5.set(**"x"**,100),**"t1"**).start();  try {  Thread.*sleep*(2000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(t5.get(**"x"**));   try {  Thread.*sleep*(3000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(t5.get(**"x"**));  } } |

### Synchronized获得的锁是可重入的

|  |
| --- |
| 即为：**同一个线**程内一个同步方法可以调用另一个同步方法，以为已经获得了锁，再去获取只是它的所标记或+1，获得了两次  如下代码：  public class Test6 {   public synchronized void m1(){  System.*out*.println(**"m1.start"**);  try {  Thread.*sleep*(2000);  } catch (InterruptedException e) {  e.printStackTrace();  }  m2();  }   public synchronized void m2(){  try {  Thread.*sleep*(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(**"m2"**);  }   public static void main(String[] args) {  Test6 t6 = new Test6();  new Thread(t6::m1,**"t1"**).start();   } }  **子类的同步方法可以调用父类的同步方法。**  public class T {  public synchronized void m(){  System.out.println("T.m()");  }  public static void main(String[] args) {  new TT().m();  }  }  class TT extends T{  public synchronized void m(){  System.out.println("TT.m() start");  super.m();  System.out.println("TT.m() end");  }  }  注意：父类中this指的是new出来的子类对象，所以两个同步方法锁定的是一个对象 |

### Synchronized锁升级

|  |
| --- |
| jdk1.6之前为重量级锁，加锁和解锁的过程需要去os（操作系统）去申请，开销比较大，1.6之后得到了优化：（锁升级）   * **偏向锁**：当如果只有一个线程调用（对象头中）记录线程ID,如果该线程后边还需要申请锁，则直接使用不需要获取锁 * **自旋锁**：当第二个线程来获取锁时，升级为自旋锁，cpu占用一直在循环等待（10次）之后升级为重量级锁（等待队列）。 * 重量级锁：   加锁代码执行时间长，线程数多用系统锁  执行时间短，线程数少，用自旋锁。 |

### CAS

|  |
| --- |
| 无锁优化（自旋），必须硬件支持，cpu提供了**CAS指令**(全称是Compare And Swap,即“比较并交换”)。  CAS指令包含三个参数：   * 共享变量内存地址A * 用于比较的值B * 共享变量的新值C   并且只有当内存中地址A处的值等于B时,才能将内存中地址A处的值更新为新值C。  作为一条CPU指令,**CAS指令本身是能够保证原子性的** 。  原理：将共享变量A(可见的)进行操作（如A+1），传入A副本B接着判断此时的A是否等于B,不等于则表明A已经被其他线程修改，循环重新去做A+1操作接着在判断，直到A==B则表明没有其他线程修改A,此时再将A的值更新为（A+1的新值）。    如下伪代码：      Java原子类：Atomic×××其中的实现如下：实际使用见：[使用AtomicXXXX类保证原子性](#_使用AtomicXXXX类保证原子性)    分类： |

### 死锁

|  |
| --- |
| 线程A在获取获取锁o1期间又去获取锁o2,  而线程B在获取锁o2期间又去获取锁o1  当线程A在没有释放锁o1后锁o2又被B线程占用，但B线程又需要获取o1后才能释放o2所以就产生死锁 如：    public class A {  private Object o1,o2;  public A (Object o1,Object o2){  this.o1 = o1;  this.o2 = o2;  }   public void m1(){  synchronized (o1){  System.*out*.println(Thread.*currentThread*().getName() + **" o1 lock"**);  try {  Thread.*sleep*(3000);  } catch (InterruptedException e) {  e.printStackTrace();  }   synchronized (o2){  System.*out*.println(Thread.*currentThread*().getName() + **" o2 lock"**);  }  }   }   public void m2(){   synchronized (o2){  System.*out*.println(Thread.*currentThread*().getName() + **" o2 lock"**);  try {  Thread.*sleep*(5000);  } catch (InterruptedException e) {  e.printStackTrace();  }  synchronized (o1){  System.*out*.println(Thread.*currentThread*().getName() + **" o1 lock"**);  }  }    }  }  public static void main(String[] args) {  Object o1 = new Object();  Object o2 = new Object();  A a = new A(o1,o2);  new Thread(a::m1,"t1").start();  new Thread(a::m2,"t2").start();  }  只有以下这四个条件都发生时才会出现死锁：   1. 互斥，共享资源X和Y只能被一个线程占用； 2. 占有且等待，线程T1已经取得共享资源X，在等待共享资源Y的时候，不释放共享资源X； 3. 不可抢占，其他线程不能强行抢占线程T1占有的资源； 4. 循环等待，线程T1等待线程T2占有的资源，线程T2等待线程T1占有的资源，就是循环等待。   反过来分析，**也就是说只要我们破坏其中一个，就可以成功避免死锁的发生**。 破坏占用且等待条件：   class Allocator {  private List<Object> als = new ArrayList<>();  // 一次性申请所有资源  synchronized boolean apply(  Object from, Object to){  if(als.contains(from) ||  als.contains(to)){  return false;  } else {  als.add(from);  als.add(to);  }  return true;  }  // 归还资源  synchronized void free(  Object from, Object to){  als.remove(from);  als.remove(to);  }  }  class Account {  // actr应该为单例  private Allocator actr;  private int balance;  // 转账  void transfer(Account target, int amt){  // 一次性申请转出账户和转入账户，直到成功  while(!actr.apply(this, target))  ；  try{  // 锁定转出账户  synchronized(this){  // 锁定转入账户  synchronized(target){  if (this.balance > amt){  this.balance -= amt;  target.balance += amt;  }  }  }  } finally {  actr.free(this, target)  }  }  } 破坏循环等待条件：申请锁的顺序一样 class Account {  private int id;  private int balance;  // 转账  void transfer(Account target, int amt){  Account left = this ①  Account right = target; ②  if (this.id > target.id) { ③  left = target; ④  right = this; ⑤  } ⑥  // 锁定序号小的账户  synchronized(left){  // 锁定序号大的账户  synchronized(right){  if (this.balance > amt){  this.balance -= amt;  target.balance += amt;  }  }  }  }  } |

### 程序运行期间出现异常，锁默认会被释放

|  |
| --- |
| **当在同步代码块中，程序出现异常，则锁被释放。这样可能导致其他线程访问到不完整的数据。如下代码：**  public class Test {  private int count = 0;  public synchronized void add(){  System.out.println(Thread.currentThread().getName() + " start");  while (true){  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + " count=" + count);  count ++;  if(count == 5){  count = 1/0;//异常，释放锁  }  }  }  public static void main(String[] args) {  Test t = new Test();  new Thread(t::add,"t1").start();  try {  Thread.sleep(3000);  } catch (InterruptedException e) {  e.printStackTrace();  }  new Thread(t::add,"t2").start();//只有t1线程释放锁后，t2才会执行  }  }  **将异常处理后就不会释放锁了，如下：**  while (true){  try {  Thread.sleep(1000);  System.out.println(Thread.currentThread().getName() + " count=" + count);  count ++;  if(count == 5){  count = 1/0;  }  } catch (Exception e) {  e.printStackTrace();  }  } |

### Volatile （线程之间可见性）

**可见性**和**禁止指令重排序**。

|  |
| --- |
| Volatile修饰 成员变量 ：表示，当数据被一个线程更改后，会通知其他线程 将数据从主内存区 更新到 cpu缓存中去。从而让 数据 实时保持一致    如下：主线程修改 flag后 如果 flag 不使用volatile 关键字 则由于 t1线程忙碌（不好判断） 没有从主内存区更新 数据最新值，则线程t1无法停止  Cpu是否忙碌，是否去主内存中刷数据，不好判断，但加volatile会去通知，**无锁同步**  public class Test {  private volatile boolean flag = true;  public void m(){  System.out.println(Thread.currentThread().getName() + " start");  while (flag){  }  System.out.println(Thread.currentThread().getName() + " end");  }  public static void main(String[] args) {  Test t = new Test();  new Thread(t::m,"t1").start();  try {  Thread.sleep(2000);  } catch (InterruptedException e) {  e.printStackTrace();  }  t.flag = false;  }  } |

### Join

|  |
| --- |
| 插队执行，在线程1中执行 xxx.join()方法 线程1就需要等待 xxx线程执行完后才能执行，线程1处于**阻塞状态**  等待子线程运行完后，主线程接着运行  如：  public class Test {   public void m(){  System.*out*.println(Thread.*currentThread*().getName() + **":子线程"**);  try {  Thread.*sleep*(10000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }   public static void main(String[] args) {  System.*out*.println(Thread.*currentThread*().getName() + **"：主线程start"**);  Test t = new Test();  Thread thread = new Thread(t::m,**"t1"**);  thread.start();  try {  thread.join();  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(Thread.*currentThread*().getName() + **"：主线程end"**);   }  } |

### yield

|  |
| --- |
| 礼让：当前线程进入就绪状态，重新竞争cpu  new Thread(() ->{  for (int i = 0; i < 100; i++) {  System.out.println(Thread.currentThread().getName() + ":" + i);  }  },"t1").start();  for (int i = 0; i < 100; i++) {  if(i % 10 == 0){  Thread.yield();  }  System.out.println(Thread.currentThread().getName() + ":" + i);  } |

### wait/notify/notifyAll

|  |
| --- |
| 等待-通知机制  **线程首先获取互斥锁，当线程要求的条件不满足时，释放互斥锁，进入等待状态；当要求的条件满足时，通知等待的线程，重新获取互斥锁**。  当调用wait()方法后，当前线程就会被阻塞，并且进入到右边的等待队列中，**这个等待队列也是互斥锁的等待队列**。 线程在进入等待队列的同时，**会释放持有的互斥锁**，线程释放锁后，其他线程就有机会获得锁，并进入临界区了。  wait()、notify()、notifyAll()方法操作的等待队列是互斥锁的等待队列，所以如果synchronized锁定的是this，那么对应的一定是this.wait()、this.notify()、this.notifyAll()；  这三个方法能够被调用的前提是已经获取了相应的互斥锁，所以我们会发现wait()、notify()、notifyAll()都是在**synchronized**{}内部被调用的。  while(条件不满足) {  wait();  } 尽量使用notifyAll() Notify()会随机通知等待队列中的一个线程，虽然只有一个线程进入临界区，但可能导致有的线程永**远不被通知** |

### 使用AtomicXXXX类保证原子性

更多见上面章节：[CAS](#_CAS)

|  |
| --- |
| Volatile修饰变量只能保证 可见性，但不能保证原子性。如：  Volatile int n = 0;  M(){  n++;  }  使用synchronized效率低，可使用AtomicInteger  多个线程执行m方法并不能保证n的原子性，但可以使用AtomicInteger类，它其中的方法是原子性的并且是可见的，但**方法和方法之间并不具有原子性**。  public class Test {  AtomicInteger count = new AtomicInteger(0);   public void add(){  for (int i = 0; i < 100; i++) {  count.getAndIncrement();  }  }   public static void main(String[] args) {  ArrayList<Thread> threads = new ArrayList<Thread>();  Test t = new Test();  for (int i = 0; i < 10; i++) {  threads.add(new Thread(t::add,**"t"** + i));  }  threads.forEach((o) -> o.start());  threads.forEach((o) -> {  try {  o.join();  } catch (InterruptedException e) {  e.printStackTrace();  }  });  System.*out*.println(t.count);  }   }  public void add(){  for (int i = 0; i < 100; i++) {  If(count.get() < 10){  count.getAndIncrement();  }  }  }  如上 get方法 和 getAndIncrement方法同时调用不具有原子性，有可能调用get()方法之后 count = 9，另一个线程执行了getAndIncrement 操作 count = 10了,这样导致线程1getAndIncrement 完后 count = 11的错误结果。 |
|  |

### Synchronized效率优化/注意

|  |
| --- |
| * Synchronized锁的粒度越细效率越高，Synchronized代码块中的**代码越少**效率越高 * 避免将锁定的对象的引用变成另外一个对象，对象发生改变，锁定的对象变了。 * 不要以字符串常量作为锁定对象如下：   String s1 = “space”;  String s2 = “space”;  S1和s2为同一个字符串，锁定的也就是同一个对象。 |

### Lock和condition

|  |
| --- |
| Lock也是管程的实现，和synchronized区别如下：    使用实例：  public class Test {  private **final Lock rt = new ReentrantLock(true);//为可重入锁，true表示公平锁**  int count;  public void addOne(){  try {  rt.lock();  count = get() + 1;  }finally {  rt.unlock();  }  }  private int get() {  try {  rt.lock();  return count;  }finally {  rt.unlock();  }  }  public static void main(String[] args) {  Test t = new Test();  for (int i = 0; i < 10; i++) {  new Thread(t::addOne,i+"线程").start();  }  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(t.count);  }  }  **Lock rt = new ReentrantLock(true);为可重入锁，true为公平锁，当有其他线程释放锁后，会先通知（等待队列前边的）线程进入，false为随机的。ReentrantLock使用CAS获取锁。**  **Lock的使用范例：**  try {  rt.lock();//加锁  }finally {  rt.unlock();//释放锁  }  **推荐的三个用锁的最佳实践，它们分别是：**   * 永远只在更新对象的成员变量时加锁 * 永远只在访问可变的成员变量时加锁 * 永远不在调用其他对象的方法时加锁，（其他对象方法可能也加锁，或有超时的操作）    tryLock()方法返回true则获取锁成功，有返回值的获取锁，**不进入阻塞状态**。  如下：  boolean flag = false;  try {  while(!(flag = **lock.tryLock()**)) {  System.out.println(Thread.currentThread().getName() + ":" + flag);  }  count++;  System.out.println(Thread.currentThread().getName() + ":" + count);  Thread.sleep(7);  } catch (InterruptedException e) {  e.printStackTrace();  }finally {  if(flag) {  lock.unlock();  }  }  **Condition**：实现了管程里的条件变量。如下实现队列：  public class BlockedQueue<T> {  private MyQueue queue = new MyQueue();  private final Lock lock = new ReentrantLock();  //条件：不满  final Condition notFull = lock.newCondition();  //条件：不空  final Condition notEmpty = lock.newCondition();  //入队  public void enq(T x){  lock.lock();  try {  while (queue.isFull()){  notFull.await();  }  queue.Enqueue();//入队操作  notEmpty.signal();//通知不为空条件等待队列的线程进入入口等待队列（即可以出队操作了）  }finally {  lock.unlock();  }  }  //出队  public void deq(){  lock.lock();  try {  while (queue.isEmpty()){  notEmpty.await();  }  queue.dequeue();//出队操作  notFull.signal();//通知不满条件等待队列的线程进入入口等待队列（即可以入队操作了）  }finally {  lock.unlock();  }  }  Dubbo中rpc的调用，异步转同步就是使用lock&condition实现，当调用rpc方法时，  Rt.await(),方法返回了后调用rt.signal()通知 |

