

# JAVA

---

## 算法(算法图解/漫画算法)

---

### 数据结构

- 数组
  - `byte[][] b2 = new byte[1][1];`  
`byte b22[][] = new byte[2][2];`  
`byte[] b3 = { (byte)1,(byte)2,3 };`  
`byte[] b32 = new byte[]{ (byte)1,(byte)2,3 };`
- 链表
  - 单向链表: 单节点 节点中保存: 值, 下一个节点地址
    - HashMap(链表深度<8时)
  - 双向链表: 头节点, 尾节点  
节点中保存: 上一个节点地址,值,下一个节点地址;  
双向循环:头结点的before节点为尾节点地址,  
尾节点的after节点为头节点  
非循环: 头结点的before节点为NULL,  
尾节点的after节点为NULL
    - LinkedHashMap
  - 获取链表长度(不含头节点): 遍历
  - 获取链表某位置节点值: 1. 遍历获取总节点数; 2. 遍历到size - index 位置节点
- 栈
  - 支持Push(入栈)和Pop(出站)操作, 维护一个栈顶索引,栈大小,栈是否空,是否满。  
Push :往栈顶插入元素  
Pop: 弹出栈顶元素
  - 栈分类
    - 顺序栈(数组实现)
    - 链式栈(链表实现,单向链表,链表尾为栈顶)
  - 使用场景
    - 表达式计算
      - 表达式 $1+(4-6*3)/2=2$ 
        - 表达式表示为后缀表达式 $1463*-2/+$ 。
        - 使用栈来计算: 碰见数字就入栈, 碰见符号先运算
    - 递归(  
斐波那契数列( $f(1)=f(2)=1;f(n)=f(n-1)+f(n-2),n \geq 3;$ ),  
阶乘( $0!=1, f(n)=n*f(n-1)$ ))
      - 递归本身就是入栈

- 括号匹配(匹配输入的括号是否全匹配)
  - 输入一个字符串 里面只含有 [, ], (, ) 四种括号 ; 现要求判断这个字符串是否满足括号匹配 。如 ([ ]) 是匹配的 ([ ]) 是不匹配的
  - 栈空时, 将字符Push栈中, 并判断, 当当前字符和栈顶字符匹配时, 出栈; 循环结束后, 若栈为空则都匹配, 反之不匹配
- #2个栈实现消息队列
  - 栈特点: 先进后出(FILO)
  - 队列特定: 先进先出(FIFO)
  - 实现: 栈1(S1), 栈2(S2),  
生产数据时, 将数据Push到S1,  
消费数据时, 从S2中Pop数据, S2数据为空时, 将S1数据全部Pop到S2中,  
然后再从S2Pop数据  
(有部分人在S2 Pop数据后, 把剩余数据Pop回S1)
- java.util.Stack extends Vector
  - 数组实现, 默认大小10  
扩容 2倍大小+1
- 队列
- 哈希表

## 排序算法(10大经典排序算法)

- 冒泡排序
  - 概述: 重复走访过要排序的元素列, 依次比较两个相邻的元素, 如果顺序错误, 就把他们交换过来, 直到没有相邻的元素需要交换
  - 提供一个待排序数据, 依次比较相邻2个数的大小, 按从小到大排序  
双层循环: 外层循环次数: arr.length - 1  
内层循环次数: arr.length - i - 1
    - 在第一趟比较完成后, 最后一个数一定是数组中最大的一个数, 所以在比较第二趟的时候, 最后一个数是不参加比较的。
  - 在第二趟比较完成后, 倒数第二个数也一定是数组中倒数第二数, 所以在第三趟的比较中, 最后两个数是不参与比较的。
- 依次类推, 每一趟比较次数减少依次
  - 时间复杂度:  $O(n^2)$  稳定性: 稳定
  - 优化: 设置是否完成标记位, 若一趟中没有任何更新, 则表示数组已排序完成, 结束后续的排序
  - ```
int[] arr = new int[10];
boolean isFinished = false;
for (int i = 0; i < arr.length - 1; i++) {
    for (int j = 0; j < arr.length - i - 1; j++) {
        if (arr[j] > arr[j + 1]) {
            temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
        }
    }
}
```

```

        isFinished = true;
    }
}
if (!isFinished) {
    break;
}
isFinished = false;
}

```

## ■ 选择排序

### ■ 概述(原理):

第一次从待排序的数据元素中选出最小(或最大)的一个元素，存放到队列的起始位置；

然后再从剩余的未排序的元素中选出最小(或最大)的一个元素，然后放到已排序队列尾。

以此类推，直到待排序元素个数为0。

(简单描述:待排序数组的每趟循环中，选出最小(或最大)值放到已排序队列尾(第一趟循环时，放到队列的其实位置)，知道所有循环完成)

### ■ 时间复杂度: $O(n^2)$ 不稳定排序算法

```

public void sort(int[] arr) {
    for (int i = 0; i < arr.length-1; i++) {
        int minIdx = i;
        for (int j = i+1; j < arr.length; j++) {
            if (arr[minIdx] > arr[j]) {
                minIdx = j;
            }
        }
        if (i != minIdx) {
            int temp = arr[i];
            arr[i] = arr[minIdx];
            arr[minIdx] = temp;
        }
    }
}

```

## ■ 插入排序

### ■ 概述(原理): 通过构建有序队列，对于未排序数据，在已排序序列中向后往前扫描，找到相应的位置并插入。

### ■ 步骤:

■ 将待排序序列的第一个元素看做有序序列，把第二个元素到最后一个元素看做待排序序列；

■ 从头到尾，依次扫描待排序序列，将待排序序列的每个元素，从已排序序列尾依次向前扫描，找到合适的位置插入(若在已排序序列中找到相同值， 则将该元素插入到已存在元素后)

### ■ 时间复杂度: $O(n^2)$ 稳定排序算法

- ```

public void sort(int[] arr){
    for (int i = 1; i < arr.length; i++) {
        int nextVal = arr[i];
        int j = i-1;
        //目标值从已排序队列尾依次扫描,处理已排序队列值位置
        for (; j >= 0 && nextVal < arr[j]; j--) {
            arr[j + 1] = arr[j];
        }
        //将目标值插入到合适的位置
        arr[j + 1] = nextVal;
    }
}

```

- 优化: 折半插入排序

- 概述: 原理同直接插入排序;  
不同: 在扫描已排序序列时, 使用折半查找法查找插入位置

- ```

public void binaryInsertSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int nextVal = arr[i];
        //折半查找扫描位置
        int low = 0;
        int high = i - 1;
        while (low <= high) {
            int mid = (high + low) / 2;
            if (arr[mid] > nextVal) {
                high = mid - 1;
            }else{
                low = mid + 1 ;
            }
        }
        for (int j = i; j > high+1; j--) {
            arr[j] = arr[j - 1];
        }
        arr[high + 1] = nextVal;
    }
}

```

- 希尔排序

- 概述(原理):  
希尔排序(Shell's Sort)又称 递减增量排序(Diminishing Increment Sort), 是一种插入排序的更高效的改进版本  
基本思想:  
先将整个待排序序列分割成若干个子序列分别进行插入排序, 待整个序列基本有序时, 再对全体记录进行一次插入排序。  
分割增量逐次递减, 直到1  
希尔增量: 每次增量为原来的一半  $gap = gap / 2$
  - 最差时间复杂度 $O(n^2)$  稳定性: 不稳定

```

■ public void sort(int[] arr) {
    //希尔增量,最坏时间复杂度O(n^2)
    int gap = arr.length / 2;
    while (gap > 0) {
        //内部为插入排序
        for (int i = gap; i < arr.length; i++) {
            int tmp = arr[i];
            int j = i - gap;
            while (j >= 0 && arr[j] > tmp) {
                arr[j + gap] = arr[j];
                j -= gap;
            }
            arr[j + gap] = tmp;
        }
        //逐步减小增量,直到1
        gap = gap / 2;
    }
}

```

#### ■ 归并排序

- 概述: 建立在归并操作上的有效算法, 是分治法的一种典型应用。  
将已有序列的子序列合并, 得到完整的有序序列; 即先使每个子序列有序, 再使序列段有序。  
将2个有序子序列合成一个1个完整序列, 成为二路归并。  
原理:

- 将序列每相邻的两个数字进行归并操作(merge), 形成 $\text{floor}(n/2+n\%2)$ 个序列, 排序后每个序列包含2个元素,
- 将上述的序列再次归并, 形成 $\text{floor}(n/4)$ 个序列, 每个序列包含4个元素  
重复2步骤, 直到所有排序完毕。

- 稳定性: 稳定
- 时间复杂度:  $O(n \log n)$   
速度仅次于快速排序
- 实现方法
  - ■ ■ 自上而下的递归
  - ■ ■ 自下而上的迭代

#### ■ @Override

```

public void sort(int[] arr) {
    int[] temp = new int[arr.length]; //在排序前, 先建好一个长度等于原数组长度的临时数组, 避免递归中频繁开辟空间
    sort(arr, 0, arr.length - 1, temp);
}

private static void sort(int[] arr, int left, int right, int[] temp) {
    if (left < right) {
        int mid = (left + right) / 2;
        sort(arr, left, mid, temp); //左边归并排序, 使得左子序列有序
    }
}

```

```

        sort(arr, mid + 1, right, temp); // 右边归并排序，使得右子序列有序
        merge(arr, left, mid, right, temp); // 将两个有序子数组合并操作
    }
}

private static void merge(int[] arr, int left, int mid, int right, int[] temp) {
    int i = left; // 左序列指针
    int j = mid + 1; // 右序列指针
    int t = 0; // 临时数组指针
    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[t++] = arr[i++];
        } else {
            temp[t++] = arr[j++];
        }
    }
    while (i <= mid) { // 将左边剩余元素填充进temp中
        temp[t++] = arr[i++];
    }
    while (j <= right) { // 将右序列剩余元素填充进temp中
        temp[t++] = arr[j++];
    }
    t = 0;
    // 将temp中的元素全部拷贝到原数组中
    while (left <= right) {
        arr[left++] = temp[t++];
    }
}

```

- Arrays.sort() 为优化版的归并排序
- 快速排序
  - 概述:
 

快速排序又是一种分治法的典型应用。本质上是冒泡排序的递归分治法。

原理：通过一趟排序将要排序的序列分割成2个独立部分，其中一部分的所有数据比另一部分的所有数据都要小；然后再按此方法对这两部分数据分别进行快速排序，整个过程可以递归进行，以此达到整个数据变成有序序列
  - 算法步骤
    - 从数列中挑出一个元素，称为 "基准" (pivot)；
    - 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区 (partition) 操作；
    - 递归地 (recursive) 把小于基准值元素的子数列和大于基准值元素的子数列排序；
  - 稳定性: 不稳定
  - 时间复杂度:  $O(n \log(n))$

- @Override
 

```

public void sort(int[] arr) {
    quickSort(arr, 0, arr.length - 1);
}

private int[] quickSort(int[] arr, int left, int right) {
    if (left < right) {
        int partitionIndex = partition(arr, left, right);
        quickSort(arr, left, partitionIndex - 1);
        quickSort(arr, partitionIndex + 1, right);
    }
    return arr;
}

private int partition(int[] arr, int left, int right) {
    // 设定基准值 (pivot)
    int pivot = left;
    int index = pivot + 1;
    for (int i = index; i <= right; i++) {
        if (arr[i] < arr[pivot]) {
            swap(arr, i, index);
            index++;
        }
    }
    swap(arr, pivot, index - 1);
    return index - 1;
}

private void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
      
```

- 堆排序
- 计数排序
- 桶排序
- 基数排序

## 排序算法的稳定性

- 概述:
 

假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中， $r[i]=r[j]$ ，且 $r[i]$ 在 $r[j]$ 之前，而在排序后的序列中， $r[i]$ 仍在 $r[j]$ 之前，则称这种排序算法是稳定的；否则称为不稳定的。
- 常见排序算法的稳定性
  - 不稳定
    - 堆排序、快速排序、希尔排序、直接选择排序

- 稳定
  - 基数排序、冒泡排序、直接插入排序、折半插入排序、归并排序
- 稳定性的意义:

如果只是简单的进行数字的排序, 那么稳定性将毫无意义。

如果排序的内容仅仅是一个复杂对象的某一个数字属性, 那么稳定性依旧将毫无意义

如果要排序的内容是一个复杂对象的多个数字属性, 但是其原本的初始顺序毫无意义, 那么稳定性依旧将毫无意义。

除非要排序的内容是一个复杂对象的多个数字属性, 且其原本的初始顺序存在意义, 那么我们需要在二次排序的基础上保持原有排序的意义, 才需要使用到稳定性的算法。

例如要排序的内容是一组原本按照价格高低排序的对象, 如今需要按照销量高低排序, 使用稳定性算法, 可以使得想同销量的对象依旧保持着价格高低的排序展现, 只有销量不同的才会重新排序。(当然, 如果需求不需要保持初始的排序意义, 那么使用稳定性算法依旧将毫无意义)

换句话说, 以某种关键字的方式排序后, 能不影响到其他关键字原来排序结果的方法就是稳定的, 比如一开始按照价格高低排序结果为 a(10元, 卖了5个) b(8元, 卖了20个) c(6元, 卖了20个) d(4元, 卖了30个), 则按照销量重拍后如果保持 d(30个, 价格为4元) b(20个, 价格为8元) c(20个, 价格为6元) a(5个, 价格为10元), 则说明该方法为稳定的, 而如果出现c在b前, 破坏了排序前b在c前的顺序, 则说明这个方法是不稳定的

## 查找算法

- 二分查找
- (树)广度优先搜索
  - 它遍历一个层的所有节点, 然后再进入下一层。

这种遍历也称为层序遍历, 它从根开始, 从左到右访问树的所有层。  
(层序遍历需要用到队列)
- (树)深度优先搜索
  - 在遍历下一个同级对象之前, 它会在每个子节点中尽可能深入
  - 有前序遍历, 中序遍历, 后序遍历  
(针对父节点的遍历次序区分的)

## 动态规划

- 动态规划(Dynamic Programming, DP)是运筹学的一个分支, 是求解决策过程最优化的过程。
- 动态规划算法通常用于求解具有某种最优性质的问题
- 动态规划概念

多阶段决策问题中, 各个阶段采取的决策, 一般来说是与时间有关的, 决策依赖于当前的状态, 又随即引起状态的转移,

一个决策序列就是在变化的状态中产生出来的, 固有"动态"的含义, 称这种解决多阶段决策的最优化问题的方法称为动态规划方法。
- 适用条件
  - 无后效性

将各阶段按一定次序排列好后, 对于某个给定的阶段状态, 它以前各个阶段的状态无



法直接影响它未来的决策,而只能通过当前的这个状态.

换句话说,每个状态都是的过去历史的一个完整总结.这就是无后向性,又称无后效性.

- 最优化原理(最优子结构)

一个最优化策略具有这样的性质,不论过去状态和决策如何,对前面的决策所形成的状态而言,余下的诸决策必须构成最优策略.

简而言之,一个最优化策略的子策略总是最优的.一个问题满足最优化原理又称其具有最优子结构性性质.

- 子问题重叠性

动态规划算法的关键在于解决冗余,这是动态规划算法的根本目的.

动态规划实质上是一种以空间换时间的技术,它在实现的过程中,不得不存储产生过程中的各种状态, 所以它的空间复杂度要大于其他的算法.

选择动态规划算法是因为动态规划算法在空间上可以承受,而搜索算法在时间上却无法承受,所以我们舍空间而取时间.

- 动态规划解决问题步骤

- (符合动态规划解决的问题:复杂问题可以被拆分成相似子问题, 且满足无后效性(已知当前阶段的状态,且之前各阶段的状态不会直接影响之后发展阶段的状态),最优子结构)

- 定义状态

- 定义状态转移方程

- 边界值初始化(构建算法或编码)

解决问题时,可通过画图,找出问题规律, 及最优子结构的算法

- 动态规划一般有2种解决问题思路

- 第一种思路是一个一维的 dp 数组:

```
int n = array.length;
int[] dp = new int[n];
for (int i = 1; i < n; i++) {
    for (int j = 0; j < i; j++) {
        dp[i] = 最值(dp[i], dp[j] + ...)
    }
}
```

这种思路运用相对更多一些, 尤其是涉及两个字符串/数组的子序列.

如: 在子数组 `array[0..i]` 中, 我们要求的子序列 (最长递增子序列) 的长度是 `dp[i]`。

- 第二种思路是一个二维 dp 数组

```
int n = arr.length;
int[][] dp = new dp[n][n];
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (arr[i] == arr[j])
            dp[i][j] = dp[i][j] + ...
        else
            dp[i][j] = 最值(...)
    }
}
```

}

本思路中 dp 数组含义又分为「只涉及一个字符串」和「涉及两个字符串」两种情况。

2.1 涉及两个字符串/数组时（比如最长公共子序列），dp 数组的含义如下：

在子数组 `arr1[0..i]` 和子数组 `arr2[0..j]` 中，我们要求的子序列（最长公共子序列）长度为 `dp[i][j]`。

2.2 只涉及一个字符串/数组时（比如最长回文子序列），dp 数组的含义如下：

在子数组 `array[i..j]` 中，我们要求的子序列（最长回文子序列）的长度为 `dp[i][j]`。

## ■ 动态规划问题

### ■ 爬楼梯(ClimbStair,简单)

- 假设你正在爬楼梯。需要  $n$  阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？

注意：给定  $n$  是一个正整数。

示例 1：

输入：2

输出：2

解释：有两种方法可以爬到楼顶。

- (一维dp数组解决)

定义状态：

假设 `dp[i]` 为第  $n$  阶时可以爬到楼顶的方法数

状态转移方程：

$dp[i] = dp[i-2] + dp[i-1] \ (i > 2)$

初始值：

`dp[0] = 1, dp[1] = 1;`

### ■ 最大连续子数组(MaxContinueSubArray,简单)

- 给定一个整数数组 `nums`，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例：

输入：[-2,1,-3,4,-1,2,1,-5,4]

输出：6

解释：连续子数组 [4,-1,2,1] 的和最大，为 6。

- (一维dp数组解决)

1.定义状态

定义 `dp[i]` 为到第  $i$  位时,最大连续子数组的和

2.定义状态转移方程

由 `dp[i]` 定义得出  $dp[i] = \max(dp[i-1] + nums[i], nums[i])$  其中  $i > 0$ , 且 `dp[i-1]` 和 `dp[i]` 取其中最大值, 且需与最大值 `max` 比较, 并保存 `max`

3.初始值及编码

`dp[0] = mn[0]`

### ■ 股票最大收益(StockMaxProfit,简单)

- 给定一个数组，它的第  $i$  个元素是一支给定股票第  $i$  天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。

注意：你不能在买入股票前卖出股票。

示例 1：

输入：[7,1,5,3,6,4]

输出: 5

解释: 在第 2 天 (股票价格 = 1) 的时候买入, 在第 5 天 (股票价格 = 6) 的时候卖出, 最大利润 = 6-1 = 5。

注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格; 同时, 你不能在买入前卖出股票。

- 定义状态

假设dp[i]为第i天最大利润

- 定义状态转移方程

由于只完成一次买入,卖出,且不能再买入前卖出,定义buy为买入金额,buy=n[0]为初始值

$dp[i] = \max\{n[i]-buy, dp[i-1]\}$ , 若  $n[i] < buy$  时, 将buy赋值为  $n[i]$

- 初始值或编码

$dp[0] = dp[1] = 0$ ;

- 子集问题(SubSets,中等)

- 子集问题

给定一组不含重复元素的整数数组 nums, 返回该数组所有可能的子集 (幂集)。

说明: 解集不能包含重复的子集。

示例:

输入: nums = [1,2,3]

输出:

```
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

- 定义状态

假设dp[x]为第x位时,存在的不重复数组

- 定义状态转移方程

$dp[0] = []$

$dp[1] = [], [1] = dp[0] + dp[0,1]$

$dp[2] = [], [1].[2], [1,2] = dp[1] + dp[0..1+2]$

$dp[3] = [], [1].[2], [1,2].[3], [1,3].[2,3], [1,2,3] = dp[2] + dp[0..2+3]$

得出  $dp[x] = dp[x-1] + dp[0..x-1+x]$   $x>0 \&\& x=0 = dp[0] = []$ ,  $0..x-1+x$  表示为, 0到x-1的数组都与x组成新数组

- 初始值(编写算法或代码)

$dp[0] = []$

- 矩阵最短路径和问题(MatrixMinPathSum,中等)

- 给定一个矩阵(m,n), 从左上角开始每次只能向右或者向下走,最后到达右下角的位置,路径上所有的数累加起来就是路径和,返回所有的路径中最小的路径和.

给定 (m,n) 如下:

1 3 5 9

8 1 3 4

5 0 6 1

8 8 4 0

■ 解法1:

使用一个二维的DP矩阵来求解:

对于DP,第一行和第一列只有一种走法,就是从左到右或从上到下的累加,可以先初始化,然后其他元素可以用转移方程填充,直到矩阵填充完成;

状态转移方程:  $dp[i][j] = \min\{dp[i-1][j], dp[i][j-1]\} + mn[i][j]$  ( $i>1, j>1$ )

■ 定义状态

定义 $dp[i][j]$ 为最终最短路径和

■ 定义状态转移方程

由于只能向右或向下走, 则存在 $d[i][j]$ 的和可能为:

a.  $d[i-1][j]+mn[i][j]$  向右到终点

b.  $d[i][j-1]+mn[i][j]$  向下到终点

由于求最短路径,所以 $d[i][j] = \min\{d[i-1][j], d[i][j-1]\}+mn[i][j]$

■ 初始值

由于第0行和第0列不满足 $d[i-1], d[j-1]$ 情况,所以,需要先初始化 $d[0][0..j], d[0..i][j]$ 的值

最终方程为:  $d[i][j] = \min\{d[i-1][j], d[i][j-1]\}+mn[i][j]$  ( $i>1, j>1$ )

解法2(优化解法):

如果使用二维数组,对于m行n列的的数组,空间复杂度是 $O(m*n)$ .

动态规划中常用的优化方法之一就是仅使用一个一维数组在进行迭代,但空间压缩也有局限性,但不能记录最后结果的路径.

如果需要完整路径还需要二维动态规划表。

解:定义大小为N的dp数组,

对于第一行,  $dp[i]=dp[i-1]+m[0][i]$ ,在求第二行中的  $dp[i]$  时可以覆盖第一行  $dp[i]$ ,

第二行 $dp[i]=\text{Math.min}(dp[i], dp[i-1]) + m[i][j]$ 。

逐行计算,最终得出 $dp[m-1]$ 值

■ 最长上升子序列(LengthOfLIS,中等)

■ 给定一个无序的整数数组, 找到其中最长上升子序列的长度。

示例:

输入: [10,9,2,5,3,7,101,18]

输出: 4

解释: 最长的上升子序列是 [2,3,7,101], 它的长度是 4。

说明:

可能会有多种最长上升子序列的组合, 你只需要输出对应的长度即可。

你算法的时间复杂度应该为  $O(n^2)$ 。

进阶: 你能将算法的时间复杂度降低到  $O(n \log n)$  吗?

■ 时间复杂度 $O(n^2)$

■ 定义状态

假设 $dp[x]$  为以x结尾时,LIS的长度,则结果为 $\max(dp[x])$

- 定义状态转移方程

假设比x小的每一个p,存在  $x > p$ , 则  $dp[x] = dp[p] + 1$ , 因为x大于p,就能构造一个以x结尾的上升子序列,

则方程为  $dp[x] = \max\{dp[p]\} + 1 \text{ (list}(p) < \text{list}(x))$

- 初始值(编写算法或代码)

$dp[0..x] = 1$

时间复杂度  $O(n \log n)$

解决方案:

- 将序列按值分成若干子序列,子序列的新值总是小于子序列最后一位, 若循环时,值小于所有子序列的最后一位(从左往右搜索,此处所有子序列的最后一位值是从小到大排列),则开始新序列

最后的子序列数就是最长上升子序列长度,最终的最长上升子序列可能有多组.

该问题类似扑克牌分牌,把牌分堆,每次选的牌都依次放入最近小余最后一张牌的牌堆里

2.其中查找当前值放到哪个子序列可以使用二分法查找存放位置,使得时间复杂度可以为  $\log n$

- //时间复杂度  $O(n^2)$