### 读写锁-ReadWriteLock实现缓存

|  |
| --- |
| 理论上使用信号量或管程可以解决所有并发问题。但java并发包还是提供了适合不同场景的并发API并做了相应优化。如：读多写少的场景，缓存。  读写锁三个基本原则：   * 允许多个线程同时读共享变量 * 只允许一个线程写共享变量 * 如果一个写线程正在执行写操作,此时禁止读线程读共享变量。   **读写锁**和**互斥锁**的重要区别：读写锁允许多个线程同时读取共享变量,互斥锁不允许，这样在读多写少的场景读写锁性能**优于互斥锁**。但读写锁的写操作是互斥的。  读写锁实现缓存：如下  private final Map<K,V> map = new HashMap<K, V>();  private ReadWriteLock rw = new ReentrantReadWriteLock();  //读锁  final Lock r = rw.readLock();  //写锁  final Lock w = rw.writeLock();  public V get(K k){  r.lock();  try {  return map.get(k);  }finally {  r.unlock();  }  }  public V put(K k,V v){  w.lock();  try {  return map.put(k,v);  }finally {  w.unlock();  }  }  缓存一般如果数据量小，则一次性put入缓存中，如果数据量大则需要类似延迟加载功能，当缓存中没有数据时去数据库中查找并放入缓存。代码如下：  public V get(K k){  V v = null;  r.lock();  try {  v = map.get(k);  }finally {  r.unlock();  }  //缓存中有则返回  if(v != null){  return v;  }  //没有则,查询数据库  w.lock();  try {  //再次判断，有可能其他线程获取到了write锁并已经写入缓存并释放了锁，当前线程获取锁后需再次验证。  v = map.get(k);  if(v == null){  v = db.get(k);  map.put(k,v);  }  }finally {  w.unlock();  }  return v;  }  如上，再次判断，有可能其他线程获取到了write锁并已经写入缓存并释放了锁，当前线程获取锁后需再次验证。    在读锁还没有释放时又去获取写锁，叫做**锁的升级，但ReadWriteLock不支持，如果这样会导致写锁一直等待。**  **但，在获取写锁后，再获取读锁是允许的，称为：锁的降级，如下：**      ReetrantReadWriteLock读写锁的实现中，需要注意的，当有读锁时，写锁就不能获得；而当有写锁时，除了**获得写锁的这个线程可以获得读锁外，其他线程不能获得读锁。**  读锁可以被多个线程获取，但写锁和读锁互斥，即读锁被获取了，就无法获取到写锁，写锁被获取到了，除了当前线程，其他线程无法获取到读锁。 |

### 读写锁的改进-StampedLock

|  |
| --- |
| StampedLock比ReadWriteLock性能更优，StampedLock提供了三种模式：   * 写锁 * 悲观读锁 * 乐观读：乐观读这个操作是无锁的   写锁、悲观读锁的语义和ReadWriteLock的写锁、读锁的语义非常类似,允许多  个线程同时获取悲观读锁,但是只允许一个线程获取写锁,写锁和悲观读锁是互斥的。不同的是:StampedLock里的写锁和悲观读锁加锁成功之后,都会返回一个stamp;然后解锁的时候,需要传入这个stamp。相关的示例代码如下。  final StampedLock sl = new StampedLock();  // 获取/释放悲观读锁示意代码  long stamp = sl.readLock();  try{  //省略业务相关代码  }finally{  sl.unlockRead(stamp);  }  //获取/释放写锁示意代码  long stamp = sl.writeLock();  try{  //省略业务相关代码  }finally{  sl.unlockWrite(stamp);  }  StampedLock的**性能之所以比**ReadWriteLock还要好,**其关键是StampedLock支持乐观读的方式**。ReadWriteLock支持多个线程同时读,但是当多个线程同时读的时候,所有的写操作会被阻塞;而StampedLock提供的乐观读,是允许一个线程获取写锁的,也就是说不是所有的写操作都被阻塞。  使用乐观读如下：**先使用乐观读，读取共享数据，之后判断是否有其他线程执行写操作，如果有则升级为悲观读锁处理，否则直接返回共享数据。**  **long stamp = sl.tryOptimisticRead()：乐观读，**  **sl.validate(stamp)为false则有写操作。否则直接使用数据（数据正确性和唯一性得到保证）**    在上面这个代码示例中,如果执行乐观读操作的期间,存在写操作,会把乐观读升级为悲观读锁。这个做法挺合理的,否则你就需要在一个循环里反复执行乐观读,直到执行乐观读操作的期间没有写操作(只有这样才能保证x和y的正确性和一致性),而循环读会浪费大量的CPU。升级为悲观读锁,代码简练且不易出错,建议你在具体实践时也采用这样的方法。  **stampedLock注意：**   * 锁不可重入 * StampedLock的悲观读锁、写锁都不支持条件变量 * 如果线程阻塞在StampedLock的readLock()或者writeLock()上时,此时调用该阻塞线程的interrupt()方法中断线程,会导致CPU飙升。     所以,使用StampedLock一定**不要调用中断操作**,如果**需要支持中断**功能,一定使用可中断的**悲观读锁readLockInterruptibly()**和**写锁writeLockInterruptibly()** 。这个规则一定要记清楚。 |

### CountDownLatch和CyclicBarrier

|  |
| --- |
| 如下实例：对账系统：核对订单库 和 派送单 库数据是否有异常。    while(存在未对账订单){  //查询未对账订单  pos = getPOrders();  // 查询派送单  dos = getDOrders();  // 执行对账操作  diff = check(pos, dos);  // 差异写入差异库  save(diff);  }  传统使用单线程效率较低，考虑使用多线程。如下：**让查询订单和派送订单另起线程，使用join让主线程等待两个线程执行完后在进行对账操作**。  while(存在未对账订单){  //查询未对账订单  Thread T1 = new Thread(()->{  pos = getPOrders();  });  T1.start();  //查询派送单  Thread T2 = new Thread(()->{  dos = getDOrders();  });  T2.start();  //等待T1、T2结束  T1.join();  T2.join();  //执行对账操作  diff = check(pos, dos);  //差异写入差异库  save(diff);  }  上边操作存在问题：每循环一次都会new新的线程，太耗费资源。如下优化：   * **用CountDownLatch实现线程等待：类似计数器。** * **使用线程池**       ContDownLatch:**像门栓一样等待初始的值(如：3)为0了继续往下执行，用来 一个线程等待其他线程 的案例。线程结束时进行-1**  如上：两个查订单和 最后的 对账操作还是 串行化的，可以当对账时去并行执行下一次的查询操作，但必须保证两个查询操作一致（一一对应）。这样对账操作类似**消费者**，查询则为：**生产者**。可以将两个消费放入两个队列中    使用三个线程：T1，T2执行查询操作，但两个必须步调一致（先完成的线程需要等待后一个线程，这样保证队列一一对应。），等T1,T2都执行完了去通知T3执行对账操作。    //订单队列  Vector<P> pos;  //派送单队列  Vector<D> dos;//执行回调的线程池  Executor executor =  Executors.newFixedThreadPool(1);  final CyclicBarrier barrier =  new CyclicBarrier(2, () -> {  executor.execute(() -> check());  });  void check() {  P p = pos.remove(0);  D d = dos.remove(0);  //执行对账操作  diff = check(p, d);  //差异写入差异库  save(diff);  }  void checkAll() {  //循环查询订单库  Thread T1 = new Thread(() -> {  while (存在未对账订单) {  //查询订单库  pos.add(getPOrders());  //等待  barrier.await();  }  });  T1.start();  //循环查询运单库  Thread T2 = new Thread(() -> {  while (存在未对账订单) {  //查询运单库  dos.add(getDOrders());  //等待  barrier.await();  }  });  T2.start();  }  如上CyclicBarrier 类会当计数器为0时会回调一个函数（对账操作），为0时会重置计数器为初始值2，上边对账操作线程池大小只有1，因为：如果是多个，可能前一个对账操作还没有完成，订单队列中可能只移除了一个，如果下一个对账操作也去执行，取出的两个查询订单数据不是一一对应的。 |
| CountDownLatch和CyclicBarrier是Java并发包提供的两个非常易用的线程同步工具类,这两个工具类用法的区别在这里还是有必要再强调一下:**CountDownLatch主要用来解决一个线程等待多个线程的场景** ,可以类比旅游团团长要等待所有的游客到齐才能去下一个景点;而**CyclicBarrier是一组线程之间互相等待** ,更像是几个驴友之间不离不弃。除此之外CountDownLatch的计数器是不能循环利用的,也就是说一旦计数器减到0,再有线程调用await(),该线程会直接通过。但CyclicBarrier的计数器是可以循环利用的 ,而且具备自动重置的功能,一旦计数器减到0会自动重置到你设置的初始值。除此之外,CyclicBarrier还可以设置回调函数,可以说是功能丰富。  总结：  CountDownLatch在调用latch.await()时等计数器的值为0后继续执行。  CyclicBarrier会等计数器的值为0时由最后一个执行完的线程调用回调函数（也可以不调），在其他线程中调用await()方法计数器会-1，线程中如果计数器不为0则线程等待其他线程。为0后会重置为初始值。 |

### Phaser

|  |
| --- |
| 阶段性的栅栏。需要继承Phaser实现方法来使用。 |

### Exchanger两个线程交换数据

|  |
| --- |
| 两个线程进行数据交换，T1调用如：如果T1线程先执行了exchanger.exchange(s);  会阻塞等待另一个线程也调用exchange()方法进行数据交换，之后两个线程继续执行。    如下列子：  Exchanger<String> exchanger = new Exchanger<>();  new Thread(new Runnable() {  @Override  public void run() {  String s = "t1";  try {  s = exchanger.exchange(s);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + ":s = " + s);  }  },"t1").start();  new Thread(new Runnable() {  @Override  public void run() {  String s = "t2";  try {  s = exchanger.exchange(s);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.out.println(Thread.currentThread().getName() + ":s = " + s);  }  },"t2").start(); |

### 并发容器

|  |
| --- |
| Java容器有：list,set,map,要将非线程安全的容器改为线程安全的容器：  **只要把非线程安全的容器封装在对象内部,然后控制好访问路径就可以了。如：**    如上这种包装可以使用Collections来完成如下：  List list = Collections.synchronizedList(new ArrayList());  Set set = Collections.synchronizedSet(new HashSet());  Map map = Collections.synchronizedMap(new HashMap());  在容器领域一个容易被忽视的“坑”是用**迭代器遍历容器。需要加锁**  List list=Collections.  synchronizedList(new ArrayList());  synchronized(list){  Iterator i = list.iterator();  while (i.hasNext())  foo(i.next());  }  jdk1.5之前也就类似上面的实现如：Vector、Stack和Hashtable，这种同步容器性能比较差。Jdk1.5后提供了性能更好的并发容器：     * List:CopyOnWriteArrayList 写时将共享变量拷贝一份，这样做的好处是：读操作无锁。CopyOnWriteArrayList迭代器是**只读**的,不支持**增删改**。因为它遍历的只是一个快照。     如果遍历array时，还有一个写操作（如添加元素），则将array拷贝一份，在新数组上进行添加元素，执行完后将array指向新的数组。    **注意**：   * CopyOnWriteArrayList 只适应与写非常少，并且能容忍读写的短暂不一致（如上面不能遍历到新添加的元素） * CopyOnWriteArrayList迭代器是只读的,不支持增删改。因为它遍历的只是一个快照。 * Map:ConcurrentHashMap和ConcurrentSkipListMap   ConcurrentHashMap的key是**无序的**,而ConcurrentSkipListMap(**跳表实现**)的key是**有序的**     * Set:CopyOnWriteArraySet和ConcurrentSkipListSet,使用如上差不多。 * Queue * 阻塞和非阻塞：当队列已满时,入队操作阻塞；当队列已空时,出队操作阻塞（阻塞队列都用Blocking关键字标识） * 单端与双端 ,单端指的是只能队尾入队,队首出队;而双端指的是队首队尾皆可入队出队   （单端队列使用Queue标识,双端队列使用Deque标识） |

## 并发编程-bug源头

### 可见性-缓存带来的问题

|  |
| --- |
| 最初的单核CPU,由于只有一个核心，cpu缓存也就只有一个，所以数据的对各个线程都是可见的，不存在问题，一个线程修改了，其他线程读取的是同一缓存的数据。如下：    一个线程对共享变量的修改，另外一个线程能够立刻看到，我们称为**可见性**。  多核时代：每个核心都有自己的缓存 |

### 原子性-线程切换

|  |
| --- |
| **线程切换：**    **我们把一个或者多个操作在CPU执行的过程中不被中断的特性称为原子性**  如：count+=1 能需要三条cpu指令。如：   * 指令1：首先，需要把变量count从内存加载到CPU的寄存器； * 指令2：之后，在寄存器中执行+1操作； * 指令3：最后，将结果写入内存（缓存机制导致可能写入的是CPU缓存而不是内存）。   如果是在指令1后切换到 其他线程 可能导致数据不正确    CPU能保证的原子操作是CPU指令级别的，而不是高级语言的操作符 |

### 编译器优化-有序性问题

|  |
| --- |
| 编译器优化可能带来有序性问题。  如：  a = 7;  b = 8;  优化后可能先执行 b = 8 后执行 a = 7;  这样就可能出现问题，如下单例写法问题：  public class Singleton {  static Singleton instance;  static Singleton getInstance(){  if (instance == null) {  synchronized(Singleton.class) {  if (instance == null)  instance = new Singleton();  }  }  return instance;  }  }  由于编译器优化，我们以为的new操作应该是：   1. 分配一块内存M； 2. 在内存M上初始化Singleton对象； 3. 然后M的地址赋值给instance变量。   但是实际上优化后的执行路径却是这样的：   1. 分配一块内存M； 2. 将M的地址赋值给instance变量； 3. 最后在内存M上初始化Singleton对象。   当线程A在执行到 2，给instance 赋了值后发生线程切换，B线程调用 getInstance()方法，判断instance ！ = null,就去返回了对象，但其实对象 并没有初始化完成，就会出现问题（如：空指针） |

### 如何解决并发bug

**方法包括：volatile**、**synchronized** 和 **final** 三个关键字，以及六项 **Happens-Before 规则**

### 可见性和有序性的解决

|  |
| --- |
| 可见性可以使用 volatile修饰来使数据被各个线程所见。如下：  // 以下代码来源于【参考1】  class VolatileExample {  int x = 0;  volatile boolean v = false;  public void writer() {  x = 42;  v = true;  }  public void reader() {  if (v == true) {  // 这里x会是多少呢？  }  }  }  分别让两个线程去执行 writer和 reader,reader方法 中 当 v==true时，x会是多少呢？  Jdk1.5之前 x可能是0 也可能是 42  0是因为 线程2使用的本地存储/缓存 中的副本，也可能副本已被跟新所以为42  但在jdk1.5之后 x = 42,1.5之后增加了Happens-Before 规则  Happens-Before 规则：**前面一个操作的结果对后续操作是可见的** 程序的顺序性规则 如下：  X = 42;  V = true;  X = 42；的操作对v = true；操作是可见的 volatile变量规则 一个volatile变量的写操作， Happens-Before 于后续对这个volatile变量的读操作，单一这条规则这和禁用缓存很像 传递性 如果 **A** Happens-Before **B ,B** Happens-Before **C 那么：A** Happens-Before **C**  如上边的代码：  因为 writer中 x=42 Happens-Before v = true;又因为 规则2  V = true的写 Happens-Before reader方法中 读 v == true,所以  **X = 42 Happens-Before v == true;即：此处 x = 42是可见的**   管程中锁的规则 这条规则是指对一个锁的解锁 Happens-Before 于后续对这个锁的加锁。  **管程**是一种通用的同步原语，在Java中指的就是synchronized,synchronized是Java里对管程的实现。  如：  synchronized (this) { //此处自动加锁  // x是共享变量,初始值=10  if (this.x < 12) {  this.x = 12;  }  } //此处自动解锁  意思在线程1 对共享变量x设置为12后，线程2在进入代码块时，能够看到线程1对x的修改，即可以看到x = 12 线程 start() 规则 这条是关于线程启动的。它是指主线程A启动子线程B后，子线程B能够看到主线程在启动子**线程B前的操作**。  换句话说就是，如果线程A调用线程B的 start() 方法（即在线程A中启动线程B），那么该start()操作 Happens-Before 于线程B中的任意操作。具体可参考下面示例代码。  Thread B = new Thread(()->{  // 主线程调用B.start()之前  // 所有对共享变量的修改，此处皆可见  // 此例中，var==77  });  // 此处对共享变量var修改  var = 77;  // 主线程启动子线程  B.start();  主线程中调用了B.start()，所以 在线程B中可以获得 var = 77这个值 线程 join() 规则 这条是关于线程等待的。它是指主线程A等待子线程B完成（主线程A通过调用子线程B的join()方法实现），当子线程B完成后（主线程A中join()方法返回），主线程能够看到子线程的操作。当然所谓的“看到”，指的是对**共享变量**的操作。  Thread B = new Thread(()->{  // 此处对共享变量var修改  var = 66;  });  // 例如此处对共享变量修改，  // 则这个修改结果对线程B可见  // 主线程启动子线程  B.start();  B.join()  // 子线程所有对共享变量的修改  // 在主线程调用B.join()之后皆可见  // 此例中，var==66  主线程A 调用 join等待子线程B,如果join操作正常返回，那么线程B中的操作 在join返回后的主线程中可见，即：共享变量数据可见  在Java语言里面，Happens-Before的语义本质上是一种可见性，A Happens-Before B 意味着A事件对B事件来说是可见的，无论A事件和B事件是否发生在同一个线程里。例如A事件发生在线程1上，B事件发生在线程2上，Happens-Before规则保证线程2上也能看到A事件的发生。 |

### 原子性的解决

|  |
| --- |
| 原子性问题是由于 **线程切换导致 解决方法：同一时间只有一个线程在执行**  “**同一时刻只有一个线程执行**”这个条件非常重要，我们称之为**互斥**。  如果我们能够保证对共享变量的修改是互斥的，那么，无论是单核CPU还是多核CPU，就都能保证原子性了。    我们把一段需要互斥执行的代码称为**临界区** Java语言提供的锁技术：synchronized 当修饰静态方法的时候，锁定的是当前类的Class对象，在上面的例子中就是Class X； 当修饰非静态方法的时候，锁定的是当前实例对象this。  **如下:**  class SafeCalc {  long value = 0L;  long get() {  return value;  }  synchronized void addOne() {  value += 1;  }  }  两个线程分别执行 addOne和 get方法，由于 happens-before原则：管程中的锁规则  在其他线程调用 addOne方法 value是可见的，但get不可见，所以也需要加锁。 锁和受保护资源的关系 **受保护资源和锁之间的关联关系是N:1的关系**。  一个资源不能被多把锁保护  如下会出现并发问题：  class SafeCalc {  static long value = 0L;  synchronized long get() {  return value;  }  synchronized static void addOne() {  value += 1;  }  }  没有关联的资源：  **用不同的锁对受保护资源进行精细化管理，能够提升性能**。这种锁还有个名字，叫**细粒度锁**。  class Account {  // 锁：保护账户余额  private final Object balLock = new Object();  // 账户余额  private Integer balance;  // 锁：保护账户密码  private final Object pwLock = new Object();  // 账户密码  private String password;  // 取款  void withdraw(Integer amt) {  synchronized(balLock) {  if (this.balance > amt){  this.balance -= amt;  }  }  }  // 查看余额  Integer getBalance() {  synchronized(balLock) {  return balance;  }  }  // 更改密码  void updatePassword(String pw){  synchronized(pwLock) {  this.password = pw;  }  }  // 查看密码  String getPassword() {  synchronized(pwLock) {  return password;  }  }  } 保护有关联关系的多个资源 如下会出现编发问题：转账问题  class Account {  private int balance;  // 转账  synchronized void transfer(  Account target, int amt){  if (this.balance > amt) {  this.balance -= amt;  target.balance += amt;  }  }  }  如果有如下转账：线程1执行：A转B 100，线程2：B转C 100,初始化都为200  则：线程1锁定的对象为A.this,线程2锁定的对象为B.this,这样不能保证并发问题    **用Account.class作为共享的锁**。  class Account {  private int balance;  // 转账  void transfer(Account target, int amt){  synchronized(Account.class) {  if (this.balance > amt) {  this.balance -= amt;  target.balance += amt;  }  }  }  }    **解决原子性问题，是要保证中间状态对外不见** |

### 编发编程需要注意的问题

|  |
| --- |
| 需注意一下问题：   * **安全性问题：正确性，程序按照我们预期执行。**   当有多个线程会同时读写同一数据，存在数据共享并且数据状态不发生改变。  使用锁来解决。   * **活跃性问题：指操作无法执行下去，如：死锁，活锁，饥饿**   活锁：如：两个人一直互相谦让。解决办法：谦让时，尝试等待一个随机的时间就可以了  饥饿：线程因无法访问所需资源,如：线程的优先级不均，可能导致优先级低的线程无法别执行，持有锁的线程，如果执行的时间过长，也可能导致“饥饿”问题。  解决“饥饿”问题的方案很简单，有三种方案：一是保证资源充足，二是**公平地分配资源**，三就是避免持有锁的线程长时间执行。这三个方案中，方案一和方案三的适用场景比较有限，因为很多场景下，资源的稀缺性是没办法解决的，持有锁的线程执行的时间也很难缩短。倒是方案二的适用场景相对来说更多一些。  那如何公平地分配资源呢？在并发编程里，主要是使用**公平锁**。所谓公平锁，是一种先来后到的方案，线程的等待是有顺序的，排在等待队列前面的线程会优先获得资源。   * **性能问题：**   第一，既然使用锁会带来性能问题，那最好的方案自然就是**使用无锁的算法和数据结构**了。在这方面有很多相关的技术，例如线程本地存储(Thread Local Storage, TLS)、写入时复制(Copy-on-write)、乐观锁等；Java并发包里面的原子类也是一种无锁的数据结构；Disruptor则是一个无锁的内存队列，性能都非常好……  第二，**减少锁持有的时间**。互斥锁本质上是将并行的程序串行化，所以要增加并行度，一定要减少持有锁的时间。这个方案具体的实现技术也有很多，例如使用细粒度的锁，一个典型的例子就是Java并发包里的ConcurrentHashMap，它使用了所谓分段锁的技术（这个技术后面我们会详细介绍）；还可以使用读写锁，也就是读是无锁的，只有写的时候才会互斥。  性能方面的度量指标有很多，我觉得有三个指标非常重要，就是：吞吐量、延迟和并发量。  吞吐量：指的是单位时间内能处理的请求数量。吞吐量越高，说明性能越好。  延迟：指的是从发出请求到收到响应的时间。延迟越小，说明性能越好。  并发量：指的是能同时处理的请求数量，一般来说随着并发量的增加、延迟也会增加。所以延迟这个指标，一般都会是基于并发量来说的。例如并发量是1000的时候，延迟是50毫秒。 |
| 总结 并发编程是一个复杂的技术领域，微观上涉及到原子性问题、可见性问题和有序性问题，宏观则表现为安全性、活跃性以及性能问题。 |

### 解决并发的万能钥匙-管程

|  |
| --- |
| 管程：**指的是管理共享变量以及对共享变量的操作过程，让他们支持并发**  翻译为Java领域的语言，就是管理类的成员变量和成员方法，让这个类是线程安全的。  Java管程模型图：    Synchronized实现的管程只支持一个条件变量，如Synchronized(this),this就是这个条件变量。  但lock和condition实现的管程可以有多个条件变量。  条件变量满足才能去执行对共享变量的操作（方法），否则需要在该条件变量对象的等待队列中等待。可以定义一个对象如：A来表示一个条件变量，那么T1线程A.wait()则会去等待队列中等待，当其T2线程A.notify()通知条件变量中的T1条件满足了，T1会重新进入入口等待队列中等待进入管程，但可能这是A条件已经不满足了,所以需要while循环验证条件变量。  如上实现非阻塞队列：  public class BlockedQueue<T> {  private MyQueue queue = new MyQueue();  private final Lock lock = new ReentrantLock();  //条件：不满  final Condition notFull = lock.newCondition();  //条件：不空  final Condition notEmpty = lock.newCondition();  //入队  public void enq(T x){  lock.lock();  try {  while (queue.isFull()){  notFull.await();  }  queue.Enqueue();//入队操作  notEmpty.signal();//通知不为空条件等待队列的线程进入入口等待队列（即可以出队操作了）  }finally {  lock.unlock();  }  }  //出队  public void deq(){  lock.lock();  try {  while (queue.isEmpty()){  notEmpty.await();  }  queue.dequeue();//出队操作  notFull.signal();//通知不满条件等待队列的线程进入入口等待队列（即可以入队操作了）  }finally {  lock.unlock();  }  } |

### 信号量

|  |
| --- |
| 知道管程出现之前，**信号量**一直是并发编程的**终结者**。  信号量模型：一个计数器，一个等待队列，三个方法。计数器和等待队列是透明的（看不到），只能通过三个方法访问。Init(),down(),up()     * Init()：设置计数器的初始值。 * down():计数器减1；如果计数器的值小于0，则当前线程被阻塞，否则当前线程继续执行。 * up()：计数器的值加1;如果此时计数器的值小于或者等于0,则唤醒等待队列中的一个线程,并将其从等待队列中移除   **这三个方法都是原子性的**。Java中信号量模型由java.util.concurrent.Semaphore实现。  上边模型如下边代码：    在Java SDK并发包里,down()和up()对应的则是acquire()和release()。  如下使用信号量模型实例：  public class SemaphoreTest {  static int *count*;  static final Semaphore *sh* = new Semaphore(1);  public static void addOne(){  try {  *sh*.acquire();  } catch (InterruptedException e) {  e.printStackTrace();  }  try {  *count* += 1;  }finally {  *sh*.release();  }  }  public static void main(String[] args) {  for (int i = 0; i < 100; i++) {  new Thread(SemaphoreTest::*addOne*,i + **"线程"**).start();  }  try {  Thread.*sleep*(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println(*count*);   }   }  如上执行过程：线程T1和线程T2,同时执行addOne()方法，由于acquire()是原子操作，如T1先进入，计数器-1=0，由于不小于0所以当前线程继续执行，T1执行acquire()完后，T2执行，计数器-1=-1，小于0，T2线程阻塞。T1执行完count+=1操作后，执行  Release()方法，计数器+1=0，等于0唤醒等待队列中的T2,并将其从等待队列中移除，T2也进行count+=1操作。**于是T2在T1执行完临界区代码之后才获得了进入临界区执行的机会,从而保证了互斥性**。  上面的例子,我们用信号量实现了一个最简单的互斥锁功能。估计你会觉得奇怪,既然有Java SDK里面提供了Lock,为啥还要提供一个Semaphore ?其实实现一个互斥锁,仅仅是Semaphore的部分功能,Semaphore还有一个功能是Lock不容易实现的,那就是:**Semaphore可以允许多个线程访问一个临界区** 。（如：连接池）  信号量的计数器,在上面的例子中,我们设置成了1,**这个1表示只允许一个线程进入临界区**,但如果我们把计数器的值设置成对象池里对象的个数N,就能完美解决对象池的限流问题了。  如下：  public class ObjPool<T,R> {  final List<T> pool;  // 用信号量实现限流器  final Semaphore sem;  // 构造函数  public ObjPool(int size,T t){  pool = new Vector<T>(){};  for(int i=0;i<size;i++){  pool.add(t);  }  sem = new Semaphore(size);  }  // 利用对象池的对象,调用func  public R exec(Function<T,R> func){  T t = null;  try {  sem.acquire();  } catch (InterruptedException e){  e.printStackTrace();  }  try{  t = pool.remove(0);  return func.apply(t);  }finally{  pool.add(t);  sem.release();  }  }  public static void main(String[] args) {  ObjPool<Long,String> pool = new ObjPool<Long,String>(10,2l);  //通过对象池获取t,之后执行  pool.exec(t -> {  System.out.println(t);  return t.toString();  });    }  } |