```
public int lengthOfLIS(int[] list) {
    int len = list.length;
    if (len == 0) {
        return 0;
    }
    int[] dp = new int[len+1];
    for (int i = 0; i < len ; i++) {
        dp[i] = 1;
    }
    int maxLen = 1;
    for (int x = 0; x < len; x++) {
        for (int p = 0; p < x; p++) {
            //list[p] 为每个小余 list[x]的数
            if (list[p] < list[x]) {
                dp[x] = Math.max(dp[x], dp[p] + 1);
            }
        }
        maxLen = Math.max(maxLen, dp[x]);
    }
    return maxLen;
}
```

- //时间复杂度  $O(n \log n)$

```
public int lengthOfLIS2(int[] list) {
    int len = list.length;
    if (len == 0) {
        return 0;
    }
    int[] dp = new int[len];
    int lisLen = 0;
    for (int i = 0; i < len; i++) {
```

```

int curr = list[i];
//二分法查找值保存位置
int start = 0, end = lisLen;
while (start < end) {
    int mid = (end + start) / 2;
    if (dp[mid] >= curr) {
        end = mid;
    } if (dp[mid] < curr) {
        start = mid + 1;
    }
}
if (start == lisLen) {
    lisLen++;
}
dp[start] = curr;
}
return lisLen;
}

```

- 最长回文子串(LongestPalindromeSubstring,中等)

- 最长回文子串

如果一个数字序列逆置之后跟原序列是一样的就称这样的数字序列为回文序列  
给定一个字符串  $s$ ，找到其中最长的回文子串，并返回该序列的长度。可以假设  $s$  的最大长度为 1000。

示例 1:

输入:

"bbab"

输出:

4

一个可能的最长回文子串为 "bab"。

示例 2:

输入:

"cbbd"

输出:

2

一个可能的最长回文子串为 "bb"。

提示:

$1 \leq s.length \leq 1000$

$s$  只包含小写英文字母

- 分析:

假设  $s(i..j)$  为最长回文子串,则存在:

a.  $s(i) == s(j)$

b.  $s(i+1..j-1)$  存在  $s(i+1) == s(j-1)$   $j \geq i+2$

- 定义状态

假设  $p[i][j]$  表示为  $s[i..j]$  是否是回文子串,若  $p[i][j]$  为回文子串,则  $p[i][j] == p[i+1][j-1]$  且  $s[i] == s[j]$

- 定义状态转移方程

若 $p[i][j] == p[i+1][j-1]$  且 $s[i] == s[j]$ ,则为回文子串,记录开始下标及子串长度

- 初始值(编写算法或代码)

当数组只有1位时 $s[i] == s[j] = \text{true}$ 为回文子串

当数组有2位时 $s[i] == s[j]$ 为回文子串

循环序列:(如:bbbab)

- 先计算单个字符的子串都为回文子串( $i, j$  间隔为0)

b b b a b

true true true true true

- 计算相邻的字符组成的子串是否是回文子串( $i, j$  间隔为1)

b b b a b

||

||

||

||

- 计算 $i, j$ 间隔为2的子串是否为回文子串,其中会用到1,2中的结果

b b b a b

|\_\_|

|\_\_|

|\_\_|

- 计算 $i, j$ 间隔为 $i$ 的子串是否为回文子串, 直至 $s[0], s[j]$ 最长回文子串

b b b a b

|\_\_|

|\_\_|

- 最终结果

b b b a b

|\_\_\_\_|

```

    public int longestPalindromeSubseq2(String s) {
        int len = s.length();
        char[] chars = s.toCharArray();
        boolean p[][] = new boolean[len][len];
        int max = 0;
        for (int step = 0; step < len; step++) {
            for (int i = 0; i + step < len; i++) {
                int j = i + step;
                //p[i][j] == p[i+1][j-1] 且s[i] == s[j],
                //只有1位数时为true
                if (i == j) {
                    p[i][j] = true;
                    //相邻的2位数
                } else if (i+1 == j) {
                    p[i][j] = chars[i] == chars[j];
                } else {
                    //相差n位的子串
                }
            }
        }
    }

```

```

        p[i][j] = chars[i] == chars[j] && p[i + 1][j - 1];
    }
    if (p[i][j]) {
        max = Math.max(max, j - i + 1);
    }
}
}
return max;
}

```

- 最长回文子序列(LongestPalindromeSubseq,中等)

- 最长回文子序列

如果一个数字序列逆置之后跟原序列是一样的就称这样的数字序列为回文序列  
给定一个字符串  $s$ ，找到其中最长的回文子序列，并返回该序列的长度。可以假设  $s$  的最大长度为 1000。

示例 1:

示例 1:

输入:

"bbbab"

输出:

4

一个可能的最长回文子序列为 "bbbb"。

示例 2:

输入:

"cbbd"

输出:

2

一个可能的最长回文子序列为 "bb"。

提示:

$1 \leq s.length \leq 1000$

$s$  只包含小写英文字母

- 定义状态

假设  $dp[i][j]$  表示为  $s[i,j]$  的最长回文子序列长度

- 定义状态转移方程

若已知  $dp[i+1][j-1]$  ( $s[i+1,j-1]$  的最长回文子序列长度) 的值,求  $dp[i][j]$  的值有 2 种情况:

- 当  $s[i]==s[j]$  时  $s[i,j]$  一定存在最长回文子序列,且长度+2,即

$dp[i][j] = dp[i+1][j-1] + 2$

- 当  $s[i]!=s[j]$  时,则需要分别  $s[i],s[j]$  加入到  $s[i+1,j-1]$  序列中,去最大的回文子序列长度,即

$dp[i][j] = \max\{dp[i][j-1], dp[i+1][j]\}$

- 初始值(编写算法或代码)

当数组只有 1 位时  $i=j$  时为回文子序列,长度为 1

$i < j$ , 所以子序列长度为 0

```

- public int longestPalindromeSubseq(String s) {
    int len = s.length();
    if (len == 1) {
        return 1;
    }
}

```



```

    }
    char[] chars = s.toCharArray();
    int dp[][] = new int[len][len];
    for (int i = 0; i < len; i++) {
        dp[i][i] = 1;
    }
    //从下往上,从左往右计算最终dp[0][len-1]
    //    for (int i = len-1; i >= 0; i--) {
    //        for (int j = i+1; j < len; j++) {
    //            if (chars[i] == chars[j]) {
    //                dp[i][j] = dp[i + 1][j - 1] + 2;
    //            } else {
    //                dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
    //            }
    //        }
    //    }
    //从左右,从下往上计算最终dp[0][len-1]
    for (int step = 1; step < len; step++) {
        for (int i = 0; i + step < len; i++) {
            int j = i + step;
            if (chars[i] == chars[j]) {
                dp[i][j] = dp[i + 1][j - 1] + 2;
            } else {
                dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[0][len - 1];
}

```

## 递归

- 分而治之，分治法

## 狄克斯特拉算法

## 贪婪算法

## 树

- 树
  - 树 (Tree) 是  $n$  ( $n \geq 0$ ) 个结点的有限集。 $n=0$  时称为空树。在任意一颗非空树中：
    - 1) 有且仅有一个特定的称为根 (Root) 的结点；
    - 2) 当  $n > 1$  时，其余结点可分为  $m$  ( $m > 0$ ) 个互不相交的有限集  $T_1$ 、 $T_2$ 、.....、 $T_n$ ，其中每一个集合本身又是一棵树，并且称为根的子树。
 此外，树的定义还需要强调以下两点：
    - 1)  $n > 0$  时根结点是唯一的，不可能存在多个根结点，数据结构中的树只能有一个根

结点。

2)  $m > 0$  时, 子树的个数没有限制, 但它们一定是互不相交的。

- 节点的度
  - 结点拥有的子树数目称为结点的度。
- 节点关系
  - 双亲节点
  - 孩子节点
  - 兄弟节点
- 节点层次
  - 根为第一层, 根的孩子为第二层, 以此类推
- 树的深度
  - 树中结点的最大层次数称为树的深度或高度
- 二叉树(BinaryTree)
  - 二叉树是  $n(n \geq 0)$  个结点的有限集合, 该集合或者为空集 (称为空二叉树), 或者由一个根结点和两棵互不相交的、分别称为根结点的左子树和右子树组成。
  - 特点
    - 每个节点最多有2棵子树, 所以二叉树中不存在度大于2的节点
    - 左子树和右子树是有顺序的, 次序不能任意颠倒
    - 即使树中某节点只有1棵子树, 也要区分它是左子树还是右子树
  - 性质
    - 在二叉树的第  $i$  层上最多有  $2^{(i-1)}$  个节点
    - 二叉树中如果深度为  $k$ , 那么最多有  $2^k - 1$  个节点 ( $k \geq 1$ )
    - $n_0 = n_2 + 1$   $n_0$  表示度数为0的节点,  $n_2$  表示度数为2的节点数
    - 在完全二叉树中, 具有  $n$  个节点的完全二叉树的深度为  $\lfloor \log_2 n \rfloor + 1$  其中  $\lfloor \log_2 n \rfloor$  是向下取整
    - 若对含  $n$  个节点的完全二叉树从上到下切从左至右进行1至  $n$  的编号, 则完全二叉树中任意一个编号  $i$  的特性为
      - (1) 若  $i=1$ , 则该节点是二叉树的根, 无双亲, 否则, 编号为  $\lfloor i/2 \rfloor$  的节点为其双亲节点
      - (2) 若  $2i > n$ , 则该节点无左孩子, 否则, 编号为  $2i$  的节点为其左孩子节点
      - (3) 若  $2i+1 > n$ , 则该节点无右孩子节点, 否则编号为  $2i+1$  的节点为其右孩子节点
  - 斜树
    - 左斜二叉树
      - 所有节点只有左子树
    - 右斜二叉树
      - 所有节点只有右子树
  - 满二叉树
    - 在一棵二叉树中, 如果所有分支节点都存在左子树和右子树, 并且所有叶子都在同一层上, 这样的二叉树成为满二叉树
    - 特点
      - 叶子只能出现在最下一层。出现在其它层就不可能达成平衡

- ■ 非叶子节点的度一定是2
    - ■ 在同样深度的二叉树中，满二叉树的节点个数最多，叶子数量最多。
- 完全二叉树
  - 对一棵具有n个节点的二叉树按层编号，如果编号为i( $1 \leq i \leq n$ )的节点与同样深度的满二叉树中编号为i的节点在二叉树中位置完全相同，则这棵二叉树成为完全二叉树
  - 特点
    - ■ 叶子节点只能出现在最下层和次下层
    - ■ 最下层的叶子节点集中在树的左部
    - ■ 倒数第二层若存在叶子节点，一定在右部连续位置
    - ■ 如果节点度为1，则该节点只有左孩子，没有右子树
    - ■ 同样节点数目的二叉树，完全二叉树深度最小
    - (满二叉树一定是完全二叉树，反之不一定)
- 二叉树的存储结构
  - 顺序存储(一维数组)
    - 二叉树的顺序存储结构就是使用一维数组存储二叉树中的结点，并且结点的存储位置，就是数组的下标索引。
    - 顺序存储一般适用于完全二叉树
  - 二叉链表(带有数据和2个指针域的链表)
    - 既然顺序存储不能满足二叉树的存储需求，那么考虑采用链式存储。由二叉树定义可知，二叉树的每个结点最多有两个孩子。因此，可以将结点数据结构定义为一个数据和两个指针域。
- 二叉树遍历
  - 从二叉树的根节点出发，按照某种次序一次访问二叉树中的所有节点，使得每个节点被访问一次，且仅被访问一次。
  - 四种遍历方式
    - 前序遍历（深度优先搜索）
      - 从二叉树的根节点出发，当第一次到达结点时就输出结点数据，按照先向左再向右的方向访问
      - ABDH I E J C F G
    - 中序遍历（深度优先搜索）
      - 从二叉树的根节点出发，当第二次到达节点时输出结点数据（第一次到达时不输出），按照先向左再向右的访问访问
      - HDI B J E A F C G
    - 后序遍历（深度优先搜索）
      - 从二叉树的根节点出发，当第三次到达结点时就输出结点数据，按照先向左再向右的方向访问
      - H I D J E B F G C A
    - 层序遍历（广度优先搜索）
      - 按照树的层次自上而下遍历二叉树

- ABCDEFGHIJ

- 已知前序遍历序列和后序遍历序列，不可以唯一确定一棵二叉树。

- 二叉树的操作

<https://www.baeldung.com/java-binary-tree>

- 创建树(使用二叉链表)

- 插入数据

- ■ 根为空时，先创建根
- ■ 若值大于当前节点，则将新节点放到右子树
- ■ 若值小于当前节点，则将新节点放到左子树
- ■ 当当前节点值为null时，则将新节点插入该位置
- ```
private Node insertToNode(Node curr, T value) {  
    if (curr == null) {  
        return new Node<>(value);  
    }  
    if (curr.compare(value) > 0) {  
        curr.left = insertToNode(curr.left, value);  
    } else if (curr.compare(value) < 0) {  
        curr.right = insertToNode(curr.right, value);  
    }  
    //若值相同,则节点已存在  
    return curr;  
}
```

- 查找数据

- ■ 值等于根节点值时，返回根
- ■ 值大于当前节点值时，搜索节点右子树
- ■ 值小于当前节点值时，搜索节点左子树
- ```
public Node get(Node curr, T value) {  
    if (curr == null) {  
        return null;  
    }  
    if (curr.compare(value) == 0) {  
        return curr;  
    }  
    return curr.compare(value) > 0 ? get(curr.left, value) : get(curr.right, value);  
}
```

- 删除数据

- ```
public Node remove(T value) {  
    return root = remove(root, value);  
}
```
- /\*\*

- 删除节点(节点可能需要重组)

- @param curr

- @param value

- @return
  - \*/
  - private Node remove(Node curr,T value) {
  - if (curr == null) {
  - return null;
  - }
  - //节点重组
  - if (curr.compare(value) == 0) {
  - // Node to delete found
  - // 一个节点没有子节点
  - if (curr.left == null && curr.right == null) {
  - return null;
  - }
  - //一个节点只有一个子节点-在父节点中，我们用唯一的子节点替换该节点。
  - if (curr.right == null) {
  - return curr.left;
  - }
  - if (curr.left == null) {
  - return curr.right;
  - }
  - //一个节点有两个孩子 -这是最复杂的情况，因为它需要对树进行重组
  - //从右结点中找到最小值的节点
  - T smallestValue = (T) findSmallestValue(curr.right);
  - curr.value = smallestValue;
  - //将最小值节点删除
  - curr.right = remove(curr.right, smallestValue);
  - return curr;
  - }
  - if (curr.compare(value)>0) {
  - curr.left = remove(curr.left, value);
  - return curr;
  - }
  - curr.right = remove(curr.right, value);
  - return curr;
  - }
  - /\*\*

- 找出结点中最小值的结点

- @param root
- @return
  - \*/
  - private T findSmallestValue(Node root) {
  - return root.left == null ? root.value : (T)findSmallestValue(root.left);
  - }

- 4种遍历

- 前序遍历

- public void prevOrderIterator(Node curr) {

```

    if (curr == null) {
        return;
    }
    System.out.print(curr.getValue() + ",");
    prevOrderIterator(curr.left);
    prevOrderIterator(curr.right);
}

```

- 中序遍历

- public void midOrderIterator(Node curr) {
 if (curr == null) {
 return;
 }
 midOrderIterator(curr.left);
 System.out.print(curr.getValue() + ",");
 midOrderIterator(curr.right);
 }

- 后序遍历

- public void postOrderIterator(Node curr) {
 if (curr == null) {
 return;
 }
 postOrderIterator(curr.left);
 postOrderIterator(curr.right);
 System.out.print(curr.getValue() + ",");
 }

- 层序遍历

- public void levelOrderIterate(Node root) {
 if (root == null) {
 return;
 }
 //add(队列满,抛异常)/offer(队列满,返回false)/put(队列满,阻塞)
 //remove(移除数据,队列空,抛异常)/poll(取数据,队列空,返回false)/take(取数据,队列空,阻塞)
 //element(从队列头查询元素,队列空,抛异常)/peek(队列为空,返回null)
 Queue<Node> queue = new LinkedList();
 ((LinkedList<Node>) queue).add(root);
 Node front;
 while (!queue.isEmpty()) {
 front = queue.remove();
 if (front.left != null) {
 ((LinkedList<Node>) queue).add(front.left);
 }
 if (front.right != null) {
 ((LinkedList<Node>) queue).add(front.right);
 }
 }
 }

```

    }
    System.out.print(front.getValue()+",");
}
}

```

- 获取根深度

- `public int length(Node root) {`  
`return root == null ? 0: 1`  
`+Math.max(length(root.left),length(root.right));`  
`}`

- 二叉树的打印

- 水平树(前序遍历，深度优先)

- `/**`

- 打印水平树2-中序遍历