### 衡量多线程性能（多少线程合适）

|  |
| --- |
| 多线程的性能衡量：延迟和吞吐量  多核时代，当cup和IO的利用率不高时，就可以通过多线程来提高性能。   * **I/O密集型计算**：I/O操作执行的时间相对于CPU计算来说都非常长   大体可以如公式：最佳线程数=CPU核数 \* [ 1 +（I/O耗时 / CPU耗时）]   * **Cpu密集型计算**：CPU执行的时间相对于I/O操作来说都非常长   对于CPU密集型的计算场景，理论上“线程的数量=CPU核数”就是最合适的。不过在工程上，线程的数量一般会设置为“CPU核数+1” |

## Java内存模型

|  |
| --- |
| Jmm是一个抽象的逻辑模型，对应jvm的实现为：主要为堆（主内存），栈（工作内存）    局部变量（基本类型（int,boolean...）,引用类型（引用））存储在栈内存中。而成员变量会随对象本身存储在堆中。  如上对应硬件：主内存对应内存，工作内存（**缓存、写缓冲区、寄存器以及其他的硬件和编译器优化**） |

## JVM

|  |
| --- |
|  |

## 类图

|  |
| --- |
| 继承：实线空心三角行，如下：A继承B    实现：虚线空心三角形，如下：自行车实现 抽象类车    聚合：空心菱形箭头的直线表示，如下：B由A组成，但并非强依赖，如：人员组成部门，但，但部门不存在了 人员依旧存在    组合：一条带实心菱形箭头直线表示，强依赖，整体不存在了 部分也就不存在了，部门组成公司，但公司不存在了 部门也就不存在了。    关联关系：带箭头的实线表示    依赖关系：带箭头的虚线表示 |

## IDEA

### Idea全局配置

|  |
| --- |
| File----> other settings ---> deafult settings |

### Idea初始设置

|  |
| --- |
| * 设置安装目录 * 设置主题（白/黑） * 设置字体样式/编码：包括控制台，编码区        * 鼠标滚轮改变字体      * 显示行号，方法分割线      * 补全功能（首字母不区分大小写）      * 拷贝其他路径代码 自动导包功能      * 鼠标放置类名/方法处给出提示 |

### JDK配置

|  |
| --- |
|  |

### 显示工具栏

|  |
| --- |
| 选上 View ---->ToolBar 和 Tool Buttons |

### IDEA JVM参数

|  |
| --- |
| 进入：Help -----> Edit Custom VM Options  如下修改 |

### idea常用快捷键

|  |
| --- |
| ctrl+shift+u：大小写转换  F2:定位到报错位置  Alt+enter:提示/错误提示，引入类等  Alt+Insert:setter/getter,构造方法，方法重写等快捷键  Shift+Shift:全局搜索  Ctrl+Alt+v:自动补全方法返回值类型和变量的  Iter:foreach循环  Itli:for循环集合  Itar:for循环数组  Ctrl+y:删除一行  Ctrl+W:按一次为选中光标所在单词，再按为这一行，再按．．．  Ctrl/alt+Shift+上下键：选中或所在行代码上移或下移  Alt+/：复制光标前一个单词  Ctrl+P:查看方法参数  **Ctrl+H：查看类的继承关系**  Ctrl + Shift + N：搜索文件在项目中  Ctrl+N:搜索类 |

### idea和eclipse比较

|  |
| --- |
| Eclipse idea  工作空间　　相当于　　项目  项目　　　　相当于　　模块  所以可以先新建　项目　再在项目下新建　多个模块  如：新建一个项目【java类别】－－》新建模块【java】--->给模块添加框架【如：maven】 |
|  |

### idea 中maven不能自动补全已经下载到本地的jar

|  |
| --- |
| 更新仓库 |

### Idea maven Junit测试

|  |
| --- |
| Pom中加入junit依赖的jar包，编写测试代码，如:  public class HdfsTest {  **@Test** public void hdfsApi() throws Exception {  Configuration conf = new Configuration();  conf.set(**"fs.defaultFS"**, **"hdfs://192.168.134.100:9000"**);  FileSystem fs = FileSystem.*get*(conf);  boolean success = fs.mkdirs(new Path(**"/test"**));  System.*out*.println(success);  }  } |

### idea 打包可运行的jar包

|  |  |
| --- | --- |
| 1. 原始方法：   File----->Project Project Structure－＞Artifacts－＞+ －＞Jar－＞From modules with dependencies　－＞选择模块和main方法类－＞设置Directory for META.INF(此处要选择到项目根目录的上一级)－＞OK  路径无误后会output layout中看到****META.INF/MAINFEST.MF文件****  ****Build－＞****Build Artifacts－＞****Build　就行了****   1. ****使用插件maven－shade－plugin加入插件****  |  | | --- | | <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-shade-plugin</artifactId>  <version>3.2.0</version>  <executions>  <execution>  <phase>package</phase>  <goals>  <goal>shade</goal>  </goals>  <configuration>  <transformers>  <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">  <mainClass>com.space.ServicePublisher</mainClass>  </transformer>  </transformers>  </configuration>  </execution>  </executions>  </plugin>  </plugins>  </build> |   ****修改main方法类然后清理打包即可****  更多内容：https://www.jianshu.com/p/7a0e20b30401 |

### 插件安装

|  |
| --- |
| 如下图安装： |

### Idea DEBUG

|  |
| --- |
|  |

### 阿里代码规范插件

|  |
| --- |
| Alibaba Java Coding Guidelines |

## Mybatis

### 优缺点

|  |
| --- |
| Hibernate 是一个 全自动 ORM框架，sql的执行过程开发人员是无法 修改 掌控的。  Mybatis是半自动，轻量级的 |

### 使用-老式配置文件方式

|  |
| --- |
| * 添加jar包或maven依赖（mybatis,mysql驱动，其他jar）   <dependency>  <groupId>org.mybatis</groupId>  <artifactId>mybatis</artifactId>  <version>3.4.1</version>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <scope>test</scope>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.25</version>  </dependency>   * 添加 全局配置文件mybatis-config.xml 和 正确填写 数据库连接信息   <configuration>  <environments default="development">  <environment id="development">  <transactionManager type="JDBC"/>  <dataSource type="POOLED">  <property name="driver" value="com.mysql.jdbc.Driver"/>  <property name="url" value="jdbc:mysql://localhost:3306/mybatis"/>  <property name="username" value="root"/>  <property name="password" value="root"/>  </dataSource>  </environment>  </environments>  <mappers>  <mapper resource="PersonMapper.xml" ></mapper>  </mappers>  </configuration>   * 编写要执行的sql的映射文件PersonMapper.xml，将xml加入到全局配置文件中。   <?xml version="1.0" encoding="UTF-8" ?>  <!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">  <mapper namespace="com.space.person">  <select id="selectPerson" resultType="com.space.bean.Person">  select \* from persons where id = #{id}  </select>  </mapper>   * 获取sessionSql,传入执行sql的唯一ID和sql参数，查询   String resource = "mybatis-config.xml";  InputStream inputStream = Resources.getResourceAsStream(resource);  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  SqlSession sqlSession = sqlSessionFactory.openSession();  Person person = sqlSession.selectOne("com.space.person.selectPerson", 1);  System.out.println(person);  这种方式 查询时 容易写错 sqlId 传入的参数也可能不对。不安全 |

### 使用-新式 接口（动态代理）

|  |
| --- |
| //定义一个 需要查询的接口  public interface PersonMapper {  Person selectPerson(Integer id);  }  <mapper namespace="com.space.bean.PersonMapper">  <select id="selectPerson" resultType="com.space.bean.Person">  select \* from persons where id = #{id}  </select>  </mapper>  Mapper Xml中 namespace 为接口的全类名，sqlID为方法名。  如下查询：  String resource = "mybatis-config.xml";  InputStream inputStream = Resources.getResourceAsStream(resource);  SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  SqlSession sqlSession = sqlSessionFactory.openSession();  PersonMapper mapper = sqlSession.getMapper(PersonMapper.class);//传入接口的class  Person person = mapper.selectPerson(1);//调用，由框架动态代理自动生成 接口的 实现类  System.out.println(person);  sqlSession.close();  框架动态代理自动生成 接口的 实现类  SqlSession 是非线程安全的 不要把他 放置为成员变量，随用随关。 |

### 全局配置文件

mybatis-config.xml

|  |
| --- |
| 1. <properties 标签 引入外部 properties 文件   <properties resource="db.properties" ></properties>，resource为类路径中，url为网络路径  <environments default="development">  <environment id="development">  <transactionManager type="JDBC"/>  <dataSource type="POOLED">  <property name="driver" value="${driver}"/>  <property name="url" value="${url}"/>  <property name="username" value="${username}"/>  <property name="password" value="${password}"/>  </dataSource>  </environment>  </environments>   1. <settings> 运行时行为 参数设置 2. <typeAliases>单个别名定义，别名不区分大小写   <typeAliases>  <typeAlias type="com.space.bean.Person" alias="\_Person"/>  </typeAliases>   1. 批量别名，如下：   <package name="com.space.bean"/> 别名默认为 类名，不区分大小写   1. 注解别名：@Alias(“xxx”)     如下为mybatis内别名，注意用户自己的别名和内置的别重复。     1. typeHandlers类型处理器，java和数据库类型的对应设置。 2. <environments>环境配置   <environments default="development">  <!--测试环境-->  <environment id="test">  <transactionManager type=""></transactionManager>//事务管理JDBC和MANAGE和自定义  <dataSource type=""></dataSource>  </environment>  <!--开发环境-->  <environment id="development">  <transactionManager type="JDBC"/>  <dataSource type="POOLED">  <property name="driver" value="${driver}"/>  <property name="url" value="${url}"/>  <property name="username" value="${username}"/>  <property name="password" value="${password}"/>  </dataSource>  </environment>  </environments>     1. databaseIdProvider 多数据库支持   <databaseIdProvider type="DB\_VENDOR">  <property name="SQL Server" value="sqlserver"/>  <property name="Msql" value="mysql"/>  <property name="Oracle" value="oracle" />  </databaseIdProvider>  <mapper namespace="com.space.bean.PersonMapper">  <select id="selectPerson" resultType="person" databaseId="mysql">  select \* from persons where id = #{id}  </select>  </mapper>   1. <mapper>,注册mapper映射文件，   <mapper resource="PersonMapper.xml" ></mapper>//xml设置  //接口设置，但xml映射文件必须和接口在同一包下，并且同名。  <mappers>  <mapper class="com.space.bean.PersonMapper"></mapper>  </mappers>  基于注解的查询：不太推荐  public interface PersonMapperAnno {  @Select("select \* from persons where id = #{id}")  Person getPerson(Integer id);  }  //全局配置文件  <mappers>  <mapper class="com.space.bean.PersonMapperAnno" ></mapper>  </mappers>  批量注册：  <package name="com.space.bean"></package>  **配置文件中标签是有顺序的** |