```

*/
public void printHorizontalTree2() {
    System.out.println(inOrderPrint(tree.getRoot(),new StringBuffer(),"",""));
}
public String inOrderPrint(BinaryTree.Node root,StringBuffer sb,String
pointer,String parentPadding) {
    if (root == null) {
        return "";
    }
    String paddings = getPadding(level(root))+parentPadding;
    inOrderPrint(root.getLeft(),sb,LEFT_SUBTREE_FLAG2,paddings);
    sb.append(paddings+" ");
    sb.append(pointer).append(root.getValue()).append(NEWLINE_FLAG);
    inOrderPrint(root.getRight(),sb,RIGHT_SUBTREE_FLAG,paddings);
    return sb.toString();
}
public String getPadding(int level) {
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < level; i++) {
        sb.append(" ");
    }
    return sb.toString();
}

```

- 垂直树(层序遍历，广度优先)

- 平衡二叉树(AVLTree)

- 自平衡二叉查找树

- 在AVL树中，任一节点对应的两颗子树的最大高度差为1，因此也被称为高度平衡树。

插入和查找的最坏时间复杂度 $O(\log n)$ 。

增加和删除元素的操作则可能需要一次或多次树旋转，以实现树平衡

- 平衡因子

- 某个节点的左子树与右子树的高度(深度)差即为该节点的平衡因子(BF,BalanceFactor);  
平衡二叉树中不存在平衡因子大于1的节点。  
在平衡二叉树中，节点的平衡因子只能取0,1,-1分别对应左右子树等高，左子树高，右子树高
- 旋转
  - AVL树的4中节点插入方式
    - LL 在A的左子树根节点的左子树上插入节点而破坏平衡，右旋
    - RR 在A的右子树根节点的右子树上插入节点而破坏平衡，左旋
    - LR 在A的左子树的根节点的右子树上插入节点而破坏平衡，先左旋后右旋
    - RL 在A的右子树根节点的左子树上插入节点而破坏平衡，先右旋后左旋
- 子主题 4
- 红黑树

## 设计模式

---

### 七大设计原则

- 单一职责原则(Single Responsibility Principle)
  - 一个类或者模块应该有且只有一个改变的原因  
一个类/接口/方法只负责一项职责或职能
  - 优点
    - 降低类的复杂度；
    - 提高类的可读性，因为类的职能单一，看起来比较有目的性，显得简单；
    - 提高系统的可维护性，降低变更程序引起的风险。
- 开闭原则(Open Close Principle)
  - 一个软件实体,如类、模块和函数应该对扩展开放,对修改关闭.即一个软件实体应该通过扩展来实现变化,而不是通过修改已有的代码来实现变化。
  - 优点
    - 开闭原则有利于进行单元测试
    - 开闭原则可以提高复用性
    - 开闭原则可以提高可维护性
    - 面向对象开发的要求
- 里氏替换原则(Liskov Substitution Principle)
  - 里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。  
里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。  
LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。  
实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范



- 具体约束
  - 子类必须实现父类的抽象方法，但不得重写父类的非抽象(已实现的)方法。
  - 子类中可增加自己特有的方法。(可以随时扩展)
  - 当子类覆盖或者实现父类的方法时,方法的前置条件(方法形参)要比父类输入参数更加宽松。否则会调用到父类的方法。
  - 当子类的方法实现父类的抽象方法时，方法的后置条件（即方法的返回值）要比父类更严格。否则会调用到父类的方法。
- 最佳实践
  - 我们最好将父类定义为抽象类，并定义抽象方法，让子类重新定义这些方法，当父类是抽象类时候，父类不能实例化
- 依赖倒置原则(Dependence Inversion Principle)
  - 这个是开闭原则的基础，具体内容：针对接口编程，依赖于抽象而不依赖于具体。
    - 1、上层模块不应该依赖底层模块，它们都应该依赖于抽象。
    - 2、抽象不应该依赖于细节，细节应该依赖于抽象。
- 接口隔离原则(Interface Segregation Principle)
  - 1、客户端不应该依赖它不需要的接口。
  - 2、类间的依赖关系应该建立在最小的接口上。

含义是：要为各个类建立它们需要的专用接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。
- 迪米特法则(Demeter Principle)
  - 一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立
 

如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。
- 组合复用原则(Composite Reuse Principle)
  - 尽量使用合成/聚合的方式，而不是使用继承

## 设计模式(23)

- 创建型(5)
  - 工厂方法(FactoryMethod)
    - 概述
      - 定义一个用于创建对象的接口，让子类决定实例化哪一个类。FactoryMethod使一个类的实例化延迟到其子类。
        - 普通工厂模式
        - 多个工厂方法模式
        - 静态工厂方法模式
    - 样例
      - JDK: J.U.C.ThreadFacotry
      - Mybatis: org.apache.ibatis.datasource.DataSourceFactory
        - JndiDataSourceFactory
        - PooledDataSourceFactory
      - Mybatis: org.apache.ibatis.transaction.TransactionFactory

- JdbcTransactionFactory
  - ManagedTransactionFactory
- 适用性
  - 1. 当一个类不知道它所必须创建的对象类的时候。
  - 2. 当一个类希望由它的子类来指定它所创建的对象的时候。
  - 3. 当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。
- 抽象工厂(AbstractFactory)
  - 概述
    - 提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

抽象工厂模式是对工厂方法模式的再升级，但是二者面对的场景稍显差别。

工厂方法模式面对的目标一般都是单类的，就比如《java设计模式之《工厂方法模式》及使用场景》中所举的例子，目标就是桌子这一类商品。

抽象工厂模式面对的是一个组合体。
  - 样例
    - 动物抽象工厂,生产猫,狗
  - 适用性
    - 1. 一个系统要独立于它的产品的创建、组合和表示时。
    - 2. 一个系统要由多个产品系列中的一个来配置时。
    - 3. 当你要强调一系列相关的产品对象的设计以便进行联合使用时。
    - 4. 当你提供一个产品类库，而只想显示它们的接口而不是实现时。
- 单例模式(Singleton)
  - 概述
    - 保证一个类仅有一个实例，并提供一个访问它的全局访问点。
  - 样例
    - 双重检查锁(DCL)单例 (延迟模式,利用synchronized,volatile)
    - 枚举单例(立即模式,利用枚举属性在编译后为new枚举对象加载)
    - 静态内部类(延迟模式,利用静态属性和静态代码块实现单例对象的加载)
  - 用途
    - 配置文件加载
    - SpringBean加载
- 建造者模式(Builder)
  - 概述
    - 将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。
  - 样例
    - StringBuilder,StringBuffer
    - Guava中的ImmutableMap.ImmutableSet等
    - Mybatis中的Example, SqlSessionFactoryBuilder 等
- 原型模式(Prototype)

- 概述
  - 用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。
- 样例
  - 实现Cloneable，重写clone方法创建新对象  
深复制可以使用ObjectInput/OutputStream
- 结构型(7)
  - 适配器模式(Adapter)
    - 概述
      - 将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。  
适配器模式有三种：类适配器、对象适配器、接口适配器
        - 类适配器
          - 通过继承类，实现对某个类方法的调用
        - 对象适配器
          - 通过传入类实例对象，实现对类方法的调用
        - 接口适配器
          - 对于一个存在n个方法的接口，只会用到某几个方法，通过构造抽象类实现(实现方法体为空)该接口的所有方法，  
然后继承该抽象类实现所需的方法。
    - 适用性
      - 1.你想使用一个已经存在的类，而它的接口不符合你的需求。
      - 2.你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。
      - 3.（仅适用于对象Adapter）你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口。
  - 样例
    - JDK(对象适配器): java.util.Arrays.asList()
    - SpringMVC(接口适配器):  
org.springframework.web.servlet.config.annotation.  
WebMvcConfigurerAdapter  
implements WebMvcConfigurer(interface)
- 桥接模式(Bridge)
  - 概述
    - 将抽象部分与它的实现部分分离，使它们都可以独立地变化。  
这种模式涉及到一个作为桥接的接口，使得实体类的功能独立于接口实现类。这两种类型的类可被结构化改变而互不影响。
  - 适用性
    - 1.你不希望在抽象和它的实现部分之间有一个固定的绑定关系。  
例如这种情况可能是因为，在程序运行时刻实现部分应可以被选择或者切换。

- 2.类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。  
这时Bridge模式使你可以对不同的抽象接口和实现部分进行组合，并分别对它们进行扩充。
- 3.对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。
- 4.正如在意图一节的第一个类图中所示的那样，有许多类要生成。  
这样一种类层次结构说明你必须将一个对象分解成两个部分。
- 5.你想在多个对象间共享实现（可能使用引用计数），但同时要求客户并不知道这一点。

- 注意点

- 1、定义一个桥接口，使其与一方绑定，这一方的扩展全部使用实现桥接口的方式。
- 2、定义一个抽象类，来表示另一方，在这个抽象类内部要引入桥接口，而这一方的扩展全部使用继承该抽象类

- 样例

- JDK: JDBC数据库访问接口

- 组合器模式(Composite)

- 概述

- 将对象组合成树形结构以表示"部分-整体"的层次结构。"Composite使得用户对单个对象和组合对象的使用具有一致性。"

- 适用性

- 1.你想表示对象的部分-整体层次结构。
- 2.你希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

- 样例

- 文件目录
- 教材的章节知识点

- 装饰模式(Decorator)

- 概述

- 动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式相比生成子类更为灵活。

装饰者与被装饰者需要有共同的超类或接口

- 适用性

- 1.在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
- 2.处理那些可以撤消的职责。
- 3.当不能采用生成子类的方法进行扩充时。

- 样例

- 饮料: 雪碧，美年达饮料的描述和价格
- JavaIO InputStream

- 与代理模式的区别

- 装饰模式在于对原有系统业务的扩展或增强
- 代理模式在于增加一些与业务无关的操作

- 外观模式(Facade)

- 概述

- 为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

- 适用性

- 1.当你要为一个复杂子系统提供一个简单接口时。子系统往往因为不断演化而变得越来越复杂。大多数模式使用时都会产生更多更小的类。这使得子系统更具可重用性，也更容易对子系统进行定制，但这也给那些不需要定制子系统的用户带来一些使用上的困难。

- Facade可以提供一个简单的缺省视图，这一视图对大多数用户来说已经足够，而那些需要更多的可定制性的用户可以越过facade层。

- 2.客户程序与抽象类的实现部分之间存在着很大的依赖性。引入facade将这个子系统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性。

- 3.当你需要构建一个层次结构的子系统时，使用facade模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，你可以让它们仅通过facade进行通讯，从而简化了它们之间的依赖关系。

- 样例

- Controller 层访问Service层方法，可将各个Service注册到一个门面上，在Controller可以直接通过门面入口访问各个Service

- 享元模式(Flyweight)

- 概述

- 运用共享技术有效地支持大量细粒度的对象。

- 适用性

- 1.一个应用程序使用了大量的对象。
    - 2.完全由于使用大量的对象，造成很大的存储开销。
    - 3.对象的大多数状态都可变为外部状态。
    - 4.如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。
    - 5.应用程序不依赖于对象标识。由于Flyweight对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值。

- 样例

- JVM字符串常量池

- 代理模式(Proxy)

- 概述

- 为其他对象提供一种代理以控制对这个对象的访问。
      - 与装饰模式类似，代理模式更适用于增加无关业务的处理；或对原始对象的保护

- 适用性

- 1.远程代理（RemoteProxy）为一个对象在不同的地址空间提供局部代表。
    - 2.虚代理（VirtualProxy）根据需要创建开销很大的对象。
    - 3.保护代理（ProtectionProxy）控制对原始对象的访问。

- 4.智能指引 (SmartReference) 取代了简单的指针，它在访问对象时执行一些附加操作。
- 3种代理
  - 静态代理
    - 即普通代理，代理类中将被代理类作为属性来调用被代理类的方法
  - 动态代理
    - 即使用JDK内置方法生成代理类：  
调用java.lang.reflect.Proxy类的静态方法newProxyInstance即可，该方法会返回代理类对象  
static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces,InvocationHandler h )  
接收的三个参数依次为：  
ClassLoader loader：指定当前目标对象使用类加载器，写法固定  
Class<?>[] interfaces：目标对象实现的接口的类型，写法固定  
InvocationHandler h：事件处理接口，需传入一个实现类，一般直接使用匿名内部类
  - Cglib代理
    - /\*\*
- Cglib子类代理工厂
 

```

      */
      public class ProxyFactory implements MethodInterceptor{
      // 维护目标对象
      private Object target;
      public ProxyFactory(Object target) {
          this.target = target;
      }
      // 给目标对象创建一个代理对象
      public Object getProxyInstance(){
          //1.工具类
          Enhancer en = new Enhancer();
          //2.设置父类
          en.setSuperclass(target.getClass());
          //3.设置回调函数
          en.setCallback(this);
          //4.创建子类(代理对象)
          return en.create();
      }
      @Override
      public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {
      }
      }
      