### 映射文件

|  |
| --- |
| 注意：  SqlSession session = sqlSessionFactory.openSession();  不带参的这种获取 session 数据不会自动提交需要 手动提交：  session.commit();  增删改的返回值可以允许为：Integer,Long,Boolean,void   1. 如下增删该查：   <select id="selectPerson" resultType="person" >  select \* from persons where id = #{id}  </select>  <insert id="insertPerson" parameterType="person">  insert into persons(name,phone)  values(#{name},#{phone})  </insert>  <update id="updatePerson" parameterType="person">  update persons set name=#{name},phone=#{phone}  where id=${id}  </update>  <delete id="deletePerson" parameterType="person">  delete from persons where id=${id}  </delete>  接口：  Person selectPerson(Integer id);  void insertPerson(Person person);  void updatePerson(Person person);  void deletePerson(Person person);   1. 使用mybatis自增主键策略   <insert id**="insertPerson"** parameterType**="person"** useGeneratedKeys**="true"** keyProperty**="id"**>  insert into persons(name,phone)  values(#{name},#{phone}) </insert>  如上 设置 userGeneratedKeys = “true” 启动自增主键策略，并设置 对应自增的字段  keyProperty = “id”,如下测试可知 主键 由mybatis生成  Person person = new Person();  person.setName("cc");  person.setPhone("12345678909");  PersonMapper mapper = session.getMapper(PersonMapper.class);  mapper.insertPerson(person);  System.out.println(person.getId());  session.commit();  session.close();  上边 person的id 可获取到  上边针对 有自增功能的数据库如：mysql   1. Oracle需要使用 序列 实现自增：   <insert id="insertPersonToOracle" parameterType="person">  <selectKey keyProperty="id" order="BEFORE" resultType="Integer">  select EMPLOYEES\_SEQ.nextval from dual  </selectKey>  insert into persons  values(#${id},#{name},#{phone})  </insert>  <selectKey> 中 order=”BEFORE” 意思为 在 sql之前 运行，把 结果封装 bean的 id属性中。   1. 参数   当传入一个参数时，映射文件中 sql ,#{xx},xx为任意  如下 多个参数时，mybatis会把多个参数封装长map,key默认为：  param1, param2.....,如下这种写法 会报错。  Person selectByIdAndName(Integer id,String name);  select \* from persons where id = #{id}  and name =#{name}  可以 预先制定map的key,如下使用注解：@Param(“key”)  Person selectByIdAndName(**@Param**(**"id"**)Integer id, **@Param**(**"name"**) String name);  **还可以 直接传入 Map,映射文件直接使用 key**  如果参数是 集合类型/数组，mybatis会特殊处理，如：  Collection类型 key为：collection  List key为：list  数组 key 为：array  使用：如：array[0]   1. 参数的获取#{xx}，${xx} 区别   #{xx}：使用占位符 ？，预编译形式。像jdbc:PreparedStatment，防止sql注入。  ${xx}：直接拼接值到sql中。  大多数 情况 应该使用 #{xx}  原生jdbc不支持占位符的 可以使用 ${xx},如：分表，按照年份拆分  Select \* from ${year}\_table where xxx  动态字段排序：  Select \* from table order by ${c\_name} ${order}  #{}更丰富的用法：制定参数类型  #{property,javaType=int,jdbcType=NUMERIC}   1. 返回map类型   如果返回key为列名，value:为 值的一条数据，则返回 类型 设置为 map就行。如下：  <select id="selectListByName" resultType="map">  select \* from persons  where name like #{name}  </select>  如果返回 多条数据，将每一条数据 封装成一个bean 放入map中,map的key指定bean中的一列，如下：使用注解  <select id**="selectListByNameOfAnn"** resultType**="person"**>  select \* from persons  where name like #{name} </select>  返回值类型设置为 bean类型，接口处使用注解 @MapKey(“id”)  **@MapKey**(**"id"**) Map<Integer,Person> selectListByNameOfAnn(String name);   1. 自定义结果集 resultMap   <select id="selectListByNameOfAnn" resultMap="MyPerson" >  select \* from persons  where name like #{name}  </select>    <resultMap id="MyPerson" type="com.space.bean.Person" >  <id column="id" property="id"/>  <result column="name" property="name"/>  <result column="phone" property="phone"/>  </resultMap>   1. 一对多联合查询   <resultMap id="Users\_order" type="com.space.bean.User">  <id column="id" property="id"/>  <result column="name" property="name"/>  <result column="age" property="age"/>  <collection property="orders" ofType="com.space.bean.Order">  <id column="order\_id" property="id"/>  <result column="orderno" property="orderNo"/>  </collection>  </resultMap>   1. 一对一联合查询：使用association   <resultMap id**="Users\_order"** type**="com.space.bean.User"**>  <id column**="id"** property**="id"**/>  <result column**="name"** property**="name"**/>  <result column**="age"** property**="age"**/>  <association property**="addr"** javaType**="com.space.bean.Addr"**>  <id column**="addr\_id"** property**="id"**/>  <result column**="city"** property**="city"**/>  </association> </resultMap>   1. 分步联合查询：   <association property="addr" select="com.space.dao.UserMapper.selectAddrById" column="addr\_id">  </association>  <select id="selectAddrById" resultType="com.space.bean.Addr">  select \* from addr where id = #{id}  </select>  select u.id id  ,u.name name  ,u.age age  ,u.addr\_id addr\_id  ,o.id order\_id  ,o.orderno orderno  from users u,orders o  where u.id = o.uid  使用其他查询语句 去查，如上 column为下边查询sql的参数，有sql语句中对应的字段提供。分步查询可以使用延迟加载,在用的时候再去查询  配置：全局配置文件中配置mybatis-config.xml  <setting name**="lazyLoadingEnabled"** value**="true"**/> <setting name**="aggressiveLazyLoading"** value**="false"**/>###false时则使用到时才去加载  如下使用后才去查询    分步查询可以使用延迟加载  <association property**="addr"** select**="com.space.dao.UserMapper.selectAddrById"** column**="addr\_id"**> </association>  <collection property**="orders"** select**="com.space.dao.UserMapper.selectOrderByUserId"** column**="id"**> </collection>  多列值传递给 联合查询语句：{key1=column1,key2=column2}  <collection  property="orders"  select="com.space.dao.UserMapper.selectOrderByUserId" column="{uid=id,uname=name}"  >  </collection>  设置collection延迟加载方式：   * 全局设置如上边association * 局部设置，这个优先于全局设置起作用，fetchType=“lazy”默认，eager为预先加载   <collection property="orders"  select="com.space.dao.UserMapper.selectOrderByUserId"  fetchType="eager"  column="{uid=id}">  </collection>     1. 鉴别器（discriminator）像Java 语言中的 switch 语句。 |

### 动态SQL

使用的是OGNL表达式

|  |
| --- |
| 动态SQL标签有：   * if * choose (when, otherwise) * trim (where, set) * foreach  1. If:判断的是传进来的参数，也可以调用 方法如：name.trim()   <select id="selectUsersByUser" resultType="com.space.bean.User">  select \* from users  Where 1=1  <if test="id !=null">  and id=#{id}  </if>  <if test="name !=null and name.trim()!=''">  and name=#{name}  </if>  </select>  注意：1=1写法，也可以用标签<where>,如果第一个多出and或or会去掉。  <select id="selectUsersByUser" resultType="com.space.bean.User">  select \* from users  <where>  <if test="id !=null">  id=#{id}  </if>  <if test="name !=null and name.trim()!='' ">  and name=#{name}  </if>  </where>  </select>   1. Trim自定义截取规则   <select id="selectUsersByUser" resultType="com.space.bean.User">  select \* from users  <trim prefix="where" prefixOverrides="and | or">  <if test="id !=null">  id=#{id}  </if>  <if test="name !=null and name.trim()!='' ">  and name=#{name}  </if>  </trim>  </select>  Prefix:前缀，prefixOverrides:前缀覆盖，suffix：后缀，suffixOverrides:后缀覆盖   1. Choose：类似 if-else if,是互斥的只能留一个   <select id="selectUsersByUser" resultType="com.space.bean.User">  select \* from users  <trim prefix="where" prefixOverrides="and | or" >  <choose>  <when test="id !=null">  id=#{id}  </when>  <when test="name !=null and name.trim()!='' ">  and name=#{name}  </when>  <otherwise>##其他  1=1  </otherwise>  </choose>  </trim>  </select>   1. Set:动态更新   <update id="updateUserByDynamic" parameterType="com.space.bean.User">  update users  <set>  <if test="name !=null">  name=#{name},  </if>  <if test="age !=null">  age=#{age},  </if>  <if test="addr !=null">  addr\_id=#{addr.id}  </if>  </set>  <where>  id=#{id}  </where>  </update>   1. foreach:遍历   <select id="selectPostIn" resultType="domain.blog.Post">  SELECT \*  FROM POST P  WHERE ID in  <foreach item="item" index="index" collection="list"  open="(" separator="," close=")">  #{item}  </foreach></select>   1. Foreach进行批量保存   <insert id="saveUsers" >  insert into users(name,age,addr\_id)  values  <foreach collection="users" item="user" separator=",">  (#{user.name},#{user.age},#{user.addr.id})  </foreach>  </insert>  Oracle批量保存：两种方式       1. 内置参数：不止有传递的参数，还有内置参数：\_parameter:代表整个参数。   \_databaseId:如果全局变量设置了databaseIdProvider，则\_databaseId代表当前数据库别名。       1. bind:将ognl表达式的值绑定到一个变量中，方便后来引用   如下：like ‘%xx%’的拼接  <select id**="selectUserByName"** resultType**="com.space.bean.User"**>  <bind name**="\_name"** value**="'%'+name+'%'"**/>  select \* from users  where name like #{\_name} </select>   1. <sql> 抽取重用的sql片段   <select id="selectUserByName" resultType="com.space.bean.User">  <bind name="\_name" value="'%'+name+'%'"/>##此处引用  select  <include refid="result\_column"/>  from users  where name like #{\_name}  </select>  ###下边定义  <sql id="result\_column">  id,name,age  </sql>  如下也可以在<sql>中写动态标签 |

### 缓存

|  |
| --- |
| Mybatis默认定义了两级缓存  如：动态菜单数据，经常使用，为了提高效率应缓存 这些数据。   1. 一级缓存：本地缓存   也是sqlSession缓存，一直打开，无法关闭，是在一次session会话的缓存，两个session会话无法公用一级缓存。  如下：两个在同一个session的相同查询，第二次查询 直接使用第一次缓存的数据。  List<User> list = mapper.selectUserByName("a");  List<User> list2 = mapper.selectUserByName("a");  System.out.println(list == list2);//此处打印 为 true  一级缓存失效情况：   * 不同session中查询 * 同一session,查询条件不同。根本原因在一级缓存中没有 * 同一session,执行了增删改操作，则缓存失效，因为有可能数据改变了   List<User> list = mapper.selectUserByName("a");  //执行修改操作  mapper.updateUserByDynamic(new User());  //执行同一查询操作  List<User> list2 = mapper.selectUserByName("a");  System.out.println(list == list2);//打印false,缓存失效需要重新查询   * 同一session,手动清空了缓存，则缓存失效。   List<User> list = mapper.selectUserByName("a");  session.clearCache();  List<User> list2 = mapper.selectUserByName("a");  System.out.println(list == list2);  一级缓存：sqlSession级别的一个map,当前sqlSession的查询的数据都会放入Map,下次同一Sqlsession的查询会先 在map中查找。   1. 二级缓存：全局缓存   一级缓存作用范围只限于同一次会话，其他会话无法使用。有些共同使用的数据，需要在多个查询/业务之间公用。  二级缓存基于 namespace,一个namespace 有一个二级缓存。  工作机制：   * 会话查询的结果保存在一级缓存 * 当会话还关闭，将一级缓存的数据保存到二级缓存，新会话可从二级缓存中查找。 * 同一会话的不同namespace的查询 会缓存到各自namespace对应的缓存中去。   使用二级缓存：   * 全局配置文件中开启全局缓存   <settings>  <setting name**="cacheEnabled"** value**="true"**/> </settings>   * 需要缓存的namespace 映射文件中配置   <cache></cache>         * 因为缓存时默认需要去序列化与反序列化，因此 缓存的类需要 实现序列化接口   public class User implements Serializable{  如下测试：  SqlSession session1 = sqlSessionFactory.openSession();  SqlSession session2 = sqlSessionFactory.openSession();  UserMapper mapper1 = session1.getMapper(UserMapper.class);  UserMapper mapper2 = session2.getMapper(UserMapper.class);  User user1 = mapper1.selectUserById(1);  session1.close();  User user2 = mapper2.selectUserById(1);  session2.close();  日志结果：    **注意：只有在会话关闭或提交之后 一级缓存数据才能缓存到 二级缓存**   1. 缓存的设置/属性  * cacheEnabled设置false,会关闭二级缓存，一级缓存一直可用   <select id="selectUserByName" resultType="com.space.bean.User" useCache="true">   * useCache默认为true,设置false只对二级缓存作用 * 每个增删改标签都有：**flushCache默认true,一级和二级缓存都被清空。**   <insert id="saveUsers" flushCache="true">  **查询标签默认为false**   * **session.clearCache()只清除一级缓存**   **本地缓存（一级缓存）作用域：默认session,设置STATEMENT则禁用一级缓存。**  <setting name**="localCacheScope"** value**="SESSION"**/> |

### 缓存原理

|  |
| --- |
| 新会话进来，先从二级缓存中查找，再去找一级缓存 |

### 第三方缓存ehcache

|  |
| --- |
| Mybatis自己的缓存比较简单，但其留有缓存接口，其他人可以去实现接口，提供自己的缓存。第三方二级缓存  引入jar包依赖：  <dependency>  <groupId>net.sf.ehcache</groupId>  <artifactId>ehcache-core</artifactId>  <version>2.6.8</version> </dependency> <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>1.7.25</version> </dependency> <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>1.7.25</version>  <scope>test</scope> </dependency> <dependency>  <groupId>org.mybatis.caches</groupId>  <artifactId>mybatis-ehcache</artifactId>  <version>1.1.0</version> </dependency>  类路径下创建ehcache.xml配置文件  <?xml version="1.0" encoding="UTF-8"?>  <ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:noNamespaceSchemaLocation="../config/ehcache.xsd">  <!-- 磁盘保存路径 -->  <diskStore path="ehcache\_out" />  <defaultCache  maxElementsInMemory="100"  maxElementsOnDisk="10000000"  eternal="false"  overflowToDisk="true"  timeToIdleSeconds="120"  timeToLiveSeconds="120"  diskExpiryThreadIntervalSeconds="120"  memoryStoreEvictionPolicy="LRU">  </defaultCache>  </ehcache>  映射文件中使用ehcache作为二级缓存，全局文件中打开缓存。  <cache type="org.mybatis.caches.ehcache.EhcacheCache"/>  使用<cache-ref>标签直接 引用 其他 namespace缓存类型如下：  <cache-ref namespace**="com.space.dao.UserMapper"**/> |

### Mybatis逆向工程

|  |
| --- |
| 根据表生成：POJO,mapper接口，映射文件等   * 导入依赖   <dependency>  <groupId>org.mybatis.generator</groupId>  <artifactId>mybatis-generator-core</artifactId>  <version>1.3.5</version> </dependency>   * 新建逆向工程配置文件generatorConfig.xml   <?xml version**="1.0"** encoding**="UTF-8"**?> <!DOCTYPE generatorConfiguration  PUBLIC **"-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"  "http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd"**> <generatorConfiguration>  <context id**="DB2Tables"** targetRuntime**="MyBatis3"**>  <!--连接目标数据库-->  <jdbcConnection driverClass**="com.mysql.jdbc.Driver"** connectionURL**="jdbc:mysql://localhost:3306/mybatis"** userId**="root"** password**="root"**>  </jdbcConnection>  <!--类型解析器-->  <javaTypeResolver >  <property name**="forceBigDecimals"** value**="false"** />  </javaTypeResolver>  <!--指定java bean 生成策略  targetPackage:包  targetProject：目标工程  -->  <javaModelGenerator targetPackage**="com.space.bean"** targetProject**="./src/main/java"**>  <property name**="enableSubPackages"** value**="true"** />  <property name**="trimStrings"** value**="true"** />  </javaModelGenerator>  <!--sql映射文件策略-->  <sqlMapGenerator targetPackage**="."** targetProject**="./src/main/resources"**>  <property name**="enableSubPackages"** value**="true"** />  </sqlMapGenerator>  <!--指定mapper接口-->  <javaClientGenerator type**="XMLMAPPER"** targetPackage**="com.space.dao"** targetProject**="./src/main/java"**>  <property name**="enableSubPackages"** value**="true"** />  </javaClientGenerator>  <!--指定生成的表 和对应java bean类名-->  <table tableName**="items"** domainObjectName**="Items"**></table>   </context> </generatorConfiguration>   * 运行：可以使用最原始的java代码，也可以使用 maven插件   List<String> warnings = new ArrayList<String>();  boolean overwrite = true;  InputStream resourceAsStream = Resources.getResourceAsStream("generatorConfig.xml");  ConfigurationParser cp = new ConfigurationParser(warnings);  Configuration config = cp.parseConfiguration(resourceAsStream);  DefaultShellCallback callback = new DefaultShellCallback(overwrite);  MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config, callback, warnings);  myBatisGenerator.generate(null); |