```

  - 静态代理和JDK代理有一个共同的缺点，就是目标对象必须实现一个或多个接口，若没有，则可以使用Cglib代理。
  - 样例

## - Spring中的对象使用CGLIB动态代理

### ■ 行为型(11)

#### ■ 责任链模式(COR)

##### ■ 概述

- 使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。这一模式的想法是，给多个对象处理一个请求的机会，从而解耦发送者和接受者。

##### ■ 适用性

- 1.有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。
- 2.你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。
- 3.可处理一个请求的对象集合应被动态指定。

##### ■ 样例

- Servlet,Jsp中的filter

##### ■ 编码方式

- ■ 责任链模式写法1: 接口方法中传入责任链对象,由责任链对象调度各个责任对象处理  
`void handle(String request, IResponsibility responsibility);`

### ■ 写法2: 责任对象中设置下一个处理者责任对象

`IResponsibility next;`

#### ■ 命令模式(Command)

##### ■ 概述

- 将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤消的操作。

##### ■ 适用性

- 1.抽象出待执行的动作以参数化某对象。
- 2.在不同的时刻指定、排列和执行请求。
- 3.支持取消操作。
- 4.支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。
- 5.用构建在原语操作上的高层操作构造一个系统。

##### ■ 样例

##### ■ 参与者

- 接收者: 命令最终执行对象
- 命令接口/命令接口实现: 创建不同的命令实现对象，封装接收者,定义对外接口调用接收者具体实现
- 请求者: 封装对命令接口的调用，提供给客户端
- 客户端: 负责创建命令，并指定其接收者，并使用请求者执行操作

#### ■ 解释器模式(Interpreter)

- 概述
  - 给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子
- 适用性
  - 当一个语言需要解释执行时，并且可以将语言中的句子表示为抽象语法树时，可以使用解释器模式
- 参与者
  - 定义一个解释器的抽象表达式，抽象一个解释操作，该接口为抽象语法树的所有节点共享
  - 实现一个终结符表达式操作
  - 实现一个非终结符表达式操作
  - 定义Context,包含除解释器外的全局信息
  - 定义Client,构建(或被给定)表示该文法定义的语言中一个特定的句子的抽象语法树。  
该抽象语法树由NonterminalExpression和TerminalExpression的实例装配而成。  
调用解释操作。
- 样例
- 迭代器模式(Iterator)
  - 概述
    - 提供一种方法顺序遍历聚合对象的各种元素，而又不暴露对象的内部表示
  - 适用性
    - 访问一个聚合对象而无需暴露对象的内部表示
    - 支持集合对象的多种表示
    - 为遍历不同的聚合对象提供一个统一的接口（即多态迭代）
  - 参与者
    - Iterator 迭代器，定义访问和遍历元素的接口
    - ConcreteIterator 具体的迭代器实现；对该聚合对象记录遍历位置
    - Aggregate 聚合对象，定义创建相关迭代器对象的接口
    - ConcreteAggregate 具体的聚合对象，实现创建迭代器对象的接口，返回ConcreteIterator
  - 样例
    - Iterable  
List  
Collection
- 中介者模式(Mediator)
  - 概述
    - 用一个中介者来封装一系列对象的交互。中介者使各对象不需要显示的相互引用，从而使耦合松懈，并且能够独立改变它们的交互
  - 适用性
    - 一组对象以定义良好但复杂通信，使得类结构体层级复杂且难以理解



- ■ 一个对象引用很多其他对象，并且直接调用对象进行通信，使得该对象难以复用
    - ■ 想定制一个分布在多个类的行为，而不想生成太多的子类
  - 参与者
    - ■ 中介者: 定义各同事间交互的操作接口
    - ■ 中介者具体实现: 了解维护各同事对象，协调各同事对象间的相互操作
    - ■ 同事类: 每个同事类都知道中介者；每个同事对象与其他对象交互时，都与中介者通信
  - 样例
- 备忘录模式(Memento)
  - 概述
    - 在不破坏封装的前提下获取对象的内部状态，并在对象外部保存该状态，使它在之后可以恢复到先前保存的状态
  - 适用性
    - ■ 保存一个对象在某个时刻的(部分)状态，这样能够在需要的时候恢复到先前的状态
    - ■ 如果用一个接口来让其他对象直接获取到这些状态，则会暴露对象的实现逻辑和破坏封装性
  - 参与者
    - ■ Memento 备忘录，保存原发器的内部状态
    - ■ Originator 原发器，创建备忘录；使用备忘录恢复先前状态
    - ■ Caretake 管理者：保存备忘录，但不能操作和检查备忘录内容
  - 样例(应用场景)
    - 浏览器回退操作
    - 编辑撤销与重做
    - 棋盘悔棋
- 观察者模式(Observer)
  - 概述
    - 定义对象间的一种一对多的依赖关系，在一个对象发生改变时，所有依赖它的对象都会被通知并自动更新
  - 适用性
    - ■ 当一个抽象模型有2个方面，其中一个方面依赖于另一个方面。将二者封装在独立的对象中，以使它们可以各自独立修改和复用
    - ■ 当一个对象改变时需要通知其他对象，而不知道有多少对象需要改变
    - ■ 当一个对象必须通知其他对象，而不能确定其他对象是谁
  - 参与者
    - ■ Subject 目标对象，知道所有的观察者；可以有任意多个观察者观察同一目标；提供注册和删除观察者的接口
    - ■ Observer 观察者，提供更新观察者的接口
    - ■ ConcreteSubject: 具体目标对象，维护观察者列表。状态在发生改变时，通知所有观察者

- ■ ConcreteObserver: 实现更新接口, 使状态与目标状态一致
  - 样例(应用场景)
    - 下单后的各个渠道的通知
- 状态模式(State)
  - 概述
    - 在一个对象的状态改变时允许改变它的行为, 看起来这个对象改变了它的类
  - 适用性
    - ■ 一个对象的行为取决于它的状态, 并且在运行时根据状态改变它的行为
    - ■ 一个对象中包含庞大的多分支条件语句, 且这些分支依赖于对象的状态  
 这些状态通常由一个或多个常量枚举组成  
 State模式将每个分支条件封装成一个独立的类中, 使得可以根据具体的状态封装成具体的对象, 而该对象可以独立的变化却不依赖其他对象。
  - 参与者
    - ■ Context 上下文对象, 定义客户端所期望的接口, 维护一个 ConcreteState实例, 以保存当前状态
    - ■ State 抽象状态, 定义一个与Context状态相关的一个接口
    - ■ ConcreteStatesubclasses 具体状态子类, 各子类实现一个与 Context状态相关的一个接口
  - 样例(应用场景)
    - ■ 订单状态的切换处理
  - 与策略模式的区别
    - ■ 状态模式的着重点在于不同的状态切换做不同的事情;  
 策略模式是根据具体情况选择策略, 不进行状态切换。
    - 2.状态模式在不同的状态下事情不同, 而策略模式做的是同一件事情, 如支付功能, 支付宝, 微信支付属于不同的策略, 可以相互替代。但状态模式每个状态的功能不同, 不能相互替代。  
 状态模式各子状态类依赖Context上下文状态  
 策略模式不依赖Context,Context中调用各策略类。
- 策略模式(Strategy)
  - 概述
    - 定义一系列算法, 将它们一一封装起来, 使得可以相互替换。策略模式使算法可以独立于它的使用者变化。
  - 适用性
    - ■ 许多相关的类仅仅是"行为各异", "策略模式"提供了用多个行为的一个行为来配置一个类的方法
    - ■ 需要使用算法的不同变体
    - ■ 算法使用客户不应该知道的数据。可以使用策略模式避免暴露复杂的, 与算法相关的数据结构
    - ■ 一个类中定义了多个行为, 并且这些行为以个条件语句的形式出

现；可以将相关条件分支移入他们的Strategy类中以替换条件语句

- 参与者
  - Strategy 抽象策略接口，定义全部算法公共接口
  - ConcreteStrategy 具体抽象类，实现具体算法
  - StrategyContext 策略上下文，维护一个ConcreteStrategy对象，持有一个Strategy引用，定义一个接口让Strategy访问它的数据
- 样例(应用场景)
  - 多渠道通知
  - 生成多线路视频地址
- 模版方法(TemplateMethod)
  - 概述
    - 定义一个操作总算法的框架，而将一些步骤延迟到子类中。模版方法模式使得子类可以不改变一个算法的结构即可重新定义该算法的某些特定步骤。模版方法是基于代码复用技术
  - 适用性
    - 一次性实现算法的公共部分，将可变的行为由子类实现
    - 各个子类中公共的行为应提出在父类中实现以避免代码重复，将代码中不同的部分封装成新的操作，用新的操作的模版方法代替这些不同的代码
    - 控制子类的扩展
  - 参与者
    - AbstractClass 抽象类，定义一系列基本操作，这些操作可以是抽象的也可以是具体的。每个操作对应算法的一个步骤，在子类中可以重新定义或实现这些操作。
    - ConcreteClass 具体实现类，实现父类中声明的抽象基本操作已完成子类特定算法的步骤，也可以覆盖父类已经实现的具体基本操作。
  - 样例
    - Servlet: Servlet(Server Applet)是Java Servlet的简称，在每一个 Servlet 都必须要实现 Servlet 接口，GenericServlet 是个通用的、不特定于任何协议的Servlet，它实现了 Servlet 接口，而 HttpServlet 继承于 GenericServlet，实现了 Servlet 接口，为 Servlet 接口提供了处理HTTP协议的通用实现，所以我们定义的 Servlet 只需要继承 HttpServlet 即可。
    - Spring中spring-context-4.3.12.RELEASE-sources.jar!org\springframework\context\support\AbstractApplicationContext.java#refresh方法
- 访问者模式(Visitor)
  - 概述
    - 表示一个作用于某对象结构的各元素的操作  
它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。
    - 访问者模式是一种将数据操作和数据结构分离的设计模式。

- 适用性
  - 1.一个对象结构包含很多类对象，它们有不同的接口，而你想对这些对象实施一些依赖于其具体类的操作。
  - 2.需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而你想避免让这些操作“污染”这些对象的类。

Visitor使得你可以将相关的操作集中起来定义在一个类中。

当该对象结构被很多应用共享时，用Visitor模式让每个应用仅包含需要用到的操作。
  - 3.定义对象结构的类很少改变，但经常需要在此结构上定义新的操作。

改变对象结构类需要重定义对所有访问者的接口，这可能需要很大的代价。

如果对象结构类经常改变，那么可能还是在这些类中定义这些操作较好。
- 参与者
  - 1.Visitor

为该对象结构中ConcreteElement的每一个类声明一个Visit操作。

该操作的名字和特征标识了发送Visit请求给该访问者的那个类。

这使得访问者可以确定正被访问元素的具体类。

这样访问者就可以通过该元素的特定接口直接访问它。
  - 2.ConcreteVisitor

实现每个由Visitor声明的操作。

每个操作实现本算法的一部分，而该算法片断乃是对应于结构中对象的类。

ConcreteVisitor为该算法提供了上下文并存储它的局部状态。

这一状态常常在遍历该结构的过程中累积结果。
  - 3.Element

定义一个Accept操作，它以一个访问者为参数。
  - 4.ConcreteElement

实现Accept操作，该操作以一个访问者为参数。
  - 5.ObjectStructure

能枚举它的元素。

可以提供一个高层的接口以允许该访问者访问它的元素。

可以是一个复合或是一个集合，如一个列表或一个无序集合。
- 样例(应用场景)

## JAVA基础

---

### JVM

- JVM基础知识
  - 构成图谱
    - .Class文件

使用javap -v 指令能看到易于我们阅读的信息

      - 0xCAFEBADE (4bytes 魔术)
      - minor\_version (2bytes 副版本号)

- major\_version (2bytes 主版本号)
 

主版本号和次版本号在class文件中各占两个字节，副版本号占用第5、6两个字节，而主版本号则占用第7、8两个字节。

jdk1.1:45, jdk1.5:49, jdk1.6:50 (0x32), jdk1.7:51, jdk1.8:51

一个JVM实例只能支持特定范围内的主版本号（Mi至Mj）和0至特定范围内（0至m）的副版本号。假设一个Class文件的格式版本号为V，仅当 $Mi.0 \leq v \leq Mj.m$ 成立时，这个Class文件才可以被此Java虚拟机支持

JVM认为加载不了这个class文件，会抛出我们经常见到的"

java.lang.UnsupportedClassVersionError: Bad version number in .class file "Error 错误

javap -v Math 查看.class版本
  - constant\_pool\_count (2bytes 常量池计数器)
 

常量池计数器默认从1开始而不是从0开始，而是以1开始，即count=1时,常量池有0个常量；count=2时，常量池有1个常量；即常量池数量+1

原因：在指定class文件规范的时候，将索引#0项常量空出来是有特殊考虑的，这样当：某些数据在特定的情况下想表达“不引用任何一个常量池项”的意思时，就可以将其引用的常量的索引值设置为#0来表示。
  - cp\_info (常量池数据区,数量:constant\_pool\_count-1)
  - 字面量(Literal)
 

哪些字面量会进入常量池中？

结论：

final类型的8种基本类型的值会进入常量池。

非final类型（包括static的）的8种基本类型的值，只有double、float、long的值会进入常量池。

常量池中包含的字符串类型字面量（双引号引起来的字符串值）。

    - 文本字符
    - 被申明为final的常量值
    - 基本数据类型值
    - 其他
  - 符号引用(Symbolic Reference)
    - 类和结构的完全限定名
 

对于某个类或接口而言，其自身、父类和继承或实现的接口的信息会被直接组装成CONSTANT\_Class\_info常量池项放置到常量池中；

类中或接口中使用到了其他的类，只有在类中实际使用到了该类时，该类信息才会在常量池中有对应的CONSTANT\_Class\_info常量池项；

类中或接口中仅仅定义某种类型的变量，JDK只会将变量的类型描述信息以UTF-8字符串组成CONSTANT\_Utf8\_info常量池项放置到常量池中，上面在类中的private Date date;JDK编译器只会将表示date的数据类型的“Ljava/util/Date”字符串放置到常量池中。

Object类，在源文件中的全限定名是 java.lang.Object 。而

class文件中的全限定名是将点号替换成“/”。Object类在class文件中的全限定名是 java/lang/Object。源文件中一个类的名字，在class文件中是用全限定名表述的。

- 字段名称和描述符

各类型的描述符

对于字段的数据类型，其描述符主要有以下几种

基本数据类型 (byte、char、double、float、int、long、short、boolean)：除long 和boolean，其他基本数据类型的描述符用对应单词的大写首字母表示。long 用J 表示，boolean 用Z 表示。

void：描述符是V。

对象类型：描述符用字符 L 加上对象的全限定名表示，

如 String 类型的描述符为Ljava/lang/String。

数组类型：每增加一个维度则在对应的字段描述符前增加一个 [，如一维数组 int[] 的描述符为 [I，二维数组 String[][] 的描述符为 [[Ljava/lang/String。

- 方法名称和描述符

方法的描述符比较复杂，包括所有参数的类型列表和方法返回值。它的格式是这样的：

(参数1类型 参数2类型 参数3类型 ...)返回值类型

注意事项：

不管是参数的类型还是返回值类型，都是使用对应字符和对应字符串来表示的，并且参数列表使用小括号括起来，并且各个参数类型之间没有空格，参数列表和返回值类型之间也没有空格。

特殊方法的方法名：

首先要明确一下，这里的特殊方法是指的类的构造方法和类型初始化方法。构造方法就不用多说了，至于类型的初始化方法，对应到源码中就是静态初始化块。也就是说，静态初始化块，在class文件中是以一个方法表述的，这个方法同样有方法描述符和方法名，具体如下：

类的构造方法的方法名使用字符串 表示

静态初始化方法的方法名使用字符串 表示。

除了这两种特殊的方法外，其他普通方法的方法名，和源文件中的方法名相同。

总结：

方法和字段的描述符中，不包括字段名和方法名，字段描述符中只包括字段类型，方法描述符中只包括参数列表和返回值类型。

无论method()是静态方法还是实例方法，它的方法描述符都是相同的。尽管实例方法除了传递自身定义的参数，还需要额外传递参数this，但是这一点不是由方法描述符来表达的。参数this的传递，是由Java虚拟机实现在调用实例方法所使用的指令中实现的隐式传递。

- ■ access\_flags (2bytes 访问标志)

访问标志，access\_flags 是一种掩码标志，用于表示某个类或者接

口的访问权限及基础属性。

- ■ this\_class (2bytes 类索引)  
类索引，this\_class的值必须是对constant\_pool表中项目的一个有效索引值。constant\_pool表  
在这个索引处的项必须为CONSTANT\_Class\_info 类型常量，表示这个 Class 文件所定义的类或接口。
- ■ super\_class (2bytes 父类索引)  
父类索引，对于类来说，super\_class 的值必须为 0 或者是对 constant\_pool 表中项目的一个有效索引值。  
如果它的值不为 0，那 constant\_pool 表在这个索引处的项必须为 CONSTANT\_Class\_info 类型常量，表示这个 Class 文件所定义的类的直接父类。当前类的直接父类，以及它所有间接父类的 access\_flag 中都不能带有ACC\_FINAL 标记。  
对于接口来说，它的Class文件的super\_class项的值必须是对 constant\_pool表中项目的一个有效索引值。constant\_pool表在这个索引处的项必须为代表 java.lang.Object 的 CONSTANT\_Class\_info 类型常量。  
如果 Class 文件的 super\_class的值为 0，那这个Class文件只可能是定义的是java.lang.Object类，只有它是唯一没有父类的类。
- ■ interfaces\_count (2bytes 接口计数器)  
接口计数器，interfaces\_count的值表示当前类或接口的【直接父接口数量】
- ■ 接口数据区  
接口表，interfaces[]数组中的每个成员的值必须是一个对 constant\_pool表中项目的一个有效索引值， 它的长度为 interfaces\_count。每个成员interfaces[i] 必须为 CONSTANT\_Class\_info类型常量，其中 【 $0 \leq i < \text{interfaces\_count}$ 】。  
. 在interfaces[]数组中，成员所表示的接口顺序和对应的源代码中给定的接口顺序（从左至右）一样，即interfaces[0]对应的是源代码中最左边的接口。
- ■ fields\_count (2bytes 字段计数器)  
字段计数器，fields\_count的值表示当前 Class 文件 fields[]数组的成员个数。  
fields[]数组中每一项都是一个field\_info结构的数据项，它用于表示该类或接口声明的类字段或者实例字段。
- ■ field\_info (字段信息数据区)  
字段表，fields[]数组中的每个成员都必须是一个fields\_info结构的数据项，用于表示当前类或接口中某个字段的完整描述。  
fields[]数组描述当前类或接口声明的所有字段，但不包括从父类或父接口继承的部分。
- ■ methods\_count (2bytes 方法计数器)  
方法计数器， methods\_count的值表示当前Class 文件 methods[]数组的成员个数。Methods[]数组中每一项都是一个 method\_info 结构的数据项。
- ■ method\_info (方法信息数据区)

方法表，methods[] 数组中的每个成员都必须是一个 method\_info 结构的数据项，用于表示当前类或接口中某个方法的完整描述。

如果某个method\_info 结构的access\_flags 项既没有设置 ACC\_NATIVE 标志也没有设置ACC\_ABSTRACT 标志，那么它所对应的方法体就应当可以被 Java 虚拟机直接从当前类加载，而不需要引用其它类。

method\_info结构可以表示类和接口中定义的所有方法，包括实例方法、类方法、实例初始化方法和类或接口初始化方法。

【methods[]数组只描述当前类或接口中声明的方法，不包括从父类或父接口继承的方法】。

- ■ attribute\_count (2bytes 属性计数器)

属性计数器，attribute\_count的值表示当前 Class 文件attributes表的成员个数。attributes表中每一项都是一个attribute\_info 结构的数据项。

- ■ attribute\_info (属性信息数据区)

属性表，attributes 表的每个项的值必须是attribute\_info结构在Java 7 规范里，Class文件结构中的attributes表的项包括下列定义的属性：InnerClasses、EnclosingMethod、Synthetic、Signature、SourceFile、SourceDebugExtension、Deprecated、RuntimeVisibleAnnotations、RuntimeInvisibleAnnotations以及BootstrapMethods属性。

对于支持 Class 文件格式版本号为 49.0 或更高的 Java 虚拟机实现，必须正确识别并读取attributes表中的Signature、RuntimeVisibleAnnotations和RuntimeInvisibleAnnotations属性。对于支持Class文件格式版本号为 51.0 或更高的 Java虚拟机实现，必须正确识别并读取 attributes表中的BootstrapMethods属性。Java 7 规范 要求任一 Java 虚拟机实现可以自动忽略 Class 文件的 attributes表中的若干（甚至全部）它不可识别的属性项。任何本规范未定义的属性不能影响Class文件的语义，只能提供附加的描述信息。

- 类加载器子系统

- 加载>Loading)

- Bootstrap Class Loader

负责加载JAVA\_HOME\lib 目录中的，或通过-Xbootclasspath参数指定路径中的，且被虚拟机认可（按文件名识别，如rt.jar）的类。由C++实现，不是ClassLoader子类

- Extension Class Loader

负责加载JAVA\_HOME\lib\ext 目录中的，或通过java.ext.dirs系统变量指定路径中的类库

- 加载过程

在加载的过程中,JVM主要做3件事情

通过一个类的全限定名来获取定义此类的二进制字节流(class 文件)在程序运行过程中,当要访问一个类时,若发现这个类尚未被加载,并满足类初始化的条件时, 就根据要被初始化的这个类的全限定名找到该类的二进制字节流,开始加载过程。



将这个字节流的静态存储结构转化为方法区的运行时数据结构。

在内存中创建一个该类的java.lang. Class对象,作为方法区该类的各种数据的访问入口

- 加载源

加载源:

zip包: jar , war, ear 等

其他文件生成: JSP生成的class

数据库中: 将二进制字节流存储至数据库中,然后在加载时从数据库中读取.有些中间件会这么做,用来实现代码在集群间分发网络

运行时计算生成: 动态代理技术,用

ProxyGenerator.generateProxyClass为特定接口生成形式为"\*\$Proxy"的代理类的二进制字节流

- 类和数组加载的区别

数组类和非数组类的类加载是不同的, 具体情况如下:

非数组类: 是由类加载器来完成。

数组类: 数组类本身不通过类加载器创建, 它是由java虚拟机直接创建, 但数组类与类加载器有很密切的关系, 因为数组类的元素类型最终要靠类加载器创建。

- 加载过程注意点

- JVM规范并未给出类在方法区中存放的数据结构

- JVM规范并没有指定Class对象存放的位置

HotSpot将Class对象存放在方法区

类实例存放在堆区

- 加载阶段和链接阶段是交叉的

类加载的过程中每个步骤的开始顺序都有严格限制,但每个步骤的结束顺序没有限制。也就是说,类加载过程中,必须按照如下顺序开始:

加载 -> 链接 -> 初始化

但结束顺序无所谓,因此由于每个步骤处理时间的长短不一就会导致有些步骤会出现交叉

- Application Class Loader

负责加载用户路径 (classpath) 上的类库。

- 类加载器

- JVM的类加载是通过ClassLoader及其子类来完成的, 类的层次关系和加载顺序:

(类加载器遵循委托层次算法[Delegation Hierarchy Algorithm])

- 加载过程中会先检查类是否被已加载, 检查顺序是自底向上, 从Custom ClassLoader到BootStrap

ClassLoader逐层检查, 只要某个classloader已加载就视为已加载此类, 保证此类在所有ClassLoader

加载一次。而加载的顺序是自顶向下, 也就是由上层来逐层尝试加载此类。

检查顺序(自底向上): CustomClassLoader->ApplicationClassLoader->ExtensionClassLoader->BootstrapClassLoader

加载顺序(自上而下): Bootstrap->Extension->Application->Custom

- 自定义类加载器
  - 继承ClassLoader
  - 重写findClass () 方法
  - 调用defineClass () 方法
- 自定义类加载器作用

- JVM自带的三个加载器只能加载指定路径下的类字节码。

- 如果某个情况下，我们需要加载应用程序之外的类文件呢？比如本地D盘下的，或者去加载网络上的某个类文件，这种情况就可以使用自定义加载器了

- 双亲委派模型

- 避免重复加载，当父亲已经加载了该类的时候，就没有必要子

ClassLoader再加载一次

- JVM在判定两个class是否相同时，不仅要判断两个类名是否相同，而且要判断是否由同一个类加载器实

例加载的

- 还要定义自己的类加载器呢？

因为Java中提供的默认ClassLoader，只加载指定目录下的jar和class，如果我们想加载其它位置的类或jar时。

- 知识点

- ClassLoader 和 Class.forName加载类区别

- ClassLoader加载字节码到内存，默认不初始化类

- Class.forName加载字节码到内存后会初始化类，即调用static代码块,初始化static属性

化static属性

- 链接(Linking)

- Verify (验证)

验证阶段比较耗时,它非常重要但不一定必要(因为对程序运行期没有影响),如果所运行的代码已经被反复使用和验证过,那么可以使用 -Xverify:none 参数关闭,以缩短类加载时间。

- 验证的目的

- 字节码校验器校验字节码是否正确

- 验证的必要性

- 验证的过程

- 文件格式验证(基于二进制字节流)

本验证阶段是基于二进制字节流进行的,只有通过本阶段验证,才被允许存到

方法区

后面的三个验证阶段都是基于方法区的存储结构进行,不会再直接操作字节

- 元数据验证(基于方法区数据结构)

- 字节码验证(基于方法区数据结构)

- 符号引用验证(基于方法区数据结构)

- Prepare (准备)

仅仅为类变量（即static修饰的字段变量）分配内存并且设置该类变量的初始值即零值，这里不包含用final修饰的static，因为final在编译的时候就会分配了（编译器的优化），同时这里也不会为实例变量

分配初始化。类变量会分配在方法区中，而实例变量是会随着对象一起分配到Java堆中。

- 分配内存并初始化0值给所有类静态变量
- Resolve (解析)
  - 所有符号内存应用被方法区(Method Area)的原始引用
- 初始化(Initialization)
- 注意点

方法是编译器自动收集类中所有类变量的赋值动作和静态语句块中的语句合并产生的,编译器收集的顺序是由语句在源文件中出现的顺序所决定的.

静态代码块只能访问到出现在静态代码块之前的变量,定义在它之后的变量,在前面的静态语句块可以赋值,但是不能访问.

实例构造器需要显式调用父类构造函数,而类的不需要调用父类的类构造函数,虚拟机会确保子类的方法执行前已经执行完毕父类的方法.因此在JVM中第一个被执行的方法的类肯定是java.lang. Object.如果一个类/接口中没有静态代码块,也没有静态成员变量的赋值操作,那么编译器就不会为此类生成方法.

接口也需要通过方法为接口中定义的静态成员变量显示初始化。接口中不能使用静态代码块,但仍然有变量初始化的赋值操作,因此接口与类一样都会生成方法.不同的是,执行接口的方法不需要先执行父接口的方法.只有当父接口中的静态成员变量被使用到时才会执行父接口的方法.

虚拟机会保证在多线程环境中一个类的方法别正确地加锁,同步.当多条线程同时去初始化一个类时, 只会有一个线程去执行该类的方法,其它线程都被阻塞等待,直到活动线程执行方法完毕.其他线程虽会被阻塞,只要有一个方法执行完,其它线程唤醒后不会再进入方法.同一个类加载器下,一个类型只会初始化一次.

- 类加载的时机

遇到 new 、 getstatic 、 putstatic 和 invokestatic 这四条指令时, 如果对应的类没有初始化, 则要对对应的类先进行初始化.这四个指令对应到我们java代码中的场景分别是:

new关键字实例化对象的时候;

读取或设置一个类的静态字段（读取被final修饰, 已在编译器把结果放入常量池的静态字段除外）;

调用类的静态方法时。

使用 java.lang.reflect 包方法时对类进行反射调用的时候。

初始化一个类的时候发现其父类还没初始化, 要先初始化其父类。

当虚拟机开始启动时, 用户需要指定一个主类, 虚拟机会先执行这个主类的初始化。

- 所有类静态变量赋初始值,执行静态代码块
- ...JVM内存结构
  - 运行时数据区
    - 方法区
    - 堆
    - 栈
    - PC寄存器(程序计数器)
    - 本地方法栈
  - 虚拟对象剖析
    - 对象创建
    - 对象访问

- 对象分配
- JIT 编译器
- JVM内存结构
  - 运行时数据区
    - 方法区
      - XX:MaxPermSize=16m
      - 为永久代的实现(PermSpace)
        - 存放类信息,
        - 常量池、
        - 字符串常量池(1.7以后转到堆区)、
        - 方法片段
      - 存放各种常量池:
        - Class类常量池
        - 运行时常量池
        - 字符串常量池(<=1.6)
        - 基本数据类型常量池
        - 及类字节码,接口/类的全局限定名
        - 字段名称和修饰符, 方法名称和修饰符
        - 所有静态变量
      - 方法区也会GC
      - 堆区(Heap Space)
    - Java堆被所有线程共享, 在Java虚拟机启动时创建。是虚拟机管理最大的一块内存。
      - 存放对象实例, Java虚拟机规范的描述是:
        - 所有的对象实例以及数组都要在堆上分配
        - Java堆是垃圾回收的主要区域, 主要采用分代回收算法。
        - 堆分类
          - <1.8
            - 新生代(Young) (Eden空间[伊甸园], From Survivor空间, To Survivor空间,S0,S1)
            - 老年代(Old)
            - 永久代, 使用堆内存, 为方法区的实现
          - 1.8
            - 新生代 (Eden空间[伊甸园], From Survivor空间, To Survivor空间) 、
            - 老年代
            - 元空间(Meta Space), 使用物理内存空间,大小受物理内存限制
              - 存储字符串常量池的字符串引用
              - Java8 中的 MetaSpace
              - 为什么取消永久代
    - 永久代经常可能出现OOM: PermSpace

- 为了融合HotSpot和JRocket,JRocket中没有永久代
  - 堆内存划分
    - 堆大小= 新生代+ 老年代。堆的大小可通过参数-Xms（堆的初始容量）、-Xmx（堆的最大容量）来指定。
    - 其中，新生代(Young)被细分为Eden 和 两个Survivor 区域，这两个Survivor 区域分别被命名为from 和to，以示区分。默认的，Eden : from : to = 8 : 1 : 1。(可以通过参数-XX:SurvivorRatio 来设定。即：Eden = 8/10 的新生代空间大小，from = to = 1/10 的新生代空间大小。
    - JVM 每次只会使用Eden 和其中的一块Survivor 区域来为对象服务，所以无论什么时候，总是有一块Survivor 区域是空闲着的
    - 新生代实际可用的内存空间为9/10 (即90%)的新生代空间
    - jdk<=1.6时，字符串常量池保存在方法区即永久代中
    - jdk>=1.7时，字符串常量池保存在堆区(字符串常量池是逻辑内部表概念，新生代,老年代是物理分层概念，实际创建对象还是在新生代，老年代中)
    - 栈区(Java虚拟机栈 Stack Space)
      - 线程私有，而且生命周期与线程相同，每个Java方法在运行的时候都会创建一个栈帧
 (Stack Frame)
        - 栈帧
          - 随着方法创建而创建，方法结束而销毁
          - 局部变量表
            - 基础数据类型的值
            - 对象的引用
          - 操作数栈
          - 动态连接
          - 方法返回地址
          - 附加信息
          - 栈异常
            - Java虚拟机规范中，对该区域规定了这两种异常情况：
              - 1. 如果线程请求的栈深度大于虚拟机所允许的深度，将会抛出StackOverflowError异常；
              - 2. 虚拟机栈可以动态拓展，当扩展时无法申请到足够的内存，就会抛出OutOfMemoryError异常。
      - PC寄存器(程序计数器)
        - 此内存区域是唯一一个在Java的虚拟机规范中没有规定任何OutOfMemoryError异常情况的区域。
      - 本地方法栈
        - 一个Native Method就是一个java调用非java代码的接口。
        - 标识符native可以与所有其它的java标识符连用，但是abstract除外。
        - native与其它java标识符连用时，其意义同非Native Method并无差别
        - 一个native method方法可以返回任何java类型，包括非基本类型，而且同样可以进行异常控制。
        - 当一个native method接收到一些非基本类型时如Object或一个整型数组时，这个方法可以访问这些非基本型的内部，但是这将使这个native方法依赖于你所访问的java类的实现
        - native method的存在并不会对其他类调用这些本地方法产生任何影响，实际上

调用这些方法的其他

类甚至不知道它所调用的是一个本地方法。JVM将控制调用本地方法的所有细节。

- 如果一个含有本地方法的类被继承，子类会继承这个本地方法并且可以用java语言重写这个方法（这个

似乎看起来有些奇怪），同样的如果一个本地方法被final标识，它被继承后不能被重写

- 执行引擎(HotSpot)

- 解释器(Interpreter)

- JIT(JustInTime)编译器(Compiler)

热点代码才会被JIT编译器编译

- 热点检测(HotSpot Detection)

运行过程中会被即时编译器编译的“热点代码”有两类:

被多次调用的方法。

被多次执行的循环体

两种情况，编译器都是以整个方法作为编译对象。这种编译方法因为编译发生在方法执行过程之中，因此形象的称之为栈上替换（On Stack Replacement, OSR），即方法栈帧还在栈上，方法就被替换了。

热点检测主要的2种方式:

基于采样的热点探测

采用这种方法的虚拟机会周期性地

检查各个线程的栈顶，如果发现某些方法经常出现在栈顶，那这个方法就是“热点方法”。这种探测方法的好处是实现简单高效，还可以很容易地获取方法调用关系（将调用堆栈展开即可），缺点是很难精确地确认一个方法的热度，容易因为受到线程阻塞或别的外界因素的影响而扰乱热点探测。

基于计数器的热点探测

采用这种方法的虚拟机会为每个方法（甚

至是代码块）建立计数器，统计方法的执行次数，如果执行次数超过一定的阈值，就认为它是“热点方法”。这种统计方法实现复杂一些，需要为每个方法建立并维护计数器，而且不能直接获取到方法的调用关系，但是它的统计结果相对更加精确严谨。

#HotSpot虚拟机中使用的是“基于计数器的热点探测”

为每个方法设置2个计数器(在确定虚拟机运行参数的前提下，这两个计数器都有一个确定的阈值，当计数器超过阈值溢出了，就会触发JIT编译)

方法调用计数器

统计方法的调用次数

回边计数器

统计方法内循环体的执行

次数

- Server Compiler(-server)

jvm启动添加 -server 设置为Server编译器

Server Compiler则是专门面向服务器端的，并为服务端的性能配置特别调整过的编译器，是一个充分优化过的高级编译器Server编译器编译

用Server Compiler 来获取更好的编译质量

- Client Compiler(-client)

jvm启动添加 -client 设置为Client编译器

Client编译器 是一个简单快速的编译器，主要关注点在于局部优化，而放弃许多耗时较长的全局优化手段。

Client编译器编译较Server编译器快，用Client Compiler获取更高的编译速度

- JIT编译器优化

- 公共子表达式的消除

- 方法内联

在使用JIT进行即时编译时，将方法调用直接使用方法体中的代码进行替换，

这就是方法内联，减少了方法调用过程中压栈与入栈的开销。同时为之后的一些优化手段提供条件。

- 逃逸分析(Escape Analysis: +XX:+DoEscapeAnalysis)

逃逸分析的基本行为就是分析对象动态作用域：当一个对象在方法中被定义后，它可能被外部方法所引用，例如作为调用参数传递到其他地方中，称为方法逃逸。

通过逃逸分析，Java Hotspot编译器能够分析出一个新的对象的引用的使用范围从而决定是否要将这个对象分配到堆上

逃逸分析包括：

- 全局变量赋值逃逸

- 方法返回值逃逸

- 实例引用发生逃逸

- 线程逃逸:赋值给类变量或可以在其他线程中访问的实例变量

通过JVM参数可指定是否开启逃逸分析:

(从jdk 1.7开始已经默认开始逃逸分析)

- XX:+DoEscapeAnalysis：表示开启逃逸分析

- XX:-DoEscapeAnalysis：表示关闭逃逸分析

- 对象的栈上内存分配

在一般情况下，对象和数组元素的内存分配是在堆内存上进行的。但是随着JIT编译器的日渐成熟，很多优化使这种分配策略并不绝对。JIT编译器就可以在编译期间根据逃逸分析的结果，来决定是否可以将对象的内存分配从堆转化为栈。

- 标量替换

标量 (Scalar) 是指一个无法再分解成更小的数据的数据。

在JIT阶段，如果经过逃逸分析，发现一个对象不会被外界访问的话，那么经过JIT优化，就会把这个对象拆解成若干个其中包含的若干个成员变量来代替。

- 同步锁消除(-XX:+EliminateLocks)

同样基于逃逸分析，当加锁的变量不会发生逃逸，是线程私有的完全没有必要加锁。在jit编译时期就可以将同步锁去掉，以减少加锁与解锁造成的资源开销。

使用参数-XX:+DoEscapeAnalysis和-XX:+EliminateLocks(锁消除必须在-server模式下)开启。

- 垃圾回收器

- 内存分配方法

1 优先在Eden分配,如果Eden空间不足虚拟机则会进行一次MinorGC

2 大对象直接接入老年代,大对象一般指的是很长的字符串或数组

3

长期存活的对象进入老年代，每个对象都有一个age，当age到达设定的年龄的时候就会进

入老年代，默认是15岁

- 指针碰撞 内存地址是连续的

- Serial和ParNew收集器

- 空闲列表 内存地址不连续

- CMS收集器和Mark-Sweep收集器

- Java 本地接口

- 本地方法库

- 面试基础问题

- String 相关

- 基本问题（字符串常量池）
  - 单独使用""引号创建的字符串都是常量，编译期就已经确定存储到String Pool中。  
使用new String("")创建的对象会存储2个对象,1个为字符串常量池对象，1个为到heap中，是运行期新创建的
- 问题升级
  - 使用只包含常量的字符串连接符如"aa"+"bb"创建的也是常量，编译期就能确定已经存储到String Pool中  
使用包含变量的字符串连接符如"aa"+s创建的对象是运行期才创建的，存储到heap中  
运行期调用String的intern()方法可以向String Pool中动态添加对象。
- 类型问题
- final 问题
- 值传递问题
- 字符串常量池(String Pool)  
C++维护的 StringTable  
<=jdk1.6 大小为:1009

=1.7 大小默认为:60013

可配置: -XX:StringTableSize=66666

(<=1.6时字符串常量池在方法区即永久代,>=1.7时字符串常量池在堆区)

字符串常量池是逻辑内部表概念(实际对象创建还是在年轻代,年老代)

堆区年轻代,年老代是物理分层概念

- String a = new String("a")

会在字符串常量池保存a

然后在堆区(Heap)创建 字符串a对象,并返回a对象地址

- String a = "a"

会检查字符串常量池是否存在"a"，若不存在，则创建

存在则直接返回地址

- String a = new String("a").intern()

同 String a = "a" 会从字符串常量池获取/保存字符串

- String a = "a";

String b = "b";

String c = a+b;

c=="ab" false

c只在堆区创建新String对象

当 c.intern() 调用后会在字符串常量池创建对象

- 增强For循环实现原理:

增强for循环使用对象的.iterator()方法进行普通for循环处理

必须支持Iterator的方法才能进行增强for循环处理

- List list = new ArrayList();

for (Iterator localIterator = list.iterator(); localIterator.hasNext(); ) {



## ■ 基本数据类型自动转换

- long(8bytes)向上转换为float(4字节)原因:

- $$V = (-1)^S * M * 2^E$$

- float类型又称为单精度浮点类型，在 IEEE 754-2008 中是这样定义它的结构的：

### 实数转二进制float类型的方法:

- 其中整数部分始终是1

float的指数位有8位，而double的指数位有11位，分布如下：

1bit (符号位)

8bits (指数位)

23bits (尾数位)

double:

1bit (符号位)

11 bits (指数位)

52bits (尾数位)

- 范围:

value of floating-point = significand  $\times$  base  $^{\text{exponent}}$ , with sign --- F.1

译为中文表达即为：

(浮点) 数值 = 尾数  $\times$  底数<sup>指数</sup>, (附加正负号) ----- F.2

于是，float的指数范围为-127~128，而double的指数范围为-1023~1024，并且指数位

是按补码的形式来划分的。其中负指数决定了浮点数所能表达的绝对值最小的数；而正指数决定了浮点数所能表达的绝对值最大的数，也即决定了浮点数的取值范围。  
float的范围为 $-2^{128} \sim +2^{128}$ ，也即 $-3.40E+38 \sim +3.40E+38$ ；double的范围为 $-2^{1024} \sim +2^{1024}$ ，也即 $-1.79E+308 \sim +1.79E+308$ 。

- 精度:

float和double的精度是由尾数的位数来决定的。浮点数在内存中是按科学计数法来存储的，其整数部分始终是一个隐含着的“1”，由于它是不变的，故不能对精度造成影响。

float:  $2^{23} = 8388608$ ，一共七位，这意味着最多能有7位有效数字，但绝对能保证的为6位，也即float的精度为6~7位有效数字；

double:  $2^{52} = 4503599627370496$ ，一共16位，同理，double的精度为15~16位。

单精度类型(float)和双精度类型(double)存储；

#float和double的精度是指,科学计数法表示下小数点后,E前的位数

- <https://study.com/academy/lesson/java-floating-point-numbers.html#:~:text=A%20float%20data%20type%20in,it%20will%20save%20as%20double>.

Java中的float数据类型存储一个精度为6-7位数的十进制值。因此，例如，可以将12.12345保存为浮点数，但不能将12.123456789保存为浮点数。

如:c:123.123456789123f:123.12346

d:1234567.1234567f:1234567.1

在用Java表示float数据类型时，我们应该在数据类型的末尾附加字母f。否则将另存为两倍。

Java中的float的默认值为0.0f。浮点数据类型用于要节省内存且计算不需要超过6或7位精度的情况。

- java.lang.Double:

POSITIVE\_INFINITY = 1.0 / 0.0

NEGATIVE\_INFINITY = -1.0 / 0.0

NaN = 0.0d / 0.0

java.lang.Float:

POSITIVE\_INFINITY = 1.0f / 0.0f

NEGATIVE\_INFINITY = -1.0f / 0.0f

NaN = 0.0f / 0.0f

- java.math.BigDecimal:

应优先使用new BigDecimal(String)构造函数,因为以float,和double为参数的构造方法,本身传入的float,double值的精度不准确;

BigDecimal使用 long intCompact和 int scale来控制精度,intCompact存储无小数点的整数,scale存储小数点右移的长度;

如1.001在BigDecimal中为intCompact=1001,scale=3;

-100在BigDecimal中intCompact=-100,scale=0;

BigDecimal中包含stringCache，因此创建BigDecimal对象时，优先转换成String类型，#BigDecimal最大支持18位(不含符号,含小数点)的数，

- 加减乘除:

加法: long 类型 +

减法: 转成加法,加负数

乘法: long类型 \*, 进位超界判断

除法: long 类型 / , 小数位保留判断

- 十进制转换为二进制数:

1.十进制整数转换为二进制整数计算的方法：十进制整数转换为二进制整数采用"除2取

余，逆序排列"法。具体做法是：用2整除十进制整数，可以得到一个商和余数；再用2去除商，又会得到一个商和余数，如此进行，直到商为小于1时为止。

如：将0.125换算为二进制，结果为：将0.125换算为二进制（0.001）<sub>2</sub>。

分析：第一步，将0.125乘以2，得0.25，则整数部分为0，小数部分为0.25。

第二步，将小数部分0.25乘以2，得0.5，则整数部分为0，小数部分为0.5。

第三步，将小数部分0.5乘以2，得1.0，则整数部分为1，小数部分为0.0。

第四步，读数，从第一位读起，读到最后一位，即为0.001。

2.十进制的小数转换成二进制可以乘2取整法，即将小数部分乘以2，然后取整数部分，剩下的小数部分继续乘以2，然后取整数部分，剩下的小数部分又乘以2，一直取到小数部分为零为止；

如果永远不能为零，就同十进制数的四舍五入一样，按照要求保留多少位小数时，就根据后面一位是0还是1，取舍，如果是零，舍掉，如果是1，向入一位。换句话说就是0舍1入。读数要从前面的整数读到后面的整数。

#0b开头表示二进制数(b不区分大小写)

10进制数不以0开头

- 八进制转二进制：

8进制的每一位数转换为3位的二进制数；

0o开头表示8进制数(o不区分大小写,或省略)

- 十六进制转二进制(10-16对应A-F)：

16进制的每一位转换为4位二进制数；

0x开头表示16进制数(x不区分大小写)

- 找不到以上基础类型数据时,再找该基本数据类型对应的包装类型

- 不同的基础类型转换只能转换为基础数据类型，不能转换为包装类型
- 包装类型在没有对应包装类型(即拆箱后的基础类型)的方法时，可以转换为更高级的基础数据类型，如：Integer 转为 long ,Float 转为 double,如：  
g(new Float(.1f)); 可调用到static void g(double d)
- 包装类型不能自动转换
- 垃圾回收
  - GC概念
    - 对象流转过程
      - 向Eden区申请内存,若空间够则分配内存  
若不够则进行一次YoungGC
  - YGC后再次向Eden区申请内存，若够，则分配内存，  
若不够则判断老年代是否够，够则分配内存，  
若老年代也不够，则进行1次FullGC
  - FullGC后，再次向老年代申请内存，若够，则分配内存  
若不够，则抛出OOM
    - 回收算法
      - 引用计数器
        - 对像有地方引用就+1，失效就-1，当计数器为0时就回收
          - Python和ActionScript3使用
          - 效率低(一直在加减计算)

循环无法确定(循环引用无法确定)

- 根路径搜索
  - 以GC roots为根搜索可达对象，如果对象之间循环引用没有根引用则为不可达对象
- 标记-清除
- 复制算法（新生代）
- 标记-整理（老年代）
- 其他
  - 比较
  - 应用场景
  - 可触性
  - Stop-The-World
    - 全局暂停  
所有java代码停止，native代码可以执行，但不能和JVM交互
    - 基本上由GC清理造成  
Dump线程，死锁检查，堆Dump
- 回收器汇总
  - 年轻代GC和年老代并行使用
    - 年轻代
      - Serial(串行):串行回收器，所有回收都是1个线程完成的，回收的时候Stop-The-World, client 默认回收器
      - ParNew: 并行回收器，多线程工作，回收时Stop-The-World server 默认回收器
      - Parallel Scavenge: 并发回收器，  
吞吐量=程序运行时间/(JVM执行回收时间+程序运行时间)
    - 年老代
      - Serial Old,Parallel Old,CMS
      - CMS不能和Parallel Scavenge:同时使用
    - 新GC
      - G1,ZGC
- GC Roots
  - 跨代引用
  - 种类
  - 对象
- JVM 优化
  - JVM调优基本概念
    - JVM将内存区分为堆区(Heap),栈区(Stack)和方法区(Perm), 堆区存放实际的对象实例，需要GC。  
栈区为每个方法执行开辟的栈表，生命周期与线程相同，存放局部变量表,操作数栈,方法返回值地址等数据

- JVM内存模型为: 年轻代(Eden,S0,S1),老年代(OldMemory),永久代(Perm,或(java8+)元数据区),  
垃圾回收GC针对于堆区年轻带和老年代, 分为Minor GC(YGC,年轻代GC),Major GC(老年代GC,FullGC)  
对象刚创建时会存放到Eden区, 在发生YGC时, 没有引用的对象会被清除内存, 有引用的对象则被标记复制到S0(第一次YGC),S1(第二次YGC),当第在S1的对象被标记15次(默认)时, 会被复制到老年区(OldMemory)。  
对象创建时, 申请内存流程: 对象创建时先在Edon区申请空间, 若空间不足, 则进行一次MinorGC,若空间还不足则向老年代申请空间, 老年代空间不足时, 进行一次FullGC,之后空间还不足时,抛出OOM,反之存放到老年区或Eden区。  
老年区存放大对象或大数组, 及生存时间长的对象。  
YGC,FullGC都会Stop-The-World,使程序停止事务处理, 只有GC进程允许进行垃圾回收。  
从JVM调优角度来看, 应尽量避免YGC和FullGC, 或使YGC和FullGC的时间足够短。
- JVM调优, 主要目的是减少GC频率和FullGC次数
  - Full GC 对整个堆内存进行整理, 包括Young,Old,Perm. FullGC过程比较慢, 应减少FullGC次数
  - 导致FullGC的原因
    - OldMemory 老年代内存不足  
调优时尽量让对象在YGC时被回收,让对象在新生代存货一段时间, 及不要创建过大的对象或数组,  
避免直接在老年代创建对象
    - Perm Generation老年代空间不足  
增大老年代空间,避免太多静态对象, 控制好新生代老年代的内存比例
    - System.gc()被显示调用  
垃圾回收不要手动触发, 尽量依靠jvm自生机制  
对JVM调优过程中, 大部分工作是对FullGC的调节
- JVM调优方法和步骤
  - 监控GC状态  
通过jvisualvm的visual gc插件查看GC状态, 分析JVM参数设置, 根据实际各区域内存划分和GC执行时间,判断是否需要优化
  - 生成堆dump文件  
通过JMX的MBean生成当前Heap信息的hprof文件  
或使用jmap生成
  - 3.分析堆dump文件  
通过jvisualvm ,jprofiler, Mat分析dump的hprof文件
  - 分析结果, 判断是否需要优化  
如果各项参数设置合理, 系统没有超时日志出现, GC频率不高, GC耗时不高, 则不需要GC优化, 如果GC时间超过1-3秒或频繁GC, 则需优化
  - 注: 如果满足下面的指标, 则一般不需要进行GC:  
Minor GC执行时间不到50ms;

Minor GC执行不频繁, 约10秒一次;

Full GC执行时间不到1s;

Full GC执行频率不算频繁, 不低于10分钟1次;

- ■ 调整GC和内存分配  
如果内存分配过大或过小, 或者采用的GC收集器比较慢, 则应该有效调整这些参数, 并找1台或多台机器进行优化对比,最后做出选择
- ■ 不断的分析和调整  
通过不断试验调整,找出最合适的配置参数,最后应用到所有服务器
- JVM优化参数参考
  - ■ 针对JVM的设置, 一般通过-Xms,-Xmx限定最小最大值, 为防止垃圾收集器在最小、最大之间收缩产生额外时间, 通常把最大最小设置为相同值
  - ■ 年轻代与年老代将根据默认比例(1:2)分配内存,可以通过调整二者比率-XX:NewRatio来调整大小, 也可以通过-XX:NewSize,-XXMaxNewSize设置绝对大小。  
同样, 为了防止年轻代堆收缩,通常设置-XX:NewSize=-XX:MaxNewSize(或直接设置-Xmn)
  - ■ 年轻代和年老代设置多大才算合理
    - ■ 更大的年轻代必然导致更小的年老代, 大的年轻代会延长普通GC周期, 但会增加每次GC时间; 小的年老代会导致更加频繁的FullGC
    - ■ 更小的年前的必然导致更大的年老代, 小的年轻代会导致频繁YGC, 但每次GC时间很短, 大的年老代会减少FullGC的频率
  - 应根据应用程序对象生命周期的分布情况设置内存大小:  
如果应用存在大量临时对象, 应该选择更大年轻代  
如果存在相对较多的持久对象, 年老代应该适当增大。  
抉择时应根据以下2点:
    - ■ 本着FullGC尽量少的原则, 让年老代尽量缓存常用对象
    - ■ 通过观察应用一段时间, 查看峰值时年老代占多少内存, 在不影响FullGC的前提下, 根据实际情况加大年老代
  - ■ 在配置比较好的机器上(多核,大内存),可以为老年代选择并行收集算法:-XX:+UseParallelOldGC
  - ■ 线程堆栈的设置: 每个线程默认开启1M的堆栈大小, 用于存放栈帧、调用参数、局部变量表、操作数栈等, 对于大部分应用来说默认值太大, 一般256K就够用;  
理论上内存不变的情况下, 减少每个线程的堆栈, 可以产生更多的线程, 但实际还受限于操作系统
- JVM调优年轻代/年老代大小选择总结
  - 年轻代大小选择
    - 响应时间优先的应用
      - 年轻代尽可能的大, 直到接近系统最低响应时间  
同时减少到达老年代的对象
    - 吞吐量优先的应用
      - 年轻代尽可能的大。

因为对响应时间没有要求，垃圾回收可以并行进行，一般适合8CPU以上的应用

- 年老代大小选择

- 响应时间优先的应用

- 年老代使用并发收集器。

- 但需要小心设置，一般考虑并发会话和会话持续时间等一些参数。

- 如果堆设置小，可能会造成内存碎片。高回收频率及应用暂停而使用传统标记清除方式;

- 如果堆设置大了，则需要较长的收集时间。

- 最优化的方法，一般需要参考:

- 并发垃圾收集信息

- 持久代并发收集次数

- 传统GC信息

- 花在年轻代和年老代回收的时间比例

- 减少年轻代和年老代花费的时间，一般会提高应用效率

- 吞吐量优先的应用

- 一般吞吐量优先的应用都有一个很大的年轻代和一个较小的年老代。原因是，这样可以尽可能回收掉大部分短期对象，减少中期的对象，而年老代尽存放长期存活对象。

- 参数选择

- 堆大小(新生代+老年代+永久代(<1.8))

- Xmx256M 最大堆内存大小

- Xms256M 最小堆内存大小

- Xmn128M 年轻代占堆内存大小,剩余为老年代，若配置了-Xmn则表示-XX:NewSize=-XX:MaxNewSize

- -Xmx4096m -Xms4096m 大小相等

- 栈大小

- Xss256K 每个线程的栈大小,默认1M

- 年轻代大小

- XX:NewSize 新生代初始内存大小,应小于-Xms

- XX:MaxNewSize 新生代最大内存大小,应小于-Xmx

- 官方推荐3/8，年轻代大小不要超过老年代

- 年老代和新生代比例

- XX:NewRatio=2

- 年老代和新生代比例

- 如:NewRatio=2 表示年老代是新生代的2倍

- 年老代占堆区2/3,新生代占1/3

- 新生代Eden和Survivor比例

- XX:SurvivorRatio=1

- 新生代里Eden和2个survivor的空间大小比例。

- XX:SurvivorRatio=1即比例为1:1:1,即S0/S1/Eden大小相等。

- JDK默认为8即S0:S1:Eden比例为1:1:8, Eden占新生代大小的8/10份。

- 设置新生代对象垃圾回收标记的最大年龄  
-XX:MaxTenuringThreshold=0
  - 默认为 15
- 方法区大小(1.8-),可能会进行GC  
-XX:PermSize=10M 方法区初始化内存大小  
-XX:MaxPermSize=10M 方法区最大内存带下
- 元数据区(>=1.8,MetaSpace),可能会进行GC  
-XX:MetaspaceSize=10M 元数据区初始化内存大小  
-XX:MaxMetaspaceSize=10M 元数据区最大内存大小
- JVM(HotSpot)7种垃圾回收器  
7种垃圾收集器作用于不同的分代，如果两个收集器之间存在连续，就说明他们可以搭配使用。  
从JDK1.3到现在，从Serial收集器-》Parallel收集器-》CMS-》G1，用户线程停顿时间不断缩短，
  - -XX:+UseSerialGC:设置年轻代为串行收集器  
新生代采用复制算法,暂停所有用户线程  
老年代采用标记-整理算法,暂停所有用户线程  
可以和CMS,SerialOld搭配使用
    - Serial收集器是最基本、发展历史最悠久的收集器，曾是（JDK1.3.1之前）虚拟机新生代收集的唯一选择。  
Serial收集器是一个单线程的收集器。“单线程”的意义不仅仅是它只会使用一个CPU或一条收集器线程去完成垃圾收集工作，更重要的是它在垃圾收集的时候，必须暂停其他所有工作的线程，直到它收集结束。  
Serial收集器是HotSpot虚拟机运行在Client模式下的默认新生代收集器。  
Serial收集器具有简单而高效，由于没有线程交互的开销，可以获得最高的单线程收集效率（在单个CPU环境中）。
  - -XX:+UseParNewGC:设置年轻代为并行收集。  
可与CMS收集同时使用。  
JDK5.0以上，JVM会根据系统配置自行设置，所以无需再设置此值。
    - ParNew收集器是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数、收集算法、Stop The World、对象分配规则、回收策略等都与Serial收集器一样。  
ParNew收集器是许多运行在Server模式下的虚拟机首选的新生代收集器，其中一个原因是，除了Serial收集器之外，目前只有ParNew收集器能与CMS收集器配合工作。  
"-XX:+UseConcMarkSweepGC": 指定使用CMS后，会默认使用ParNew作为新生代收集器。  
"-XX:+UseParNewGC": 强制指定使用ParNew。
  - -XX:+UseParallelGC:设置年轻代为并行收集器  
Parallel Scavenge JDK1.8默认年轻代收集器  
多线程,复制算法  
只能与SerialOld和ParallelOld搭配使用



- Parallel Scavenge收集器是一个新生代收集器，使用复制算法，且是并行的多线程收集器。

Parallel Scavenge收集器关注点是达到一个可控制的吞吐量（吞吐量 = 运行用户代码时间 / （运行用户代码时间 + 垃圾收集时间）），而其他收集器关注点在尽可能的缩短垃圾收集时用户线程的停顿时间。

Parallel Scavenge收集器提供了两个参数来用于精确控制吞吐量，一是控制最大垃圾收集停顿时间的

-XX:MaxGCPauseMillis参数，二是控制吞吐量大小的

-XX:GCTimeRatio参数；

- -XX:MaxGCPauseMillis=n 参数允许的值是一个大于0的毫秒数，收集器将尽可能的保证内存垃圾回收花费的时间不超过设定的值（但是，并不是越小越好，GC停顿时间缩短是以牺牲吞吐量和新生代空间来换取的，如果设置的值太小，将会导致频繁GC，这样虽然GC停顿时间下来了，但是吞吐量也下来了）。

-XX:GCTimeRatio=n 参数的值是一个大于0且小于100的整数，也就是垃圾收集时间占总时间的比率，默认值是99，就是允许最大1%

（即 $1/(1+99)$ ）的垃圾收集时间,公式为 $1/(1+n)$ 。

-XX:UseAdaptiveSizePolicy 自适应大小策略，如果这个参数打开之后，虚拟机会根据当前系统运行情况收集监控信息，动态调整新生代的比例、老年大小等细节参数，以提供最合适的停顿时间或最大的吞吐量，这种调节方式称为GC自适应的调节策略。

-XX:ParallelGCThreads=n:设置并行收集器收集时使用的CPU数。并行收集线程数。

- Serial Old收集器(jdk1.5之前与ParallelScavenge搭配)

单线程,标记-整理算法

- Serial Old收集器是Serial收集器的老年代版本，他同样是一个单线程收集器，使用" 标记-整理" 算法。

Serial Old收集器主要用于Client模式下的虚拟机使用。

Server模式下的两大用途：一、在JDK1.5及之前的版本与Parallel Scavenge收集器搭配使用；二、作为CMS收集器的后备方案，在并发收集发生Conturrent Mode Failure时使用

- -XX:+UseParalledlOldGC:设置并行年老代收集器

多线程,标记-整理算法

Paraller Old收集器(1.6+)

- Parallel Old收集器是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。

在JDK1.6中才出现。

- -XX:+UseConcMarkSweepGC:设置年老代为并发收集器

多线程,标记-清除算法

- CMS收集器是一种以获取最短回收停顿时间为目标的收集器。

目前很大一部分的Java应用集中在互联网或者B/S系统的服务端上。

CMS收集器是基于“标记-清除”算法实现，它的整个运行过程可以分为：初始登记（标记一下GC Roots能直接关联到的对象，这个过程速度很快）、并发标记（进行GCRoots Tracing的过程）、重

新标记（修正并发标记期间因用户线程继续运作而导致标记产生变动的那一部分对象的标记记录，速度稍慢）、并发清除（清除死亡的对象）4个步骤；其中，初始标记和重新标记仍然需要“Stop The World”。

CMS收集器运行的整个过程中，最耗费时间的并发标记和并发清除过程收集器线程和用户线程是一起工作的，所以总体来说，CMS收集器的内存回收过程是与用户线程一起并发执行的。

优点：并发收集、低停顿。

缺点：一：CMS收集器对CPU资源非常敏感。虽然在两个并发阶段不会导致用户线程停顿，但是会因为占用了一部分线程而导致应用程序变慢，总吞吐量下降。CMS默认启动的回收线程数是（CPU数量+3）/4。

二：CMS收集器无法处理浮动垃圾，可能出现“Conturrent Mode Failure”失败而导致另一次Full GC产生。由于CMS并发清除阶段用户线程还在运行，伴随着程序还在产生新的垃圾，这一部分垃圾出现在标记之后，CMS无法在当次收集中处理掉它们，只能留到下次再清理，这一部分垃圾称为“浮动垃圾”。也正是由于在垃圾收集阶段用户线程还在运行，那么也就需要预留有足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等待老年代填满之后再行收集，需要预留一部分空间给并发收集时用户程序使用。可以通过“-XX:CMSInitiatingOccupancyFraction”参数设置老年代内存使用达到多少时启动收集。

三：由于CMS收集器是一个基于“标记-清除”算法的收集器，那么意味着收集结束会产生大量碎片，有时候往往还有很多内存未使用，可是没有一块连续的空间来分配一个对象，导致不得不提前触发一次Full GC。CMS收集器提供了一个“-XX:UseCMSCompactAtFullCollection”参数（默认是开启的）用于在CMS收集器顶不住要FullGC时开启内存碎片整理（内存碎片整理意味着无法并发执行不得不停顿用户线程）。参数“-XX:CMSFullGCsBeforeCompaction”来设置执行多少次不压缩的Full GC后，跟着来一次带压缩的（默认值是0，意味着每次进入Full GC时都进行碎片整理）。

#### ■ CMS并发收集器配置:

-XX:+UseCMSCompactAtFullCollection：使用并发收集器时，开启对老年代的压缩,默认开启。

-XX:CMSFullGCsBeforeCompaction=0：上面配置开启的情况下，这里设置多少次Full GC后，对老年代进行压缩,默认为0

#### ■ -XX:+UseG1GC :设置G1垃圾回收器(1.7+,1.9的默认垃圾回收器)

G1将新生代,老年代的物理划分取消了,取而代之的是G1将堆分成了若干区域(Region),它仍然属于分代回收器。

不过，这些区域的一部分包含新生代，新生代的垃圾收集依然采用暂停所有应用线程的方式，将存活对象拷贝到老年代或者Survivor空间。老年代也分成很多区域，G1收集器通过将对象从一个区域复制到另外一个区域，完成了清理工作。这就意味着，在正常的处理过程中，G1完成了堆的压缩（至少是部分堆的压缩），这样也就不会有cms内存碎片问题的

存在了。

- G1的堆分区(Region)中包括Eden,Servivor,Old,Humongous 4个部分
- 在G1中, 还有一种特殊的区域, 叫Humongous区域。如果一个对象占用的空间超过了分区容量50%以上, G1收集器就认为这是一个巨型对象。这些巨型对象, 默认直接会被分配在年老代, 但是如果它是一个短期存在的巨型对象, 就会对垃圾收集器造成负面影响。为了解决这个问题, G1划分了一个Humongous区, 它用来专门存放巨型对象。如果一个H区装不下一个巨型对象, 那么G1会寻找连续的H分区来存储。为了能找到连续的H区, 有时候不得不启动Full GC。
- G1收集器是当今收集器技术发展的最前沿成果之一;  
相比其它收集器, 具有如下特点:
  - 1、并行与并发: G1能够重发利用多CPU、多核环境下的优势, 使用多个CPU来缩短Stop-The-World停顿时间。
  - 2、分代收集: 与其他收集器一样, 分代概念在G1中依然存在。
  - 3、空间整合: 与CMS的“标记-清理”算法不同, G1从整体来看是基于“标记-整理”来实现的收集器, 从局部(两个Region之间)上来看是基于“复制”算法实现的, 这两种算法都意味着G1运作期间不会产生内存空间碎片, 收集后能够提供整体的可用内存。
  - 4、可预测停顿: G1除了追求低停顿之外, 还能建立可预测的停顿时间模型, 能让使用者明确指定在一个长度为M毫秒的时间片段内, 消耗在垃圾收集上的时间不得超过N毫秒。
- G1收集器的运作大致可分为:
  - 1、初始标记: 需要停顿, 耗时短;
  - 2、并发标记:
  - 3、最终标记: 需要停顿, 可并发执行;
  - 4、筛选标记:
- 并行 (Parallel): 指多条垃圾收集线程并行工作, 但此时用户线程仍然处于等待状态。  
并发 (Concurrent): 指用户线程与垃圾收集线程同时执行 (但不一定是并行, 可能是交替执行), 用户线程继续工作, 而垃圾收集程序运行在另一个CPU上。
- 垃圾回收统计信息
  - XX:+PrintGC
  - XX:+PrintGCDetails
    - -XX:+PrintGC 输出GC日志
    - XX:+PrintGCDetails 输出GC的详细日志
    - XX:+PrintGCTimeStamps 输出GC的时间戳 (以基准时间的形式)
    - XX:+PrintGCDateStamps 输出GC的时间戳 (以日期的形式, 如 2013-05-04T21:53:59.234+0800)
    - XX:+PrintHeapAtGC 在进行GC的前后打印出堆的信息
    - XX:+PrintGCApplicationStoppedTime // 输出GC造成应用暂停的时间
    - Xloggc:../logs/gc.log 日志文件的输出路径

- 字符串常量池大小(数量)  
Number of buckets in the interned String table  
-XX:StringTableSize=66666  
≤jdk1.6 默认大小为:1009

=1.7 大小默认为:60013

- 业务场景考虑
  - 高频业务处理
  - 定时任务
  - 服务类型
- 软硬件环境
  - 机器配置
  - 关联服务
- 常用JVM命令
  - jvisualvm(jdk1.6+)
    - 查看java进程各种状态  
(VisualGC插件)  
S0C 是指: Survivor0区的分配空间  
S0U 是指: Survivor1区的已经使用的空间  
EC是指:Eden区所使用的空间  
EU是指: Eden区当前使用的空间  
OC是指: 老年代分配的空间  
OU是指: 老年代当前使用的空间  
PC是指: 持久待分配的空间  
PU是指: 持久待当前使用的空间  
YGC是指: 年轻代发生的次数, 这里是354次  
YGCT是指: 年轻代发送的总时长, 这里是54.272秒, 因此每次年轻代发生GC, 即平均每次stop-the-world的时长为54.272/354=0.153秒。  
FGC是指: 年老代回收的次数, 或者成为FullGC的次数。  
FGCT是指: 年老代发生回收的总时长。  
GCT是指: 包括年轻代YGC及年老代FGC的总时间。
  - jconsole(jdk1.5+)
    - 查看java进程各种状态
  - jps(JavaVirtualMachineProcessStatusTool)
    - 查看java进程和PID
  - jstack
    - 查看java进程堆栈信息
  - jmap
    - Java内存分析工具, 可查看heap信息和导出heap数据  
jmap -heap PID 查看当前应用Heap情况
  - iostat
  - vmstat
  - tcpdump

- 配置开启JMX

- -Djava.rmi.server.hostname=192.168.32.177
- Dcom.sun.management.jmxremote
- Dcom.sun.management.jmxremote.rmi.port=1199
- Dcom.sun.management.jmxremote.port=1199
- Dcom.sun.management.jmxremote.authenticate=false
- Dcom.sun.management.jmxremote.ssl=false
- -Dcom.sun.management.jmxremote.port: 这个是配置远程connection的端口号的, 要确定这个端口没有被占用
- Dcom.sun.management.jmxremote.ssl=false
- Dcom.sun.management.jmxremote.authenticate=false: 这两个是固定配置, 是JMX的远程服务权限的
- Djava.rmi.server.hostname: 这个是配置server的IP的, 要使用server的IP最好在机器上先用hostname -i看一下IP是不是机器本身的IP, 如果是127.0.0.1的话要改一下, 否则远程的时候连不上。

- JMM 与同步

- 并发编程模型

(2个关键问题:

线程之间如何通讯及线程之间如何同步)

- 通讯

- 共享内存模型

- 线程之间共享程序的公共状态

线程之间通过写-读内存中的公共状态来隐式通讯

- 消息传递模型

- 线程之间没有公共状态

线程之间必须通过明确的发送消息来显示进行通讯

- 同步

指程序用于控制不同线程之间操作发生的相对顺序的机制

- 共享内存模型

- 同步显示进行, 必须制定某方法或某段代码在线程之间互斥执行

- 消息传递模型

- 消息的发送必须在消息的接收之前, 所以同步是隐式进行的

- Java的并发采用的是共享内存模型

- 线程间的通讯是隐式的

- JMM(Java内存模型)

基于CAS

JSR-133(JDK5)

<https://blog.csdn.net/w372426096/article/details/80898407>

JMM(Java Memory Model) JAVA内存模型主要目标是定义程序中的变量(指实例字段、静态字段等, 不包含局部变量和方法参数, 应为后2种为线程私有)在虚拟机中存储到内存与从内存读取出来的规则细节, Java 内存模型规定所有变量都存储在主内存中, 每条

线程还有自己的工作内存，工作内存保存了该线程使用到的变量到主内存副本拷贝，线程对变量的所有操作（读取、赋值）都必须在自己的工作内存中进行而不能直接读写主内存的变量，不同线程之间无法相互直接访问对方工作内存中的变量，线程间变量值的传递均需要在主内存来完成。

Java JMM 内存模型是围绕并发编程中原子性、可见性、有序性三个特征来建立的

- Java中，所有实例域、静态域、数组元素存储在堆内存中，堆内存存在线程之间共享。JMM中所指的变量即为堆内存共享变量。

-

Java线程之间通信通过JMM控制，

JMM决定一个线程对共享变量的写入

- 主内存

- 硬件的物理内存

- 工作内存

- 寄存器和高速缓存

- 工作内存中的变量是主内存中的一份拷贝

- 线程操作,每个线程都有各自的工作内存，互相不可访问

都是读写工作内存，然后通过JMM控制器刷新到主内存

- 所有共享变量都在主内存区创建，线程读取共享变量时，现在本地内存中创建主内存共享变量副本，然后再读取副本内容；修改共享变量时，先写入本地内存，然后由JMM控制器刷新到主内存中

- 重排序

- 编译器重排序

- 编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序

- 处理器重排序

- 指令重排序

- 内存重排序

- JAVA重排序过程: JAVA源码->编译器重排序->指令级并行重排序->内存重排序->最终执行的指令序列

- 内存屏障指令(memory barriers)

- as-if-serial语义

- 不管怎么重排序，同一线程内程序的执行结果不能被改变

编译器，runtime,处理器都必须遵守as-if-serial语义

- 重排序对多线程的影响

- 多线程下重排序可能会影响执行结果

- 顺序一致性内存模型

- 一个线程中所有操作必须按照程序的顺序执行

- (不管是否同步)所有线程都只能看到一个单一的操作顺序执行。在顺序一致性内存模型中，每个操作都必须原子执行且立刻对所有线程可见

- happens-before (先行发生规则)

阐述操作之间的内存可见性

Happen-Before的规则有以下几条

程序次序规则（Program Order Rule）：在一个线程内，程序的执行规则跟程序的书写规则是一致的，从上往下执行。

管程锁定规则（Monitor Lock Rule）：一个Unlock的操作肯定先于下一次Lock的操作。这里必须是同一个锁。同理我们可以认为在synchronized同步同一个锁的时候，锁内先行执行的代码，对后续同步该锁的线程来说是完全可见的。

volatile变量规则 (volatile Variable Rule) : 对同一个volatile的变量, 先行发生的写操作, 肯定早于后续发生的读操作

线程启动规则 (Thread Start Rule) : Thread对象的start()方法先行发生于此线程的没一个动作

线程中止规则 (Thread Termination Rule) : Thread对象的中止检测 (如: Thread.join(), Thread.isAlive()等) 操作, 必行晚于线程中所有操作

线程中断规则 (Thread Interruption Rule) : 对线程的interruption () 调用, 先于被调用的线程检测中断事件(Thread.interrupted())的发生

对象中止规则 (Finalizer Rule) : 一个对象的初始化方法先于一个方法执行Finalizer()方法

传递性 (Transitivity) : 如果操作A先于操作B、操作B先于操作C,则操作A先于操作C

#### - 1. 程序顺序规则

- 一个线程中的每个操作, 都happens-before于该线程中任意的后续操作

#### - 2. 监视器锁(管程锁定)规则

- 对一个监视器的解锁, happens-before于对同一个监视器的加锁

#### - 3. volatile变量规则

- 对一个volatile域的写, happens-before于后续任意对该volatile域的读

#### ■ ■ 线程启动规则

- 假定线程A在执行过程中, 通过执行ThreadB.start()来启动线程B, 那么线程A对共享变量的修改在接下来线程B开始执行后确保对线程B可见。  
start 先于 run方法

#### ■ ■ 线程终止规则

- 假定线程A在运行的过程中, 通过制定ThreadB.join()等待线程B终止, 那么线程B在终止之前对共享变量的修改在线程A等待返回后可见。

#### ■ ■ 线程中断规则

- interrupt操作,必须发生在捕获该动作之前

#### ■ ■ 传递规则

- 如果A happens-before 于 B,B happens-before于C, 则A一定 happens-before于C

#### ■ ■ 对象终结规则

- 初始化必须发生在finalize之前

#### ■ 同步

可保证指令不进行重排序

#### ■ synchronized

- 保证可见性和原子性, 互斥

#### ■ volatile

#### ■ 只处理内存可见性, 非原子性;

保证变量在线程工作内存和主内存之间的一致

原理: 内存锁, cpu缓存中存储内存地址, 修改时通知总线修改, 实现同步

线程有自己独立的快速缓存区从主内存中读取, 修改时会令其他快速缓存失效, 然后从主内存中读取

或

写volatile对象时, JMM会把该线程对应的本地内存中的共享变量值刷新到主

内存中

读取volatile对象时，JMM会把该线程对应的本地内存置为失效，然后从主内存中读取变量

- 可见性

对一个volatile的读总能看到，任意线程对该变量最后的写

- 原子性

对单个操作的volatile对象的读写具有原子性

对volatile的复合操作不具有原子性（i++之类）

- volatile内存语义实现

- JMM通过限制编译器，处理器重排序来实现volatile写/读内存语义

- volatile 变量写前任何操作不能重排序

- volatile读后任何操作不能重排序

- 第一个操作是写，第二个操作是读不能重排序

- lock

- 锁释放和获取的内存语义

- 当线程释放锁时，JMM会把该线程对应的本地内存中的共享变量刷新到主内存。

- final

- 保证可见性

- 0 Copy

- mmap

- sendfile

- JMX

- java.security.AccessController

- ```
if (System.getSecurityManager() != null) {
    AccessController.doPrivileged(new PrivilegedExceptionAction() {
    @Override
    public Object run() throws Exception {
        ReflectionUtils.makeAccessible(initMethod);
        return null;
    }
    });
}
```

- Java对象内存结构

(3部分:对象头(Header),实例数据(InstanceData),对齐填充(Padding))

- Java对象在堆上分配内存

- 对象头(Header)

(Mark Word + kclass)

- 对象自身的运行时数据(Mark Word):

hashCode,GC分代年龄,锁标志位,线程持有的锁,偏向线程ID,偏向时间戳等  
占用空间:

32位: 4byte

64位: 8byte



- 类型指针(klass):对象指向它的类元数据的指针, 虚拟机通过这个指针来确定这个对象是那个类的实例;数组需要单独保存数组大小;  
 占用空间:  
 32位: 4bytes  
 64位: 8bytes(未开启压缩), 4bytes(开启压缩)  
 数组长度信息占用空间: 4bytes
    - 从JDK 1.6 update14 开始, 64 bit JVM 正式支持了
      - XX:+UseCompressedOops 开启压缩指针, 起到节约内存占用的新参数。
      - XX:-UseCompressedOops关闭压缩指针
 如果 UseCompressedOops 是打开的, 则以下对象的指针会被压缩:
      - 所有对象的 klass 属性
      - 所有对象指针实例的属性
      - 所有对象指针数组的元素 (objArray)
  - 计算对象头大小:
    - 32位虚拟机对象头大小= Mark Word (4B) + klass(4B) = 8B
    - 64位虚拟机对象头大小= Mark Word (8B) + klass(4B,开启压缩) = 12B
  - 若为数组时,需单独保存数组长度: 4B (32bits)
- 实例数据(InstanceData)  
 ( 8 种基本类型+对象 reference)
- 8 种基本类型和 reference 大小在虚拟机上都是固定的:
 

| Primitive Type | Memory Required(bytes) |
|----------------|------------------------|
| boolean        | 1                      |
| byte           | 1                      |
| short          | 2                      |
| char           | 2                      |
| int            | 4                      |
| float          | 4                      |
| long           | 8                      |
| double         | 8                      |
| Reference      | 4                      |
- 对齐填充(Padding)  
 $0 \leq \text{padding} < 8$
- 由于虚拟机内存管理体系要求 Java 对象内存起始地址必须为 8 的整数倍, 换句话说, Java 对象大小必须为 8 的整数倍, 当对象头+实例数据大小不为 8 的整数倍时, 将会使用Padding机制进行填充, 譬如, 64 位虚拟机上 new Object() 实际大小为:
    - Mark Word (8B) + klass(4B)[开启指针压缩] = 12B
    - 但由于Padding机制, 实际占用空间为: Mark Word (8B) + klass(4B)[开启指针压缩]+Padding(4B) = 16B
- 简述new一个对象Object占多大内存(答案:16B):  
 对象在堆内存中保存,对象占用空间分为3个部分:对象头(MarkWord+类型指针),实例数据,对齐填充;  
 1.其中对象头由MarkWord(存放对象运行时一些标记,锁表记等)(占用8B大小),和类型指针(32位占用4B,64位占用8B,在JDK6 64位以后,默认开启指针压缩

(XX:+UseCompressedOops),64位占用4B)

2.实例数据: 基本数据类型分别为 boolean(1B), byte(1B), char(2B),short(2B),int(4B),long(8B),float(4B),double(8B),reference类型(32位4B,64位8B)

3.对齐填充: JVM规范中要求Java对象必须以8的倍数的地址开头,若对象大小不足8的倍数时,需要填充到大于自身的最小8的倍数,  $0 \leq \text{padding} < 8$

#所以new Object()至少占用 对象头(8B+4B(64位开启指针压缩)=12B)+实例数据(0B)+padding(4B) = 16B

## 类加载

- 类代码块

- 普通类 代码块编译后 转换为 无参构造方法体  
(若已存在显示的无参构造方法, 则将代码块插入到无参构造方法最开始位置)
- 静态类代码块

- 普通类加载

- Class.forName(className)  
加载类字节码到内存, 并初始化类, 为类静态变量赋初始值, 并执行静态代码块
- ClassLoader.loadClass(className)  
ClassLoader加载类,只是将字节码加载到内存, 并不会初始化类(即为类静态变量赋初始值, 调用类的static代码块)  
Thread.currentThread().getContextClassLoader().loadClass(className);
- 调用类中常量时不会初始化类
- 调用类中静态属性时, 会为类静态变量赋初始值, 初始化父类及子类, 及执行父类static代码块及子类static代码块
- new 对象时, 会加载并初始化父类及子类对象,执行父类static代码块和父类无参构造方法, 及子类static代码块和子类构造方法和为类静态变量赋初始值,

- 内部类