### 原理

|  |
| --- |
| 运行流程：   1. 获取sqlSessionFactory对象   把所有配置文件解析并保存在 Configuration对象中，返回包含Configuration的DefaultSqlSessionFactory对象  注意：MapperStatement 代表一个增删改查详细信息   1. 获取sqlSession对象   返回一个defaultSqlSession对象，包括以个Executor和Configuration  这一步会创建Executor     1. 获取映射接口的代理对象（MapperProxy）   获取代理对象   1. 执行增删改查方法     总结：   * 根据配置文件（全局和映射文件），初始化Configuration对象 * 创建defaultSqlSession对象，包括以个Executor和Configuration * defaultSqlSession.getMapper()，拿到Mapper接口对象的代理类MapperProxy * 执行增删改查方法 * 调用defaultSqlSession的增删该查 * 创建StatementHandler对象，同时创建 parameterHandler,ResultsSetHandler * 调用StatementHandler预编译参数以及设置参数 * 调用StatementHandler增删该查 * ResultsSetHandler封装结果   四大对象创建都会有一个拦截器的封装 |
|  |

### 插件

|  |
| --- |
| 插件编写：   * 实现Interceptor 接口   public class Myplugin implements Interceptor  //插件签名，哪个对象的哪个方法需要拦截  @Intercepts({  @Signature(type = StatementHandler.class,method = "parameterize",args = Statement.class)  })  public class Myplugin implements Interceptor {  //拦截目标对象的目标方法  public Object intercept(Invocation invocation) throws Throwable {  Object proceed = invocation.proceed();//执行目标方法：放行  return null;  }  //包装目标对象，为目标对象创建代理对象  public Object plugin(Object o) {  //使用当前拦截器包装目标对象  Object wrap = Plugin.wrap(o, this);  return wrap;//返回当前target创建的动态代理  }  //将插件注册时的property属性设置进来  public void setProperties(Properties properties) {  System.out.println("插件配置信息" + properties);  }  }   * 插件签名，哪个对象哪个方法需要拦截   使用注解签名：  @Intercepts({  @Signature(type = StatementHandler.class,method = "parameterize",args = Statement.class)  })   * 注册插件：全局配置文件注册插件   <plugins>  <plugin interceptor**="com.space.dao.Myplugin"**>  <property name**="username"** value**="space"**/>  <property name**="pwd"** value**="121"**/>  </plugin> </plugins>    如上 日志：在执行StatementHandler.parameterize()方法前被拦截  多个插件按照插件配置创建代理对象，会先执行 后边的插件在执行前边的插件，层层代理    如下实例： |

### 分页插件：PageHelper

|  |
| --- |
| 加入依赖：  <dependency>  <groupId>com.github.pagehelper</groupId>  <artifactId>pagehelper</artifactId>  <version>5.1.10</version>  </dependency>  <dependency>  <groupId>com.github.jsqlparser</groupId>  <artifactId>jsqlparser</artifactId>  <version>2.0</version>  </dependency>  全局配置文件 注册插件：  <plugins>  <plugin interceptor**="com.github.pagehelper.PageInterceptor"**>  </plugin> </plugins>  使用：  UserMapper mapper1 = session1.getMapper(UserMapper.class); PageHelper.*startPage*(2,3); List<User> users = mapper1.selectAll();    Doc:https://pagehelper.github.io/ |

## Spring

### IOC

|  |
| --- |
| 控制反转：对于spring框架来说，就是由spring来负责控制对象的生命周期和对象间的关系.  传统软件系统，对象和对象/模块和模块之间耦合度很高，如下图：    如果使用一个第三方 去扮演一个中间人角色，将各个对象粘合起来，如下图：    各个对象之前没有依赖关系，各自不需要知道其他对象的情况。  之前对象A如果需要使用对象B,需要自己去new 或者拿到已创建的对象，控制权都在自己手里。  引入IOC后IOC容器会主动创建并注入B对象到A中，控制权不在自己手里，这就是控制反转。 |

### Spring两种获取对象的方式

|  |
| --- |
| * Id获取 * 运行时获取：通过类名获取，如果配置中配置了多个相同类的对象，spring就会报错，无法分辨出是要获取哪个对象。   如下：  Xml:  <bean class="com.space.bean.Student" id="stu"></bean>  Java:  ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");  Student stu = (Student) context.getBean("stu");//id获取  Student stu2 = context.getBean(Student.class);//运行时获取  System.out.println(stu == stu2);  加载beans.xml获取ApplicationContext对象，在加载配置文件时，创建Student对象（通过反射调用Student类的无参构造方法） |

### 对象属性的赋值

|  |
| --- |
| Xml中设置：  <bean class="com.space.bean.Student" id="stu">  <property name="id" value="1"/>  <property name="name" value="space"/>  <property name="age" value="12"/>  </bean>  Spring 会调用对象的每个属性的setter方法去设置值。  特殊字符可以使用如下：  <property name="name">  <value><![CDATA[<张三>]]></value>  </property>  **P命名空间给属性赋值：**  <bean id="user"  class="com.southwind.entity.User"  p:id="1" p:name="张三"  p:age="23"  p:car-ref="car">  </bean> |

### 对象创建方式

|  |
| --- |
| * 构造器创建（有参数和无参数）   public Student(Integer id, String name, Integer age) {  this.id = id;  this.name = name;  this.age = age;  }  Xml:  <bean class="com.space.bean.Student" id="stu">  <constructor-arg name="id" value="1"/>  <constructor-arg name="name" value="space"/>  <constructor-arg name="age" value="111"/>  </bean>   * 静态工厂方法：有些对象我们无法创建，需使用提供的工厂方法创建   public class StaticCarFactory {//工厂类  private static Map<Integer,Car> cars;  static{  cars = new HashMap<Integer,Car>();  cars.put(1, new Car(1,"奥迪"));  cars.put(2, new Car(2,"奥拓"));  }  public static Car getCar(int num){  return cars.get(num);  }  }  <bean id="car" factory-method="getCar" class="com.space.bean.StaticCarFactory">  <constructor-arg value="2"></constructor-arg>//参数  </bean>   * 实例工厂方法：需要实例工厂类去调用其方法获取对象的情况   public class StaticCarFactory {  private static Map<Integer,Car> cars;  static{  cars = new HashMap<Integer,Car>();  cars.put(1, new Car(1,"奥迪"));  cars.put(2, new Car(2,"奥拓"));  }  public Car getCar(int num){  return cars.get(num);  }  }  Xml:需要实例化工厂对象  <bean id="carFactory" class="com.space.bean.StaticCarFactory"></bean>  <bean id="car" factory-bean="carFactory" factory-method="getCar">  <constructor-arg value="2"></constructor-arg>  </bean> |

### 自定义类/list/属性的注入

|  |
| --- |
| <bean class="com.space.bean.Student" id="stu">  <property name="addr" ref="addr"/>//引用下边需要的对象  </bean>  <bean id="addr" class="com.space.bean.Addr">  <property name="id" value="1"/>  <property name="city" value="西安"/>  </bean>  List:  <bean class="com.space.bean.Student" id="stu">  <property name="addr" ref="addr"/>  <property name="books">  <list>  <ref bean="book1"/>  <ref bean="book2"/>  </list>  </property>  </bean>  <bean id="book1" class="com.space.bean.Book">  <property name="id" value="1"/>  <property name="name" value="语文"/>  </bean>  <bean id="book2" class="com.space.bean.Book">  <property name="id" value="2"/>  <property name="name" value="数学"/>  </bean> |

### Spring中的Bean/bean关系

|  |
| --- |
| Bean的作用域：  bean 是根据 scope 来生成的，表示 bean 的作用域，scope 有 4 种类型：  singleton，单例，表示通过 Spring 容器获取的该对象是唯一的；（**默认**）   * prototype，原型，表示通过 Spring 容器获取的对象都是不同的；（**克隆模式**） * request，请求，表示在一次 HTTP 请求内有效； * session，会话，表示在一个用户会话内有效。   单例则指：定义的bean，每次获取（id获取）都是同一个对象  原型则指：定义的bean，每次获取（id获取）会克隆定义的bean对象返回  Spring中bean的继承：是指继承属性值  <bean id="user" class="com.southwind.entity.User">  <property name="id" value="1"></property>  <property name="name" value="张三"></property>  <property name="age" value="23"></property>  </bean>  <bean id="user2" class="com.southwind.entity.User" parent="user">  //这里可以覆盖父属性值  </bean>  依赖关系：  Spring bean是按照配置顺序 创建对象的，如果A对象依赖B对象，则需要配置依赖关系：这样就会先创建被依赖的对象。  <bean class="com.space.bean.Student" id="stu" depends-on="addr">  </bean>    <bean class="com.space.bean.Addr" id="addr">  </bean> |

### 注入方式

|  |
| --- |
| * 构造器 * Setter   手动：现实的配置出属性如下：  <bean class="com.space.bean.Student" id="stu">  <property name="addr" ref="addr"/>//引用下边需要的对象  </bean>  <bean id="addr" class="com.space.bean.Addr">  <property name="id" value="1"/>  <property name="city" value="西安"/>  </bean>  自动：byName和byType  <bean class="com.space.bean.Student" id="stu" autowire="byName">  </bean>  <bean id="addr" class="com.space.bean.Addr">  <property name="id" value="1"/>  <property name="city" value="西安"/>  </bean>  byName会按照属性名去找对象自动注入  byType会按照类型去注入（如果有多个该类型的对象则报错）   * 接口 |

### Spring获取外部资源文件

|  |
| --- |
| 如下配置数据库连接池：  <context:property-placeholder location="db.properties"></context:property-placeholder>  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="driverClass" value="${driver}"/>  <property name="jdbcUrl" value="${url}"/>  <property name="user" value="${username}"/>  <property name="password" value="${password}"/>  </bean> |

### Spring容器

|  |
| --- |
| Spring提供了两个容器：   * bean工厂（太简单，用的不多） * 应用上下文ApplicationContext   应用上下文有多个实现： |

### 基于注解的自动配置

|  |
| --- |
| 组件扫描配置：  <context:component-scan base-package="com.space"></context:component-scan>   * @Controller：对应Controller的注解 * @Service:对应业务类的注解 * @Repository:对应Dao持久层注解 * @Component:bean注解 * @Autowired:表示自动装载默认：byType，默认id:(类名首数字母小写)   @AutoWired可以用在类的任意方法上（构造器，setter方法，其他方法）如下：  @Autowired  @Qualifier("myUserService")  public void setUserService(UserService userService){  this.userService = userService;  }  required = false：非必须匹配，没匹配到不会报异常，赋值 null  @Autowired(required = false)  @Qualifier("userService")  public void setUserService(UserService userService){  this.userService = userService;  }   * @Qualifier：结合@Autowired使用可以按照byName注入 如下：   @Autowired  @Qualifier("myUserService")//指定id  private UserService userService;  Service注册时需指定id:  @Service("myUserService")  Controller:  @Controller  public class UserController {  @Autowired  private UserService userService;  public User getUserById(Integer id){  return userService.getUserById(id);  }  }  Service:  @Service  public class UserServiceImpl implements UserService{  @Autowired  private UserDao userDao;  public User getUserById(Integer id) {  return userDao.getUserById(id);  }  }  Dao:  @Repository  public class UserDaoImpl implements UserDao { |

### AOP

|  |
| --- |
| 面向切面编程：使用动态代理实现    应用：   * 日志打印 * 事务控制 * 安全 * 性能监控   AOP的使用：  导入依赖：  <dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjrt</artifactId>  <version>1.9.1</version> </dependency> <dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjweaver</artifactId>  <version>1.9.1</version> </dependency>  设置自动扫描和AOP自动生成代理配置  <context:component-scan base-package**="com.space"**></context:component-scan> <aop:aspectj-autoproxy ></aop:aspectj-autoproxy>  编写切面对象类：  @Aspect  @Component  public class LoggerAspect {  @Before("execution(public int com.space.bean.CalImpl.\*(..))")  public void before(JoinPoint joinPoint){  //获取方法名  String name = joinPoint.getSignature().getName();  //获取参数列表  String args = Arrays.toString(joinPoint.getArgs());  System.out.println(name+"的参数是:"+args);  }  @After("execution(public int com.space.bean.CalImpl.\*(..))")  public void after(JoinPoint joinPoint){  //获取方法名  String name = joinPoint.getSignature().getName();  System.out.println(name+"方法结束");  }  @AfterReturning(value = "execution(public int com.space.bean.CalImpl.\*(..))",returning = "result")  public void afterReturn(JoinPoint joinPoint,Object result){  //获取方法名  String name = joinPoint.getSignature().getName();  System.out.println(name+"方法的结果是"+result);  }  @AfterThrowing(value = "execution(public int com.space.bean.CalImpl.\*(..))",throwing = "ex")  public void afterThrowing(JoinPoint joinPoint,Exception ex){  //获取方法名  String name = joinPoint.getSignature().getName();  System.out.println(name+"方法抛出异常："+ex);  }  }  切点类：  @Component  public class CalImpl implements Cal{  public int add(int num1, int num2) {  // TODO Auto-generated method stub  return num1+num2;  }  public int sub(int num1, int num2) {  // TODO Auto-generated method stub  return num1-num2;  }  public int mul(int num1, int num2) {  // TODO Auto-generated method stub  return num1\*num2;  }  public int div(int num1, int num2) {  // TODO Auto-generated method stub  return num1/num2;  }  }  执行：  ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");  Cal cal = (Cal) context.getBean("calImpl");  cal.add(1,2);  cal.div(4,2);  cal.mul(5,1);  cal.sub(2,3);  **Before** 在方法被调用之前调用通知 **After** 在方法完成之后调用通知，无论方法执行是否成功 **After-returning** 在方法成功执行之后调用通知 **After-throwing** 在方法抛出异常后调用通知 |