```
class Test {  
    void test() {  
        class X { // "Test$1X"  
        }  
    }  
    void test2() {  
        class X { // "Test$2X"  
        }  
        class Y { // "Test$1Y"  
        }  
        System.out.println(new X()){ } // "Test$1"  
        System.out.println(new X()){ } // "Test$2"  
        System.out.println(new Y()){ } // "Test$3"  
    }  
    // Test$Test2$1Foo  
    static class Test2 {  
        void test() {  
            class Foo {
```

```

        void bar(){}
    }
}
}
}

```

- 局部类LocalClass
  - 类名为 所在类\$1类名
  - 如: ChildClass\$1LocalClass
  - 若相同的LocalClass名出现多次, 则类名依次为
  - ChildClass\$1LocalClass
  - ChildClass\$2LocalClass
  - 序号从1开始
  - 只能定义class,不能为enum,interface
- 成员类MemberClass
  - 类名为 所在类\$类名
  - 如: ChildClass\$InnerClass.class
  - enum,interface编译后为static
    - public static enum MemberEnumReflexClassParent {}
    - public static abstract interface MemberInterfaceReflexClassParent{}
  - class 编译后依然为class
  - 成员类调用基类的成员变量时, 将该成员变量作为成员类的成员变量, 变量名为this\$0, 并被final修饰
  - 并生成参数为所在类对象的构造方法
- 匿名内部类AnonymousClass
  - 不能显示的继承和实现其他类或接口
  - 类名为所在类\$1.class
  - 如: ChildClass\$1.class
  - new Class(){};
    - extents 基类
    - 无interface类
  - new Interface(){};
    - extents Object
    - 实现 基类
  - 匿名内部类被final修饰, 即 final class
- 匿名类/局部类调用所在类的成员变量和局部变量时,需要将传入的变量转换为final,防止成员变量和局部变量在匿名类中被修改, 变量在所在类运行结束后, 可能依然在匿名类/局部类中被使用, 导致内存泄漏。
  - 匿名类/局部类调用所在类的成员变量和局部变量时,会将变量生成为自己的成员变量, 并生成该成员变量的构造方法, 在调用位置使用改构造初始化对象。 , 并被final修饰。
  - 使用所在类成员变量时, 将所在类加入到成员变量中并创建创建基于所在类的构造方法, 即 new Clasz(this,变量) , 变量名为this\$0;
  - 使用所在类局部变量时, 会将局部变量加入到成员变量中, 并创建创建基于所在类

的构造方法，，即 new Clasz(final 变量)：局部变量的变量名为val\$变量名，并被final修饰。

创建匿名类/局部类实例，会转换为如下方式(转换为调用构造方法)：