## Spring MVC

### 核心组件

|  |
| --- |
|  |

### 工作流程

|  |
| --- |
|  |

### 使用

|  |
| --- |
| * 使用idea maven web骨架创建项目 * 添加依赖   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>4.3.1.RELEASE</version>  </dependency>   * 配置web.xml 前端控制器，将所有请求交给 该servlet处理   <servlet>  <servlet-name>springmvc</servlet-name>  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <!-- 指定springmvc.xml的路径 -->  <param-value>classpath:springmvc-servlet.xml</param-value>  </init-param>  </servlet>  <servlet-mapping>  <servlet-name>springmvc</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping>   * 创建视图解析器文件并配置：springmvc-servlet.xml   <!--自动注解扫描-->  <context:component-scan base-package="com.space"></context:component-scan>  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <!-- 前缀 -->  <property name="prefix" value="/"></property>  <!-- 后缀 -->  <property name="suffix" value=".jsp"></property>  </bean>  如:前缀 prefix :/WEB\_INF/ 加 + 逻辑视图名称 + 后缀 suffix:.jsp 就是返回的view     * 创建Handler类   @Controller  public class HelloHandler {  @RequestMapping("/index")  public String index(){  System.out.println("index...");  return "index";  }  }  （1）DispatcherServlet 接收到 URL 请求 index，结合 @RequestMapping("/index") 注解将该请求交给 index 业务方法。  （2）执行 index 业务方法，控制打印日志，并返回 "index" 字符串。  （3）结合 springmvc.xml 中的视图解析器配置，找到目标资源：/index.jsp，即根目录的 index.jsp，将该 JSP 资源返回客户端完成响应。 |

### 常用注解

|  |
| --- |
| * **@RequestMapping**:将url请求和业务方法映射，在控制器类和方法处均可使用，如下：   @Controller  @RequestMapping("/helloHandler")  public class HelloHandler {  @RequestMapping(value="hello")  public String hello(){  System.out.println("hello world");  return "index";  }  }  常用的参数：  value(默认参数)  Method:请求的类型（Get,Post,Put,Delete）  @RequestMapping(value = "index",method = RequestMethod.GET)  public String index(){  System.out.println("index...");  return "index";  }  Pararms:需要携带的参数和参数满足的条件，参数需同时满足。  @RequestMapping(value = "index",params = {"name","id=10"})   * **@RequestParam参数绑定**   public String index(@RequestParam("name") String name,@RequestParam("id") int id){  System.out.println("index...");  return "index";  }  进行参数映射绑定，和类型转换（HandlerAdapter完成）   * **RESTful风格参数获取@PathVariable**   @RequestMapping(value = "rest/{name}")  public String index(@PathVariable("name") String name){  System.out.println("index...");  return "index";  }   * **获取cookie值@CookieValue**   @RequestMapping(value = "/cookie")  public String index(@CookieValue(value = "JSESSIONID") String sessionId){  System.out.println("index...");  return "index";  }   * **参数和对象的绑定**   @RequestMapping("/addUser")  public String getPOJO(**User user**){  System.out.println(user);  return "index";  }  只要前台表单中的name和对象中属性名称相符合就可以自动绑定   * **@ResponseBody:直接返回字符串到客户端，不经过视图解析器，如（json数据）**   将业务方法返回的对象，通过 HttpMessageConverter 接口转为指定格式的数据，JSON、XML等，响应给客户端。  @RequestMapping("/json")  @ResponseBody  public String returnJson(){  return "test json data";  }   * **@RequestBody**:   读取 HTTP 请求参数，通过 Spring MVC 提供的 HttpMessageConverter 接口将读取的参数转为 JSON、XML 格式的数据，绑定到业务方法的形参。 |
|  |

### 乱码解决

|  |
| --- |
| 在web.xml中添加过滤器：  <filter>  <filter-name>encodingFilter</filter-name>  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>  <init-param>  <param-name>encoding</param-name>  <param-value>UTF-8</param-value>  </init-param>  </filter>  <filter-mapping>  <filter-name>encodingFilter</filter-name>  <url-pattern>/\*</url-pattern>  </filter-mapping>  处理 @ResponseBody 中文乱码：在 springmvc.xml 中配置消息转换器。  <mvc:annotation-driven >  <!-- 消息转换器 -->  <mvc:message-converters register-defaults="true">  <bean class="org.springframework.http.converter.StringHttpMessageConverter">  <property name="supportedMediaTypes" value="text/html;charset=UTF-8"/>  </bean>  </mvc:message-converters>  </mvc:annotation-driven> |

### 转发和重定向

|  |
| --- |
| * “redirect：/index”   会将逻辑视图当成url处理，除非显示指出物理视图如：  "redirect:/index.jsp"   * “forward:/hello”   当是重定向和转发时：均会将逻辑视图当url处理，除非显示指出物理视图。 |

### 请求数据绑定

数据的绑定由：HandlerAdapter完成

|  |
| --- |
| * **基本类型**   @RequestMapping("/json")  @ResponseBody  public String returnJson(**int a**){  return "test json data";  }  Handler方法如果带参数，会和请求过来中的参数进行匹配设置，如果请求没有对应形惨名的参数则报错，有则进行类型转换，无法转换则报错。之后执行handler方法。   * **参数设置**   @RequestMapping("/json")  @ResponseBody  public String returnJson(@RequestParam(value = "a",required = true,defaultValue = "1") int a){  return "test json data:" + a;  }  required=true：则参数必须有，DefaultValue:默认值，有参数则覆盖默认值。   * **数组**   http://localhost:8080/json?array=1&array=1  @RequestMapping("/json")  @ResponseBody  public String returnJson(String[] array){  return "test json data:" + array;  }   * **POJO对象**   @RequestMapping("/user")  @ResponseBody  public String returnJson(**User user**){  return "注册信息User:" + user;  }   * **POJO级联关系：使用属性.xxxx 来封装**   地址编号：<input type="text" name="**address.id**"/><br/>  地址：<input type="text" name="address.name"/><br/>   * **List:Spring mVC不支持直接转换List类型，需要封装**   <form action="listType" method="post">  用户1姓名：<input type="text" name="users[0].name"/><br/>  用户1年龄：<input type="text" name="users[0].age"/><br/>  用户2姓名：<input type="text" name="users[1].name"/><br/>  用户2年龄：<input type="text" name="users[1].age"/><br/>  用户3姓名：<input type="text" name="users[2].name"/><br/>  用户3年龄：<input type="text" name="users[2].age"/><br/>  <input type="submit" value="提交"/>  </form>  **先封装：**  public class UserList {  private List<User> users;  }  @RequestMapping(value="/listType")  @ResponseBody  public String listType(UserList userList){  }   * **set**   **和 List 一样，需要封装自定义包装类，将 Set 集合作为属性。不同的是，使用 Set 集合，需要在包装类构造函数中，为 Set 添加初始化对象。**  public class UserSet {  private Set<User> users = new HashSet<User>();  public UserSet(){  users.add(new User());  users.add(new User());  users.add(new User());  }  }  @RequestMapping(value="/setType")  @ResponseBody  public String setType(UserSet userSet){  StringBuffer sbf = new StringBuffer();  for(User user:userSet.getUsers()){  sbf.append(user);  }  return "用户："+sbf.toString();  }  页面**：**  <form action="setType" method="post">  用户1姓名：<input type="text" name="users[0].name"/><br/>  用户1年龄：<input type="text" name="users[0].age"/><br/>  用户2姓名：<input type="text" name="users[1].name"/><br/>  用户2年龄：<input type="text" name="users[1].age"/><br/>  用户3姓名：<input type="text" name="users[2].name"/><br/>  用户3年龄：<input type="text" name="users[2].age"/><br/>  <input type="submit" value="提交"/>  </form>   * **Map:需要自定义包装类**   public class UserMap {  private Map<String,User> **users**;  }  **Handler:**  @RequestMapping(value="/mapType")  @ResponseBody  public String mapType(UserMap userMap){  StringBuffer sbf = new StringBuffer();  for(String key:userMap.getUsers().keySet()){  User user = userMap.getUsers().get(key);  sbf.append(user);  }  return "用户："+sbf.toString();  }  页面：users['a'].age中为key  <form action="mapType" method="post">  用户1姓名：<input type="text" name="users['a'].name"/><br/>  用户1年龄：<input type="text" name="users['a'].age"/><br/>  用户2姓名：<input type="text" name="users['b'].name"/><br/>  用户2年龄：<input type="text" name="users['b'].age"/><br/>  用户3姓名：<input type="text" name="users['c'].name"/><br/>  用户3年龄：<input type="text" name="users['c'].age"/><br/>  <input type="submit" value="提交"/>  </form>   * **JSON**   <script type="text/javascript">  var user = {  "name":"张三",  "age":22  };  $.ajax({  url:"jsonType",  data:JSON.stringify(user),###此处需转换  type:"post",  contentType: "application/json;charse=UTF-8",####不能省略  dataType:"text",  success:function(data){  var obj = eval("(" + data + ")");  alert(obj.name+"---"+obj.age);  }  })  </script>  @RequestMapping(value="/jsonType")  @ResponseBody  public User jsonType(@RequestBody User user){  //修改年龄  user.setAge(user.getAge()+10);  //返回前端  return user;  }  **@RequestBody:**  读取 HTTP 请求参数，通过 Spring MVC 提供的 HttpMessageConverter 接口将读取的参数转为 JSON、XML 格式的数据，绑定到业务方法的形参。 |

### 原理

|  |
| --- |
|  |

### FastJson的简单使用

|  |
| --- |
| 添加依赖：  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>1.2.58</version>  </dependency>  配置消息转换器:  <mvc:annotation-driven >  <!-- 消息转换器 -->  <mvc:message-converters register-defaults="true">  <bean class="org.springframework.http.converter.StringHttpMessageConverter">  <property name="supportedMediaTypes" value="text/html;charset=UTF-8"/>  </bean>  <bean class="com.alibaba.fastjson.support.spring.FastJsonHttpMessageConverter"/>  </mvc:message-converters>  </mvc:annotation-driven>  @RequestMapping("/user")  @ResponseBody  public User returnJson( User user){  System.out.println(user);  return user;  }  文档：  https://github.com/alibaba/fastjson/wiki/%E5%9C%A8-Spring-%E4%B8%AD%E9%9B%86%E6%88%90-Fastjson |

### 业务数据（返回数据）绑定

|  |
| --- |
| 业务数据一般存储在JSP的四大域中：pageContext,request,session,application  Spring mvc 提供了如下几种添加业务数据的方式：   * Map * Model * ModelAndView * @SessionAttributes * @ModelAttribute   绑定到request中：  **Map**方法：业务方法传入Map类型的参数，给map中添加值即可    如下：  **@RequestMap**Map ma**ping**(**"/user"**) public String returnJson(User user,p){  map.put(**"user"**,user);  return **"index"**; }  Jsp:  ${user.name}  **Model:与map类似**  @RequestMapping("/user2")  public String returnJson(User user, **Model model**){  model.addAttribute("user",user);  return "index";  }  **ModelAndView:与Model,Map不同它包含了业务数据和视图信息，所以方法返回必须为ModelAndView类型。**  @RequestMapping("/user3")  public ModelAndView returnJson(User user){  ModelAndView modelAndView = new ModelAndView();  modelAndView.addObject(“user”，user);  modelAndView.setViewName("index");  return modelAndView;  }  如上设置视图的方式有多种：            等等....  **@ModelAttribute:注解绑定数据，它会在Springmvc调用任何一个handler方法之前执行，并设置数据到request(默认)中，业务方法中重新设置会覆盖该设置。**  **@ModelAttribute** public User getUserData(){  User u = new User();  u.setId(11);  u.setName(**"space6666666"**);  return u; }  会设置如：返回值类型首字母小写的为Key,返回值为value,  如果没有返回值，则需在方法带如Map/Model参数在方法中设置，如下：    **绑定到session中：**@SessionAttributes类注解设置，@SessionAttribute方法注解取值。  使用注解：@SessionAttributes(value = “user”):将Model中属性名为user的数据放入session中。如下同时request域中也会有user的对象。（request不影响）  @Controller  @SessionAttributes(value = "user")  public class HelloHandler {  }  @RequestMapping("/user")  public String returnJson2(User user,Map map){  map.put("user",user);  return "index";  }  如下放入多个对象：  @SessionAttributes(value = {"user","url"})  使用@SessionAttribute注解获取值：  @RequestMapping("/user")  public String returnJson2(User user,Map map,@SessionAttribute String url){  map.put("user",user);  System.out.println(url);  return "index";  } |

### 获取HttpServletRequest

|  |
| --- |
| 只需要在业务方法中加入 HttpServletRequest 参数就可直接使用如下：  @RequestMapping("/user4")  public ModelAndView returnJson5(User user, HttpServletRequest request){  ModelAndView modelAndView = new ModelAndView();  modelAndView.addObject("user",user);  modelAndView.addObject("url",request.getRequestURI());  modelAndView.setViewName("index");  return modelAndView;  }  HttpSession也一样如上获取：  @RequestMapping("/user5")  public String returnJson5(User user, HttpSession session){  System.out.println(((User)session.getAttribute("user")).getName());  return "index";  } |