```
ChildClass.2LocalClass lc = new ChildClass.2LocalClass(this, i);
```

```
ChildClass.2LocalClass lc2 = new ChildClass.2LocalClass(this, i);
```

## 注解

- 注解
  - interface Annotation
    - @interface 隐式继承(extends) java.lang.annotation.Annotation
    - 1个RetentionPolicy
    - 1~n个ElementType
  - 成员变量：注解的成员变量都是方法，可以有默认值。  
返回值为8大基本数据类型，String,Class,enum,Annotation及以上类型数组
  - value
    - 如果Annotation里有一个名为“value”的成员变量，使用该Annotation时，可以直接使用XXX(val)形式为value成员变量赋值，无须使用name=val形式。
  - implement注解
    - 注解隐式继承了Annotation接口，所以可以实现一个注解
    - 可动态为某个标识添加注解的实现
  - 主要用途
    - 生成文档
    - 编译检查
    - 编译时动态处理
    - 运行时动态处理，如反射
- 内置注解
  - @Override
    - 检查该方法是否重载方法。如果发现其父类，或者是引用的接口中并没有该方法时，会报编译错误。
  - @Deprecated
    - 标记过时方法。如果使用该方法，会报编译警告
  - @SuppressWarnings
    - 指示编译器忽略注解中声明的警告
  - 元注解
    - @Documented
      - 标记注解是否会包含在文档中
    - @Retention
      - 标记其它注解类型保留的生命周期
        - RetentionPolicy.SOURCE
          - 标记Annotation只保留在源代码中，编译器编译时，直接丢弃

这种Annotation

- RetentionPolicy.CLASS
  - 标记编译器把Annotation记录在class文件中。当运行Java程序时，JVM中不再保留该Annotation
- RetentionPolicy.RUNTIME
  - 编译器把Annotation记录在class文件中。当运行Java程序时，JVM会保留该Annotation，程序可以通过反射获取该Annotation的信息。
- @Target
  - 标记注解作用位置
    - java.lang.annotation.ElementType
      - ElementType.TYPE
        - 修饰类，接口或枚举类型
      - ElementType.FIELD
        - 修饰成员变量
      - ElementType.METHOD
        - 修饰方法
      - ElementType.PARAMETER
        - 修饰方法参数
      - ElementType.CONSTRUCTOR
        - 修饰构造器
      - ElementType.LOCAL\_VARIABLE
        - 修饰局部变量
      - ElementType.ANNOTATION\_TYPE
        - 修饰注解
      - ElementType.PACKAGE
        - 修饰包
  - @Inherited
    - 标注子类自动继承父类注解  
(子类extends父类会继承注解  
子类实现接口不继承注解  
子接口extends父接口不继承注解)
  - JDK7+
    - @SafeVarargs
      - JDK7, 忽略任何使用参数为泛型变量的方法或构造函数调用产生的警告
    - @FunctionalInterface
      - JDK8, 标记一个匿名函数或函数式接口
    - @Repeatable
      - JDK8, 标记某注解可以在同一声明上使用多次

## 泛型(generics)

- 泛型的本质是参数化类型
  - 泛型提供了编译时类型安全监测机制
  - 泛型作用于编译阶段
  - Java库中 E表示集合的元素类型, K 和 V分别表示表的关键字与值的类型
  - T (需要时还可以用临近的字母 U 和 S) 表示“任意类型”
  - 类型变量放在修饰符 (这里是 public static) 的后面, 返回类型的前面。
  - 解决容器的类型安全, 使用泛型后, 能让编译器在编译的时候借助传入的类型参数检查对容器的插入
  - 主要为Collection类型
  - 泛型的使用
    - 泛型接口(generics interface)
      - `public interface Generator {...}`
    - 泛型类(generics class)
      - `public class Pair<T extend Object,U extend Object> { ... }`
    - 泛型方法(generics method)
      - `public static T getMiddle(T... a) {.. }`
  - 协变: 子类能向父类转换 `Animal a1=new Cat();`
  - 逆变: 父类能向子类转换 `Cat a2=(Cat)a1;`
  - 不变: 两者均不能转变
  - 泛型与向上转型的概念
- 泛型上下边界

- extends,super关键字与通配符?
  - `List < E extends Fruit>`
    - 作用于类, 接口, 方法, 变量泛型定义
  - `List < ? >` 通配符
    - 具体类型为任意类型
    - 作用于方法, 变量泛型定义
    - 不能向通配符泛型写入  
可以读取, 但必须使用Object接
  - `List < ? extends Fruit>`

上边界通配符

    - 具体类型必须是Fruit的子类类型
    - 作用于方法, 变量泛型定义
    - 上界<? extends T>不能往里存, 只能往外取,取T类型
    - `Plate<? extends Fruit> fruitPlate = new Plate(new Apple());`
  - `List < ? super Fruit>`

下边界通配符

    - 泛型具体类型必须是Fruit的父类类型

- 作用于方法，变量泛型定义
  - 下界<? super T>可以存，但往外取只能放在Object对象里
  - Plate<? super Fruit> fruitPlate = new Plate(new Fruit());
- //普通通配符不能存，只能用Object接收读取
 

```
Collection<?> c = new ArrayList();
Object a = c.iterator().next();
//c.add(new Object());
//下界通配符super可以存，但取只能用Object接
Collection<? super Integer> css = new ArrayList();
css.add(1);
css.add(2);
Object aa = css.iterator().next();
//上届界通配符extends 可以取，但不能存
Collection<? extends Number> cssn = new ArrayList();
//cssn.add(new Integer(1));
Number aaa = cssn.iterator().next();
```
- PECS（Producer Extends Consumer Super）原则，
 

如果参数化类型表示一个生产者，就使用<? extends T>；如果它表示一个消费者，就使用<? super T>。

频繁往外读取内容的，适合用上界Extends。

经常往里插入的，适合用下界Super。
- 泛型重载了extends，super关键字来解决通用泛型的表示
- extends还被用来指定擦除到的具体类型，比如，表示在运行时将E替换为Fruit,注意E表示的是一个具体的类型，但是这里的extend和通配符连续使用<? extends Fruit>这里通配符?表示一个通用类型，它所表示的泛型在编译的时候，被指定的具体的类型必须是Fruit的子类。比如List<? extends Fruit> list= new ArrayList，ArrayList<>中指定的类型必须是Apple,Orange等
- 静态方法与泛型
  - 静态方法无法访问类上定义的泛型；如果静态方法操作的引用数据类型不确定的时候，必须要将泛型定义在方法上。
  - 即：如果静态方法要使用泛型的话，必须将静态方法也定义成泛型方法
- 泛型数组
  - 在java中是“不能创建一个确切的泛型类型的数组”
    - 也就是说下面的这个例子是不可以的：
 

```
List[] ls = new ArrayList[10];
```

 而使用通配符创建泛型数组是可以的，如下面这个例子：
 

```
List[] ls = new ArrayList[10];
```

 这样也是可以的：
 

```
List[] ls = new ArrayList[10];
```

# 反射

- JAVA反射机制:

在运行时，对于任意一个类，能够知道类的所有属性和方法;

对于任意对象，都能够调用任意方法和属性;

这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。

动态创建类实例。

- 主要类对象

- java.lang.Class
    - java.lang.reflect.Field
    - java.lang.reflect.Method
    - java.lang.reflect.Constructor
    - java.lang.Package

- 2大类方法

- getXXXX
      - 获取所有内容，含父类继承  
getField,getMethod
    - getDeclaredXXX
      - 获取直接作用的内容，不含父类继承
    - getGenericXXX
      - 获取带泛型的对象  
ec.getGenericInterfaces()  
ec.getGenericSuperclass()

- Package

- claz.getPackage()
      - 获取类所在的包
    - Annotation[] as = pkg.getAnnotations();
      - 获取package所有注解，包含继承注解
    - Annotation a = pkg.getAnnotation(Annotation.class)
      - 根据注解类型获取注解对象
    - (JDK8+)Annotation[] at = pkg.getAnnotationsByType(Annotation.class);
      - 根据注解类型获取注解对象数组，若为可重复注解时返回所有类型注解对象
    - pkg.getDeclaredAnnotation...;
      - 获取直接作用于包的注解，不含继承的注解

- Class

- 获取Class对象的4种方式
      - .getClass()
      - 类.class
      - Class.forName(String)
      - ClassLoader.getSystemClassLoader().loadClass()



- 创建类实例
  - 无参构造方法创建
    - `Clazz clas = Class.forName("className");`  
`clas.newInstance();`
  - 有参构造方法
    - `Constructor[] cs = (Constructor[]) clas.getConstructors();`  
`for (Constructor c:cs) {`  
`if (c.getParameterCount() > 0) {`  
`rc = (ReflexClass) c.newInstance("");`  
`}`  
`}`
- `Class<?> ec = clas.getEnclosingClass()`
  - 返回该类是在那个类中定义的，比如直接定义的内部类或匿名内部类
- `Constructor<?> encs = clas.getEnclosingConstructor();`
  - 该类是在哪个构造函数中定义的，比如构造方法中定义的匿名内部类
- `Method em = clas.getEnclosingMethod();`
  - 该类是在哪个方法中定义的，比如方法中定义的匿名内部类
- `Class asSubclass(Class clazz)` 把传递的类的对象转换成代表其子类的对象
- `T cast(Object obj)` 把对象转换成代表类或是接口的对象
- `getClassLoader()` 获得类的加载器
- `getClasses()` 返回一个数组，数组中包含该类中所有公共类和接口类的对象
- `getDeclaredClasses()` 返回一个数组，数组中包含该类中所有类和接口类的对象
- `forName(String className)` 根据类名返回类的对象
- `getName()` 获得类的完整路径名字
- `newInstance()` 创建类的实例
- `getPackage()` 获得类的包
- `getSimpleName()` 获得类的名字
- `getSuperclass()` 获得当前类继承的父类的名字
- `getInterfaces()` 获得当前类实现的类或是接口
- `isAnnotation()` 如果是注解类型则返回true
- `isAnnotationPresent(Class<? extends Annotation> annotationClass)` 如果是指定类型注解类型则返回true
- `isAnonymousClass()` 如果是匿名类则返回true
- `isArray()` 如果是一个数组类则返回true
- `isEnum()` 如果是枚举类则返回true
- `isInstance(Object obj)` 如果obj是该类的实例则返回true
- `isInterface()` 如果是接口类则返回true

- isLocalClass() 如果是局部类则返回true
  - isMemberClass() 如果是内部类则返回true
  - .getClass().getGenericInfo() 获取泛型信息
  - .getClass().getTypeParameters()
- Constructor
  - Constructor<?>[] cs1 = clazz.getConstructors();
    - 获取类的所有public构造方法
  - Constructor<?>[] dcs = clazz.getDeclaredConstructors();
    - 获取类定义的所有构造方法
  - Constructor<?> ics = clazz.getConstructor(String.class);
    - 获取指定参数类型的构造方法
  - getConstructor(Class...<?> parameterTypes) 获得该类中与参数类型匹配的公有构造方法
  - getDeclaredConstructor(Class...<?> parameterTypes) 获得该类中与参数类型匹配的构造方法
  - Constructor方法
    - newInstance(Object... initargs) 根据传递的参数创建类的对象
- Field
  - getField(String name) 获得某个公有的属性对象
  - getFields() 获得所有公有的属性对象
  - getDeclaredField(String name) 获得某个属性对象
  - getDeclaredFields() 获得所有属性对象
  - Field方法
    - equals(Object obj) 属性与obj相等则返回true
    - get(Object obj) 获得obj中对应的属性值
    - set(Object obj, Object value) 设置obj中对应属性值
- Method
  - getMethod(String name, Class...<?> parameterTypes) 获得该类某个公有的方法
  - getMethods() 获得该类所有公有的方法
  - getDeclaredMethod(String name, Class...<?> parameterTypes) 获得该类某个方法
  - getDeclaredMethods() 获得该类所有方法
  - Method方法
    - invoke(Object obj, Object... args) 传递object对象及参数调用该对象对应的方法
    - Type[] empty = em.getGenericParameterTypes(); 获取参数列表的参数泛型列表  
实际为ParameterizedType类型

- `Type[] empt= em.getGenericParameterTypes();`
    - `((ParameterizedType)gcm1t[0]).getActualTypeArguments()[0]`
  - `//获取Class[]的参数列表`  
`Class[] cs = gcm1.getParameterTypes();`
  - `//获取定义的泛型`  
`TypeVariable[] tvs = gcm1.getTypeParameters();`
- Annotation
  - `getAnnotation(Class annotationClass)` 返回该类中与参数类型匹配的公有注解对象
  - `getAnnotations()` 返回该类所有的公有注解对象
  - `getDeclaredAnnotation(Class annotationClass)` 返回该类中与参数类型匹配的所有注解对象
  - `getDeclaredAnnotations()` 返回该类所有的注解对象
- 枚举Enum
  - enum关键字定义的枚举  
 隐式继承自extends `java.lang.Enum`  
 泛型为当前定义对象
  - 枚举不能定义  
`readObject,writeObject,readObjectNoData,writeReplace,readResolve` 等方法处理反序列化  
`java.io.ObjectStreamClass`中定义序列化时单独对enum做了处理
  - enum 类型不支持 `public` 和 `protected` 修饰符的构造方法，因此构造函数一定要是 `private` 或 `friendly` 的。也正因为如此，所以枚举对象是无法在程序中通过直接调用其构造方法来初始化的  
 同时在`Construct.newInstance`中判断了枚举不能被实例化处理：  
`Cannot reflectively create enum objects`
  - 定义 enum 类型时候，如果是简单类型，那么最后一个枚举值后不用跟任何一个符号；但如果有定制方法，那么最后一个枚举值与后面代码要用分号';'隔开，不能用逗号或空格。
  - 枚举中的值在编译后会成为 `final static` 成员变量
    - `javap EnumObj.class`  
`Compiled from "EnumObj.java"`  
`public final class org.kangspace.common.reflex.EnumObj extends`  
`java.lang.Enum<org.kangspace.common.reflex.EnumObj>`  
`implements java.io.Serializable {`  
`public static final org.kangspace.common.reflex.EnumObj A;`  
`public static org.kangspace.common.reflex.EnumObj[] values();`  
`public static org.kangspace.common.reflex.EnumObj`  
`valueOf(java.lang.String);`  
`public static void main(java.lang.String[]);`  
`static {};`  
`}`

# Java中的引用

影响VM对象内存回收操作

<https://droidyue.com/blog/2014/10/12/understanding-weakreference-in-java/>

[https://blog.csdn.net/qq\\_39192827/article/details/85611873](https://blog.csdn.net/qq_39192827/article/details/85611873)

- 强引用  
普通编码中的对象赋值就是强引用  
存在强引的对象用不会被GC回收
- 软引用(`java.lang.ref.SoftReference`)  
存在软引用的对象在(不存在其他强引用)内存不足时才会被回收  
可用`get()`获取被引用对象,使用引用对象需要校验`null`
- 弱引用(`java.lang.ref.WeakReference`)  
存在弱引用的对象在(不存在其他强引用)GC时, 会被回收  
可用`get()`获取被引用对象,使用引用对象需要校验`null`
- 虚引用(`java.lang.ref.PhantomReference`)  
不能使用`get()`获取被引用对象
  - 虚引用使用场景主要由两个。  
一, 它允许你知道具体何时其引用的对象从内存中移除。而实际上这是Java中唯一的方式。(在对象被销毁时, 把该引用加入到引用队列中)  
第二点, 虚引用可以避免很多析构时的问题。`finalize`方法可以通过创建强引用指向快被销毁的对象来让这些对象重新复活。然而, 一个重写了`finalize`方法的对象如果想要被回收掉, 需要经历两个单独的垃圾收集周期。

## 动态代理

- 静态代理(通过传入被代理对象方式,生成代理对象,硬编码)
- `java.lang.reflect.Proxy`(基于JDK的动态代理):  
基于接口的代理,被代理类必须实现某个接口,使用`java.lang.reflect.InvocationHandler`处理及调用被代理类方法
  - `(NormalObject) Proxy.newProxyInstance(object.getClass().getClassLoader(), object.getClass().getInterfaces(), new InvocationHandler() {`  
    `@Override`  
    `public Object invoke(Object proxy, Method method, Object[] args) throws`  
    `Throwable {`  
        `System.out.println("动态代理前执行");`  
        `//执行目标对象方法`  
        `Object returnValue = method.invoke(object, args);`  
        `System.out.println("动态代理后执行");`  
        `return returnValue;`  
    `}`  
    `});`  
■ JDK动态代理不会代理接口的default方法

- 基于CGLIB的动态代理:CGLIB使用ASM操作字节码可对接口和类进行代理,使用生成子类的方式,及java.lang.reflect.InvocationHandler方式进行方法代理处理  
Proxy.newProxyInstance(object.getClass().getClassLoader(),  
object.getClass().getInterfaces(), new InvocationHandler() {  
@Override

```

        public Object invoke(Object proxy, Method method, Object[] args) throws
        Throwable {
            System.out.println("动态代理前执行");
            //执行目标对象方法
            Object returnValue = method.invoke(object, args);
            System.out.println("动态代理后执行");
            return returnValue;
        }
    });
}

```

- CGLIB会代理default方法
- public final void run()
 

```

{
    MethodInterceptor tmp4_1 = this.CGLIB$CALLBACK_0;
    if (tmp4_1 == null)
    {
        tmp4_1;
        CGLIB$BIND_CALLBACKS(this);
    }
    if (this.CGLIB$CALLBACK_0 != null)
        return;
    super.run();
}

```

## 字节码操作

- BCEL(Byte Code Engineering Library)
- ASM Java字节码操作和分析框架, 能够操作现有类及动态生成新类  
<https://asm.ow2.io/>  
<https://www.cnblogs.com/fanguangdexiaoyuer/p/12522765.html>
- cglib(CodeGenerationLibrary)  
基于ASM, 实现类动态代理框架
  - 与Java原生Proxy动态代理的区别:  
Java原生Proxy代理只支持代理接口, 无法代理没有接口的普通类  
cglib能够代理普通类和接口, 是增功能更强(原理是生成子类)
  - Enhancer  
Enhancer可能是CGLIB中最常用的一个类, 和Java1.3动态代理中引入的Proxy类差不多(如果对Proxy不懂, 可以参考这里)。和Proxy不同的是, Enhancer既能够代理普通的class, 也能够代理接口。Enhancer创建一个被代理对象的子类并且拦截所有的方法调用(包括从Object中继承的toString和hashCode方法)。Enhancer不

能够拦截final方法，例如Object.getClass()方法，这是由于Java final方法语义决定的。基于同样的道理，Enhancer也不能对final类进行代理操作。这也是Hibernate为什么不能持久化final class的原因。

利用callback对象实现方法拦截:net.sf.cglib.proxy.MethodInterceptor extends Callback

```
enhancer.setCallback();
```

```
■ public static NormalObject cglibDynamicProxyFn() {
    Enhancer enhancer = new Enhancer();
    enhancer.setSuperclass(NormalObject.class);
    /* enhancer.setCallback(new FixedValue() {
        @Override
        public Object loadObject() throws Exception {
            return "Hello cglib";
        }
    });*/
    //方法代理
    enhancer.setCallback(new MethodInterceptor() {
        @Override
        public Object intercept(Object o, Method method, Object[] args,
        MethodProxy methodProxy) throws Throwable {
            if ("action".equals(method.getName())) {
                System.out.println("cglib 代理方法执行前:" + method.getName());
                Object o1 = methodProxy.invokeSuper(o, args);
                System.out.println("cglib 代理方法执行后:" + method.getName());
                return o1;
            }
            return methodProxy.invokeSuper(o, args);
        }
    });
    NormalObject proxy = (NormalObject) enhancer.create();
    System.out.println(proxy.toString());
    System.out.println(proxy.getClass());
    return proxy;
}
```

#### ■ ImmutableBean

通过名字就可以知道，不可变的Bean。ImmutableBean允许创建一个原来对象的包装类，这个包装类是不可变的，任何改变底层对象的包装类操作都会抛出IllegalStateException。但是我们可以通过直接操作底层对象来改变包装类对象。这有点类似于Guava中的不可变视图

#### ■ Bean generator

cglib提供的一个操作bean的工具，使用它能够在运行时动态的创建一个bean。

#### ■ Bean Copier

cglib提供的能够从一个bean复制到另一个bean中，而且其还提供了一个转换器，用来在转换的时候对bean的属性进行操作

- Javassist  
基于ASM
  - CClass,CMethod,CField等

# 并发编程

- Thread
  - extends Thread  
@override void run(){}
  - implements Runnable  
@override void run(){}
  - 线程状态
    - NEW  
新创建了一个线程对象。
    - RUNNABLE  
线程对象创建后，其他线程(比如main线程)调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取cpu的使用权。
    - RUNNING  
可运行状态(runnable)的线程获得了cpu 时间片 (timeslice) ，执行程序代码
    - BLOCKED  
阻塞状态是指线程因为某种原因放弃了cpu 使用权，也即让出了cpu timeslice，暂时停止运行。直到线程进入可运行(runnable)状态，才有机会再次获得cpu timeslice 转到运行(running)状态。阻塞的情况分三种：
      - (一). 等待阻塞：运行(running)的线程执行o.wait()方法，JVM会把该线程放入等待队列(waiting queue)中。
      - (二). 同步阻塞：运行(running)的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池(lock pool)中。
      - (三). 其他阻塞：运行(running)的线程执行Thread.sleep(long ms) 或t.join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入可运行(runnable)状态。
    - DEAD  
线程run()、main() 方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。
  - 暂停线程
    - suspend  
暂停
      - 问题： 使用suspend暂停线程时，会导致公共同步对象被暂停线程

独占，导致无法被其他线程访问

- resume

恢复

- 终止线程

- thread.interrupt()

中断线程

- thread.interrupted

测试当前线程是否已经中断，清除中断标记

- thread.isInterrupted()

测试当前线程是否已经中断，不清除中断标记

- 设置对出标志位

- stop

- 使用stop会时线程终止时，线程内的数据结果不一致，

如：obj.a,b="a","aa";