### 自定义数据类型转换器

|  |
| --- |
| 从页面传过来的参数均为String类型，如果handler方法参数类型为int,框架会去转换（如果能转换成功。）其他类型框架无法转换如：Date,这就需要我们自定义一个转换器。  实现Converter接口：  public class DateConverter implements Converter<String,Date> {  private String pattern;  //构造方法设置格式化的参数如：yyyy-MM-dd  public DateConverter(String pattern) {  this.pattern = pattern;  }  @Override  public Date convert(String s) {  SimpleDateFormat sdf = new SimpleDateFormat(this.pattern);  try {  return sdf.parse(s);  } catch (ParseException e) {  e.printStackTrace();  }  return null;  }  }  Xml中设置：  <bean id="conversionService" class="org.springframework.context.support.ConversionServiceFactoryBean">  <property name="converters">  <set>  <bean class="com.space.util.DateConverter">  <!-- 调用有参构造函数创建 bean -->  <constructor-arg type="java.lang.String" value="yyyy-MM-dd"/>  </bean>  </set>  </property>  </bean>  <mvc:annotation-driven conversion-service="conversionService"/>//有该标签在该标签上设置  使用：User有date类型的属性  访问：http://localhost:8080/user6?date=2019-11-12  **@RequestMapping**(**"/user6"**) public String dateTest(User user){  System.*out*.println(user.getDate().toString());  return **"index"**; }  其他自定义类型： |

### RESTful

|  |
| --- |
| 传统web开发中，form表单只支持 post 和 get方式。通过添加  HiddenHttpMethodFilter 过滤器来支持 put和delete方式。  原理如下：  检测请求参数中是否包含\_method这个参数，如果包含则获取其值，判断是哪种方式后进行转换，然后继续传递。具体如下：   * 在form表单中添加隐藏参数\_method 值为：PUT/DELETE * Web.xml中配置 HiddenHttpMethodFilter * 使用   <form action="/httpPut" method="post">  name<input type="text" name="name"/>  <input type="hidden" name="\_method" value="PUT"/>  <input type="submit" value="修改"/>  </form>  <form action="/httpDelete" method="post">  name<input type="text" name="name"/>  <input type="hidden" name="\_method" value="DELETE"/>  <input type="submit" value="删除"/>  </form>  <filter>  <filter-name>HiddenHttpMethodFilter</filter-name>  <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class> </filter>  <filter-mapping>  <filter-name>HiddenHttpMethodFilter</filter-name>  <url-pattern>/\*</url-pattern> </filter-mapping>  @PutMapping("/httpPut")  public String httpPut(@RequestParam("name") String name,Map map){  map.put("name",name);  return "hello";  }  @DeleteMapping("/httpDelete")  public String httpDelete(@RequestParam("name")String name,Map map){  map.put("name",name);  return "hello";  }  如下获取restful形式的路径参数：  @DeleteMapping(value = "/delete/{id}")  public String delete(@PathVariable(value = "id") int id){  courseDAO.deleteById(id);  return "redirect:/getAll";  } |

### 文件上传/下载

|  |
| --- |
| Springmvc使用 fileupload组件。  **加入依赖：**  <dependency>  <groupId>commons-io</groupId>  <artifactId>commons-io</artifactId>  <version>1.3.2</version>  </dependency>  <dependency>  <groupId>commons-fileupload</groupId>  <artifactId>commons-fileupload</artifactId>  <version>1.2.1</version>  **JSP 页面做如下处理：**  input 的 type 设置为 file  form 表单的 method 设置为 post（get 请求只会将文件名传给后台）  form 表单的 enctype 设置为 multipart/form-data，以二进制的形式传输数据  <form action="/upload" method="post" enctype="multipart/form-data">  <input type="file" name="img"/>  <input type="submit" value="上传"/>  </form>  <c:if test="${filepath !=null}">  <img src="/${filepath}">  </c:if>  **Xml重配置bean:**  <bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">  <!-- 处理文件名中文乱码 -->  <property name="defaultEncoding" value="utf-8"/>  <!-- 设置多文件上传，总大小上限，不设置默认没有限制，单位为字节，1M=1\*1024\*1024 -->  <property name="maxUploadSize" value="1048576"/>  <!-- 设置每个上传文件的大小上限 -->  <property name="maxUploadSizePerFile" value="1048576"/>  </bean>  <!-- 设置异常解析器 -->  <bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">  <property name="defaultErrorView" value="/error.jsp"/>  </bean>  **Handler方法：**  @RequestMapping("/upload")  public String fileUpload(@RequestParam(value = "img")MultipartFile img,HttpServletRequest request) throws IOException {  if(img.getSize() > 0){  String path = request.getSession().getServletContext().getRealPath("file");  String filename = img.getOriginalFilename();  File file = new File(path,filename);  img.transferTo(file);//上传文件  request.setAttribute("filepath","file/" + filename);  }  return "index";  }  **多文件上传：Jsp传入数组file,handler参数绑定数组**  **（1）JSP**  <form action="uploads" method="post" enctype="multipart/form-data">  file1:<input type="file" name="imgs"><br />  file2:<input type="file" name="imgs"><br />  file3:<input type="file" name="imgs"><br />  <input type="submit" name="提交">  </form>  <c:if test="${filePaths!=null }">  <h1>上传的图片</h1><br />  <c:forEach items="${filePaths }" var="filePath">  <img width="300px" src="<%=basePath %>${filePath}"/>  </c:forEach>  </c:if>  （2）业务方法，使用 MultipartFile 数组对象接收上传的多个文件  @RequestMapping(value="/uploads", method = RequestMethod.POST)  public String uploads(@RequestParam MultipartFile[] imgs, HttpServletRequest request)  throws Exception {  //创建集合，保存上传后的文件路径  List<String> filePaths = new ArrayList<String>();  for (MultipartFile img : imgs) {  if (img.getSize() > 0) {  String path = request.getSession().getServletContext().getRealPath("file");  String fileName = img.getOriginalFilename();  File file = new File(path, fileName);  filePaths.add("file/"+fileName);  img.transferTo(file);  }  }  request.setAttribute("filePaths", filePaths);  return "uploads";  }  **文件下载：依赖配置等都和上传一样，如下：**  <a href="/download?filename=**${**filename**}**">下载</a>  @RequestMapping("/download")  public void download(@RequestParam(value = "filename")String fileName,HttpServletRequest request,  HttpServletResponse response){  if(fileName != null){  String path = request.getSession().getServletContext().getRealPath("file");  File file = new File(path,fileName);  OutputStream out = null;  if(file.exists()){  try {  //设置下载完毕不打开文件  response.setContentType("application/force-download");  //设置文件名  response.setHeader("Content-Disposition", "attachment;filename="+fileName);  out = response.getOutputStream();  out.write(FileUtils.readFileToByteArray(file));  out.flush();  } catch (IOException e) {  e.printStackTrace();  }finally {  if(out != null){  try {  out.close();  } catch (IOException e) {  e.printStackTrace();  }  }  }  }  }  } |

### 数据效验

|  |
| --- |
| 数据效验有两种方式：   * 基于validator接口进行效验   自定义效验器：实现validator接口  public class UserValidator implements Validator {  @Override  public boolean supports(Class<?> aClass) {  return User.class.equals(aClass);  }  @Override  public void validate(Object o, Errors errors) {  ValidationUtils.rejectIfEmpty(errors, "name", null, "姓名不能为空");  }  }  Xml重配置：  <!-- 基于 Validator 的配置 -->  <mvc:annotation-driven validator="studentValidator"/>  <bean id="studentValidator" class="com.southwind.validator.StudentValidator"/>  使用：两个参数必须挨着写。  @ResponseBody  @PostMapping("/user/save")  public String saveUser(@Validated User user, BindingResult bindingResult){  if(bindingResult.hasErrors()){  return "error";  }  return "success";  }   * 使用Annotation JSR-303标准进行效验   导入依赖：  <!-- JRS-303 -->  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-validator</artifactId>  <version>5.1.3.Final</version>  </dependency>  <dependency>  <groupId>javax.validation</groupId>  <artifactId>validation-api</artifactId>  <version>1.1.0.Final</version>  </dependency>  <dependency>  <groupId>org.jboss.logging</groupId>  <artifactId>jboss-logging</artifactId>  <version>3.1.1.GA</version>  </dependency>  注解需要验证的bean    @NotEmpty(message = "姓名不能为空")  private String name;  Xml配置：  <mvc:annotation-driven />  使用：  @ResponseBody  @PostMapping("/user/save2")  public String saveUser2(@Valid User user, BindingResult bindingResult){  if(bindingResult.hasErrors()){  return bindingResult.getFieldError().getDefaultMessage();  }  return "success";  } |

## Mysql

### 建表

|  |
| --- |
| create table A\_XSJBXXB(  xh int primary key auto\_increment,##主键自增  xm varchar(20),  sr Date,  jtdz varchar(100),  bjdm varchar(30),  lxdh varchar(50),  ssbh varchar(50)  );  修改表明：  rename table 旧表名 to 新表名; |

### 修改表

|  |
| --- |
| 添加字段：  alter table 表名 add 字段名 字段类型（长度）  //批量增加字段  　　alter table 表名 add (字段名1 字段类型（长度）,字段名2 字段类型（长度）,...) |

### 创建存储过程

|  |
| --- |
| if (exists (select \* from sys.objects where name = 'usp\_searchstudents'))  drop proc usp\_searchstudents  go  create proc usp\_searchstudents(@name varchar)  as  select x.xm,x.lxdh,b.bjmc,b.bzr,b.ssxb  from A\_XSJBXXB x,A\_BJXXK b  where x.bjdm = b.bjdm  and x.xm=@name; |

### 多行插入

|  |
| --- |
| insert into A\_XSJBXXB(xm,sr,jtdz,bjdm,lxdh)  values('A','1995-10-1','西安-土门1','3061','029-11111111'),  ('B','1995-10-2','西安-土门2','3061','029-11111112'),  ('C','1995-10-3','西安-土门3','3062','029-11111113'),  ('D','1995-10-4','西安-土门4','3062','029-11111114'); |

## 自己实现IOC

|  |
| --- |
| 依赖：  <dependency>  <groupId>dom4j</groupId>  <artifactId>dom4j</artifactId>  <version>1.6.1</version> </dependency>  public interface ApplicationContenxt {  public Object getBean(String name); }  public class ClassPathXmlApplicationContext implements ApplicationContenxt {   private Map<String,Object> iocContainer = new HashMap<String,Object>();//容器   public ClassPathXmlApplicationContext(String xmlPath) {  SAXReader reader = new SAXReader();  InputStream resourceAsStream = ClassPathXmlApplicationContext.class.getClassLoader().getResourceAsStream(**"bean.xml"**);  try {  Document document = reader.read(resourceAsStream);  Element rootElement = document.getRootElement();  Iterator<Element> iterator = rootElement.elementIterator(**"bean"**);  while (iterator.hasNext()){  Element bean = iterator.next();  String id = bean.attributeValue(**"id"**);  String className = bean.attributeValue(**"class"**);  //反射创建对象  Class<?> beanClass = Class.*forName*(className);  Constructor<?> constructor = beanClass.getConstructor();  Object newInstance = constructor.newInstance();   Iterator<Element> property = bean.elementIterator(**"property"**);  while (property.hasNext()){  Element property\_Element = property.next();  String fieldName = property\_Element.attributeValue(**"name"**);  String fieldValue = property\_Element.attributeValue(**"value"**);   String methodName = **"set"** + fieldName.toUpperCase().substring(0,1) + fieldName.substring(1);  Field field = beanClass.getDeclaredField(fieldName);  Method method = beanClass.getDeclaredMethod(methodName,field.getType());  Object value = null;  String fieldType = field.getType().getName();  if(**"long"**.equals(fieldType)){  value = Long.*parseLong*(fieldValue);  }  if(**"int"**.equals(fieldType)){  value = Integer.*parseInt*(fieldValue);  }  if(**"java.lang.String"**.equals(fieldType)){  value = fieldValue;  }  //反射执行方法  method.invoke(newInstance,value);   }  //将对象存入容器中  iocContainer.put(id,newInstance);  }   } catch (Exception e) {  e.printStackTrace();  }    }   public Object getBean(String name) {  return iocContainer.get(name);  } } |

## 自己实现Spring mvc

|  |
| --- |
| 如下：自定义注解：  @Target(ElementType.TYPE)  @Retention(RetentionPolicy.RUNTIME)  public @interface MyController {  String value() default "";  }  代码如github：git@github.com:anycs/mySpringMvc.git |

## Maven

### 依赖管理

|  |
| --- |
| 同一仓库管理依赖jar包，实现了高复用性。  Jar包之间的依赖maven自动维护。以坐标形式定位依赖文件。 |

### 一键构建

|  |
| --- |
| 清理---编译---测试----报告--------打包----------------部署 ：一键完成。 |

### 安装

|  |
| --- |
| 下载maven,解压，配置环境变量  export M2=/opt/apache-maven-3.5.0  export PATH=${M2}/bin:$PATH  版本：  mvn -v |

### Maven仓库

|  |
| --- |
|  |

### Maven项目目录结构

|  |
| --- |
| 规范化了项目目录结构。 |

### 常用命令

|  |
| --- |
| 清理项目：  mvn clean  编译项目：  mvn compile  执行测试(还编译了源代码)：  mvn test  mvn package ##编译并执行测试 并打包  mvn install ##安装jar包到本地仓库 |
|  |

### Maven生命周期

|  |
| --- |
|  |

### 项目对象模型（POM）

|  |
| --- |
| 项目对象模型（pom文件）主要包括：   * 项目自身信息 * 项目运行所依赖的jar包信息 * 项目运行环境信息（如：tomcat,jdk） |

### Idea 配置maven

|  |
| --- |
| 设置 运行参数，-DarchetypeCatalog=internal 断网情况也可创建 |

### 使用骨架创建maven项目

|  |
| --- |
| * 使用骨架创建java maven项目：   maven-archetype-quickstart    使用骨架不会创建 resources资源文件目录，需要自己手动创建并设置为资源文件类型。  文件夹右击----》make directory as ---->  (推荐不适用骨架 创建 java工程)   * 使用骨架创建 web项目：     选择 maven-archetype-webapp 骨架 |

### Maven依赖作用于

|  |
| --- |
| <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.11</version>  <scope>provided</scope>  </dependency>  Provided:jar包在运行的容器中已经支持，只影响编译和测试阶段  Compile:默认设置，在编译，测试，运行阶段都起作用。  Test:只在测试阶段起作用。 |