```
thread.run(){
```

```
    a = "b";
```

```
    Thread.sleep(1000L);
```

```
    b="bb";
```

```
}
```

在b="bb"未执行前stop线程，最终b="bb"未被赋值，

结果为: "a" "aa"

- 异常

- 线程优先级

- 最小为1，最大为10

默认为5

- 优先级有继承性，A线程中启动B,B的优先级同A

- 优先级具有随机性，相同处理逻辑，不保证优先级高的线程先执行完

- 线程组:

不指定线程组时，使用Parent线程组

- Thread.join

等待线程结束

- Thread.sleep

- 释放CPU资源，不释放锁

- synchronized

- 可以修饰方法，代码块

保证内存可见性及线程安全

非公平锁

可重入锁

同步不具备继承性

synchronized方法弊端：占用整个方法资源，导致其他调用方阻塞等待该锁，

可用synchronized代码块

多线程访问下会阻塞

- synchronized可重入



- 子类调用父类的synchronized方法，  
若父类方法中以this作为锁对象，则该this为子类实例
- synchronized锁是作用于线程的
- 同一线程在调用自己类中其他synchronized方法/块或调用父类的synchronized方法/块都不会阻碍该线程的执行，就是说同一线程对同一个对象锁是可重入的，而且同一个线程可以获取同一把锁多次，也就是可以多次重入。因为java线程是基于“每线程（per-thread）”，而不是基于“每调用（per-invocation）”的（java中线程获得对象锁的操作是以每线程为粒度的，per-invocation互斥体获得对象锁的操作是以每调用作为粒度的）
- 重入锁是怎么实现可重入性的：其实现方法是为每个锁关联一个线程持有者和计数器，当计数器为0时表示该锁没有被任何线程持有，那么任何线程都可能获得该锁而调用相应的方法；当某一线程请求成功后，JVM会记下锁的持有线程，并且将计数器置为1；此时其它线程请求该锁，则必须等待；而该持有锁的线程如果再次请求这个锁，就可以再次拿到这个锁，同时计数器会递增；当线程退出同步代码块时，计数器会递减，如果计数器为0，则释放该锁  
锁对象的对象头MarkWord区域会保存锁信息
- synchronized(Class)  
static synchronized{}
  - 所有使用当前Class的线程持有同一类锁
- synchronized(Object)  
synchronized(this)
  - 所有使用当前对象的线程持有同一个对象锁
- String 常量池特性
  - String常量池对同一个字符串会使用相同的内存地址，所以用synchronized(String)时为同一锁
- synchronized1.6优化内容  
<https://zhuanlan.zhihu.com/p/92808298>
  - 1.6之前,synchronized为重量级锁,因为synchronized的monitor是依赖操作系统完成的,所以需要程序由用户态切换为内核态,  
JVM中monitorenter和monitorexit字节码依赖于底层的操作系统的Mutex Lock来实现,由于使用Mutex Lock需要将当前线程挂起并从用户态切换到内核态来执行,这种操作是非常昂贵的
  - 1.6时优化为锁升级流程，引入了偏向锁,轻量级锁,重量级锁  
每个对象都由对象头MarkWord,指向类的指针及数组长度组成,MarkWorld记录了对象和锁的有关信息;  
锁升级主要依赖MarkWord中锁标志位和释放偏向锁标志位  
一般的synchronized同步锁升级步骤是：  
偏向锁 -> 轻量级锁 -> 重量级锁  
JDK1.6优化目的是为了减少synchronized带来的线程切换次数;  
1.偏向锁  
当有线程访问同步代码或方法时，线程只需要判断对象头的Mark Word中判断一下是否有偏向锁指向线程ID。  
偏向锁记录过程:

线程抢到了对象的同步锁(锁标志为01参考上图即无其他线程占用)  
对象Mark Word 将是否偏向标志位设置为1  
记录抢到锁的线程ID  
进入偏向状态  
偏向锁能够做到,在短时间内若同一个线程再次访问加锁的同步代码或方法时,只需要去判断对象头MarkWord是否有偏向锁指向当前线程id,不需要再进入monitor竞争对象;

#### ■ 轻量级锁

一旦出现 其他线程竞争资源,偏向锁就会被撤销,偏向锁的插销需要等待全局安全点, 暂停持有该锁的线程, 同时检查该线程是否还在执行该方法, 如果是, 则升级锁为轻量级锁, 反之则其他线程抢占;

轻量级锁意味着标示该资源现在处于竞争状态。

当有其他线程想访问加了轻量级锁的资源时, 会使用自旋锁优化, 来进行资源访问;

自旋策略:

JVM 提供了一种自旋锁, 可以通过自旋方式不断尝试获取锁, 从而避免线程被挂起阻塞。这是基于大多数情况下, 线程持有锁的时间都不会太长, 毕竟线程被挂起阻塞可能会得不偿失。

#### 3.重量级锁:

自旋失败, 很大概率 再一次自选也是失败, 因此直接升级成重量级锁, 进行线程阻塞, 减少cpu消耗。

当锁升级为重量级锁后, 未抢到锁的线程都会被阻塞, 进入阻塞队列

总结:

synchronized锁升级实际上是把本来的悲观锁变成了在一定条件下 使用无锁(同样线程获取相同资源的偏向锁), 以及使用乐观(自旋锁cas)和一定条件下悲观(重量级锁)的形式。

#### ■ volatile

##### ■ 只能修饰成员变量

保证内存可见性, 主要解决多线程环境下访问资源的同步性

直接操作为线程安全

复合操作(i++,i=i+1)为线程不安全,因为+/-操作不是线程安全

多线程访问下不会阻塞

#### ■ ThreadLocal

##### ■ 提供了线程存储数据的能力

##### ■ threadLocal.get()

threadLocal.set()

##### ■ ThreadLocal.ThreadLocalMap

ThreadLocalMap为ThreadLocal的静态内部类

ThreadLocal 为每个Thread创建一个ThreadLocalMap;

ThreadLocalMap中维护了Entry<ThreadLocal,value>[] table数组(数组默认大小:16,空间不够时扩充为原空间的2倍);

ThreadLocal.get()时, 先获取当前线程中的ThreadLocalMap(),若不存在, 则创建ThreadLocalMap,然后保存当前ThreadLocal对象的null值。

##### ■ Entry extends WeakReference

Entry继承自WeakReference (弱引用, 生命周期只能存活到下次GC前), 但只有Key是弱引用类型的, Value并非弱引用。(问题马上就来

了)

- Entry[] 索引确定:  
firstKey.threadLocalHashCode & (INITIAL\_CAPACITY - 1)  
取自AtomicInteger ThreadLocal.nextHashCode 自增  
自增增量为: 0x61c88647  
0x61c88647是斐波那契散列乘数,它的优点是通过它散列(hash)出来的结果分布会比较均匀, 可以很大程度上避免hash冲突
- InheritableThreadLocal extends  
java.lang.ThreadLocal
  - 重写了ThreadLocal3个方法  
将只影响Thread类中的inheritableThreadLocals变量,  
不操作threadLocals变量
    - T childVal(T parentValue)
    - ThreadLocalMap getMap(t)
    - createMap(Thread t,firstValue)
  - 创建线程默认会继承父类的inheritableThreadLocals中的值
- ThreadLocal特性
  - ThreadLocal和Synchronized都是为了解决多线程中相同变量的访问冲突问题, 不同的点是Synchronized是通过线程等待, 牺牲时间来解决访问冲突
  - ThreadLocal是通过每个线程单独一份存储空间, 牺牲空间来解决冲突, 并且相比于Synchronized, ThreadLocal具有线程隔离的效果, 只有在线程内才能获取到对应的值, 线程外则不能访问到想要的值。
  - 正因为ThreadLocal的线程隔离特性, 使他的应用场景相对来说更为特殊一些。在android中Looper、ActivityThread以及AMS中都用了ThreadLocal。当某些数据是以线程为作用域并且不同线程具有不同的数据副本的时候, 就可以考虑采用ThreadLocal。
- ThreadLocal.HASH\_INCREMENT = 0x61c88647;()
  - 实例变量threadLocalHashCode, 每当创建ThreadLocal实例时这个值都会累加 0x61c88647, 目的在上面的注释中已经写的很清楚了: 为了让哈希码能均匀的分布在2的N次方的数组里, 即 Entry[] table
- 问题:
  - ThreadLocalMap Entry为什么使用弱引用
    - 为了处理非常大和生命周期非常长的线程, 哈希表使用弱引用作为key。  
当ThreadLocal对象被置为null时, 如果Entry中key为强引用时, 在线程存活周期内, ThreadLocal在堆中的内存不会被回收, 因为Entry中强引用了ThreadLocal对象, 导致内存泄漏。
  - 内存泄漏
    - 当ThreadLocal值使用后没有remove(),  
由于ThreadLocalMap的key是弱引用, 而Value是强引用。这就导致了一个问题, ThreadLocal在没有外部对象强引用时, 发生GC时弱引用Key会被回收, 而Value不会回收。  
当线程没有结束, 但是ThreadLocal已经被回收, 则可能导致线程

中存在ThreadLocalMap<null, Object>的键值对，造成内存泄露。  
(ThreadLocal被回收，ThreadLocal关联的线程共享变量还存在)。

注意: 在调用threadLocal.get(),set(),remove()后，ThreadLocal有机会把部分key为null的值清除掉

所以，使用使用ThreadLocal后及时清理ThreadLocal的值。

- 为了防止此类情况的出现，我们有两种手段。
  - 1、使用完线程共享变量后，显示调用ThreadLocalMap.remove方法清除线程共享变量；  
既然Key是弱引用，那么我们要做的事，就是在调用ThreadLocal的get()、set()方法时完成后再调用remove方法，将Entry节点和Map的引用关系移除，这样整个Entry对象在GC Roots分析后就变成不可达了，下次GC的时候就可以被回收。
  - 2、JDK建议ThreadLocal定义为private static，这样ThreadLocal的弱引用问题则不存在了。
- 内存溢出针对的是ThreadLocalMap value
- java.util.concurrent.ThreadFactory
  - 接口,只有1个 Thread newThread(Runnable r) 方法
- java.util.concurrent.ThreadPoolExecutor (ExecutorService)
  - Executors
    - //创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程（无限创建）。  
//最大线程数Integer.MAX\_MAX\_VALUE , keepAliveTime: 60s  
// SynchronousQueue 没有容量，是无缓冲等待队列，是一个不存储元素的阻塞队列  
ExecutorService cachedThreadPool =  
Executors.newCachedThreadPool();
      - CachedThreadPool使用的是SynchronousQueue的  
入队：offer(E e)  
出队：poll(long timeout, TimeUnit unit)  
工作线程调用poll阻塞，等待timeout时间，如果超时，则返回null并回收线程；如果在等待期内，有任务入队，则成功返回任务，继续执行线程while循环。  
在runWorker()中getTask()方法中处理的。
    - //创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。  
//LinkedBlockingQueue 队列最大长度:Integer.MAX\_MAX\_VALUE 是一个无界缓冲等待队列  
ExecutorService fixedThreadPool =  
Executors.newFixedThreadPool(100);

- //创建一个定长线程池，支持定时及周期性任务执行。  
 //最大线程数:Integer.MAX\_VALUE  
 //DelayedWorkQueue  
 ExecutorService scheduledThreadPool =  
 Executors.newScheduledThreadPool(100);
- //创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。  
 //LinkedBlockingQueue 队列最大长度:Integer.MAX\_VALUE  
 ExecutorService singleThreadExecutor =  
 Executors.newSingleThreadExecutor();
- //自定义创建线程池  
 ThreadPoolExecutor executor = new ThreadPoolExecutor(0,1,0,  
 TimeUnit.MILLISECONDS,new LinkedBlockingQueue<>(10));
- 无法加入队列时抛出RejectedExecutionException
- execute流程
  - 1、如果线程池中的线程数量少于corePoolSize，就创建新的线程来执行新添加的任务
  - 2、如果线程池中的线程数量大于等于corePoolSize，但队列workQueue未满，则将新添加的任务放到workQueue中
  - 3、如果线程池中的线程数量大于等于corePoolSize，且队列workQueue已满，但线程池中的线程数量小于maximumPoolSize，则会创建新的线程来处理被添加的任务
  - 4、如果线程池中的线程数量等于了maximumPoolSize，就用RejectedExecutionHandler来执行拒绝策略
- ThreadPoolExecutor参数
  - corePoolSize 核心线程池数, 表示活动的最小工作线程数(当队列未满时,最大工作线程数(当工作线程>=corePoolSize,且队列未满时,此时corePoolSize为最大工作线程数, 其他任务进入队列排队;最后若入队失败, 则执行reject(Runnable command));  
 当队列已满时,会创建<maximumPoolSize大小的非核心线程数);
  - maximumPoolSize 线程池最大工作线程数
  - keepAliveTime 活动线程的最大空闲时间
  - TimeUnit unit 活动线程最大空闲时间单位
  - allowCoreThreadTimeOut:false 允许核心线程超时,默认为false,当为true时,空闲keepAliveTime后,工作线程退出
  - BlockingQueue workQueue 任务队列
- 方法
  - Future<> submit()->{} 提交一个可以获取返回结果的任务  
 内部创建一个 java.util.concurrent.FutureTask,然后将FutureTask 加入到execute中,并返回改对象, 若执行成功, 则返回null,反之抛出异常
  - void execute()->{} 提交一个任务
  - void shutdown()
  - void shutdownNow()
- java.util.concurrent.ForkJoinPool/ForkJoinTask

- Fork/Join框架其实就是指由ForkJoinPool作为线程池、ForkJoinTask(通常实现其三个抽象子类)为任务、ForkJoinWorkerThread作为执行任务的具体线程实体这三者构成的任务调度机制;  
ForkJoin框架的作用主要是为了实现将大型复杂任务进行递归的分解,直到任务足够小才直接执行,从而递归的返回各个足够小的任务的结果汇集成一个大的任务的结果,依次类推最终得出最初提交的那个大型复杂任务的结果.  
ForkJoinPool 最适合的是计算密集型的任务.
- ForkJoinPool管理ForkJoinTask线程池  
WorkQueue:支持工作窃取和外部任务提交的队列,  
初始化容量大小 $1 < 13 = 2^{13} = 8K$   
最大容量大小 $1 < 26 = 2^{26} = 64M$   
内部为ForkJoinTask数组
- ForkJoinWorkerThread:  
ForkJoinWorkerThread是被ForkJoinPool管理的工作线程,在创建出来之后都被设置成为了守护线程,由它来执行ForkJoinTasks
- ForkJoinTask:与FutureTask一样, ForkJoinTask也是Future的子类,不过它是一个抽象类,其实现过程中与ForkJoinPool相互交叉
  - RecursiveAction:不返回结果的任务
  - RecursiveTask:要返回结果的任务
  - CountedCompleter:操作完成触发钩子函数的操作(1.8+)
- fork:任务异步执行
- join:等待执行结果
- get:获取异步任务结果
- invoke:立即执行任务,并等待返回结果
- invokeAll:批量执行任务,并等待它们执行结束
- 任务的完成状态查询:isDone、isCompletedNormally、isCancelled、isCompletedAbnormally
- Concurrent包  
java.util.concurrent: J.U.C
  - java.util.concurrent.locks.AbstractQueuedSynchronizer抽象队列同步器
    - 由int state状态值,Node双向链表等待队列及状态值操作,队列操作,线程挂起/激活等相关方法组成(LockSupport.park,unpark())
- 状态值的修改依赖于sun.misc.Unsafe.compareAndSwapXXX()方法进行
  - 原子类  
java.util.concurrent.atomic
    - 原理
      - CAS: CompareAndSwap  
CAS需要3个操作数: 内存地址(V),  
旧的预期值(A) 和新值(B)。  
CAS执行时, 当且仅当V符合A时,处理器采用B更新V,否则不更新;不论是否更新, 都返回旧值

- 使用sun.misc.Unsafe.compareAndSwapInt()  
或sun.misc.Unsafe.compareAndSwapLong()提供  
sun.misc.Unsafe.getUnsafe()方法限制了只有启动类加载器  
(BootstrapClassLoader)才能访问该方法  
这些方法总是返回旧值
    - AtomicInteger 中 getAndAdd(),  
addAndGet(),incrementAndGet(),decrementAndGet() 中  
直接调用 unsafe.getAndAddInt(this, valueOffset, -1) ,  
unsafe.getAndAddLong  
等方法, 这些方法总返回旧值  
在这些方法中, 使用do{oldVal =  
getVal()}while(compareAndSwapInt(this,oldVal,newVal))  
一直尝试更新数据, 直到成功
    - AtomicInteger  
AtomicIntegerArray
      - volatile value
    - AtomicBoolean
    - AtomicLong  
AtomicLongArray
    - AtomicReference  
AtomicReferenceArray
  - 多线程同步
    - java.util.concurrent.CountDownLatch(AQS)
      - 原理
        - 利用 java.util.concurrent.locks.AbstractQueuedSynchronizer 实  
现创建Sync对象,  
基于AQS state计数器和AQS阻塞队列(等待线程会被加入到双向链  
表中Node)  
初始值为线程数量
    - countDown()方法:  
类似AtomicInteger, 基于同步器计数  
countDown()时: 利用CAS 更新state值, 每次减一  
最后一个执行countDown的线程会执行doReleaseShared()来释放阻塞队列中的线程
    - await():  
执行await()后在AQS阻塞队列中创建一个等待节点, 并  
LockSupport.park()阻塞当前线程, 直到最后一个线程调用countDown()释放  
LockSupport.unpark()当前线程
      - 作用
        - 使一个或多个线程等待其他线程完成后再执行
      - 用途
        - 1、项目启动时的核心服务检测
        - 2、模拟高并发
        - 3、zookeeper连接
- Zookeeper , Apache curator中使用了大量CountDownLatch 等待线程执行完成操

作

- 问题

- 只能使用1次

- java.util.concurrent.CyclicBarrier 循环屏障(ReentrantLock)

- 原理

- 1.CyclicBarrier内部定义了一个可重入锁ReentrantLock（非公平锁）对象和一个Condition，每当一个线程调用await方法的时候，计数器减1，同时判断计数器是否为0，为0执行runnable barrierAction方法，重置栅栏，并唤醒所有在lock队列里面等待的线程，否则进入lock的等待队列。

最后一个线程调用await()后，会执行最后的Runnable方法

所有线程进入屏障后，最后一个线程执行完Runnable方法后，所有线程突破屏障继续执行后续操作。

([https://blog.csdn.net/qq\\_39241239/article/details/87030142](https://blog.csdn.net/qq_39241239/article/details/87030142))

- CycleBarrier 定义了一个Generation类来标识一次循环是否broken中断

- cycleBarrier.reset()重置屏障,等待队列继续执行;

cycleBarrier.await()等到最后一个线程执行该方法时(返回:当前线程的到达索引,其中index getParties () -1表示第一个到达，零表示最后一个到达),触发屏障runAction,进入下一次循环,等待队列继续执行

- 示例：赛马

- 作用

- 让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，所有被屏障拦截的线程才会继续运行,当前线程被阻塞,等到执行完成屏障点回调时,激活各等待线程,线程继续执行;

可重置，可重复执行

- 用途

- 可以用于多线程计算数据，最后合并计算结果的场景。(且各线程在合并结果后可使用最终合并的结果继续进行业务处理)

- 与CountDownLatch区别

- CyclicBarrier强调的是n个线程相互等待，直到完成再执行任务，计数器可以重置，复用，所以叫循环栅栏。

即n个线程等待执行完barrierAction后继续执行

- CountDownLatch允许一个或多个线程等待直到其他线程完成操作，只能使用一次。

- java.util.concurrent.Semaphore 信号量(AQS)

- 原理

- 基于AQS(AbstractQueuedSynchronizer)

Semaphore内部原理是通过AQS实现的。Semaphore中定义了Sync抽象类，而Sync又继承了AbstractQueuedSynchronizer，Semaphore中对许可的获取与释放，是使用CAS通过对AQS中state的操作实现的。

Semaphore是J.U.C包下的许可控制类，维护了一个许可集，通常用于限制可以访问某些资源（物理或逻辑的）的线程数目，或对资源访问的许可控制。

- Semaphore对许可的分配有两种策略，公平策略和非公平策略，没有明确指明时，默认为非公平策略。

- 公平策略：根据方法调用顺序（即先进先出；FIFO）来选择线程、获得许可。非公平策略：不对线程获取许可的顺序做任何保证。

- 用途



- 服务限流，控制线程的并发数量

只有指定数量的线程可以执行，其他线程进入等待队列

- Lock

- (synchronized)Object.wait/notify/notifyAll

- 1、wait()、notify/notifyAll() 方法是Object的本地final方法，无法被重写。
      - wait 释放锁和CPU资源
    - 2、wait()执行后拥有当前锁的线程会释放该线程锁，并处于等待状态（等待重新获取锁）
    - 3、notify/notifyAll() 执行后会唤醒处于等待状态线程获取线程锁、只是notify()只会随机唤醒其中之一获取线程锁，notifyAll() 会唤醒所有处于等待状态的线程抢夺线程锁。

- java.util.concurrent.locks.ReentrantLock

- 原理

- ReentrantLock 为可重入独占锁，乐观锁(CAS)  
利用AQS 中Node双向链表保存线程队列  
同一时间只有一个线程在执行ReentrantLock.lock()方法后面得任务  
通过实现AbstractQueuedSynchronizer创建Sync类,再创建FairSync和NoFairSync实现公平锁和非公平锁

- 公平锁(FairSync extends Sync extends AbstractQueuedSynchronizer)

- 按加锁顺序获取锁  
(使用hasQueuedPredecessors()检查是否在队列中已存在等待线程,且等待线程不是当前线程,如果在当前线程之前有一个排队线程,则为true; 如果当前线程位于队列的头部或队列为空, 则为false,这也是与非公平锁获取锁的区别)

- 加锁.lock()

- lock.lock();

首先会判断 AQS 中的 state 是否等于 0(compareAndSetState(0, 1)), 0 表示目前没有其他线程获得锁, 当前线程就可以尝试获取锁。若当前状态值为0,且当前线程成功将0设置为1,则将当前锁的独占线程设置为当前线程,反之尝试获取锁acquire(1);  
注意:尝试之前会利用 hasQueuedPredecessors() 方法来判断 AQS 的队列中是否有其他线程, 如果有则不会尝试获取锁(这是公平锁特有的情况)。  
如果队列中没有线程就利用 CAS 来将 AQS 中的 state 修改为1, 也就是获取锁, 获取成功则将当前线程置为获得锁的独占线程(setExclusiveOwnerThread(current))。  
如果 state 大于 0 时, 说明锁已经被获取了, 则需要判断获取锁的线程是否为当前线程(ReentrantLock 支持重入), 是则需要将 state + 1, 并将值更新。  
(CAS-Compare and Swap基于CPU,CAS指令,原子性操作)

- CAS判断state是否为0来表示锁是否被占用；  
若未被占用，则独占锁
- 否则，尝试获取锁 `acquire()`
- 若尝试获取锁成功（锁就是被当前线程占用的，重入计数+1即可；或者锁正好被释放）
- 获取锁失败，则需要创建一个Node等待节点（包含了线程信息）入队，`acquireQueued(final Node node, int arg)`;
- 挂起线程 `acquireQueued`，挂起之前，会先尝试获取锁，只有确认失败之后，则挂起锁，并设置前置Node的状态为SIGNAL（以保障在释放锁的时候，可以保证唤醒Node的后驱节点线程）
  - 写入队列
    - 如果 `tryAcquire(arg)` 获取锁失败，则需要用 `addWaiter(Node.EXCLUSIVE)` 将当前线程写入队列中。
  - 写入之前需要将当前线程包装为一个 Node 对象(`addWaiter(Node.EXCLUSIVE)`)。
  - 释放锁.`unlock()`
    - 1.释放锁时,判断当前线程是否为持有独占锁的线程,不是则抛出 `IllegalMonitorStateException`异常,
    - 若是,则状态值直接减1(`releases`),若状态值为0时,则设置独占锁的线程为 `null(setExclusiveOwnerThread(null))`;
    - 2.唤醒下一个等待线程,若 `head.next==null || head.next.waitStatus>0` 则从双向链表的尾部,`waitStatus<=0`的节点,直到节点不为空,然后将该节点的线程唤醒;
- `LockSupport.unpark(s.thread)`
- 非公平锁(`NonFairSync extends Sync extends AbstractQueuedSynchronizer`)
- 随机获取锁
- (即,每次`lock()`时直接尝试获取锁,加锁时不校验等待队列是否有其他线程,其他操作同公平锁操作)
- `ReentrantLock`
  - `.lock()` 加锁,公平锁与非公平锁单独实现获取锁操作
  - `.tryLock()`尝试获取锁,返回成功/失败,该操作为非公平获取锁操作,不加入等待队列
- `public boolean tryLock() {`  
`return sync.nonfairTryAcquire(1);`  
`}`
- `.unlock()` 释放锁
- 小结
  - 创建锁对象 `Lock lock = new ReentrantLock()`
  - 在希望保证线程同步的代码之前显示调用 `lock.lock()` 尝试获取锁，若被其他线程占用，则阻塞
  - 执行完之后，一定得手动释放锁，否则会造成死锁 `lock.unlock()`；一般来讲，把释放锁的逻辑，放在需要线程同步的代码包装外的`finally`块中
  - `lock()` 和 `unlock()` 配套使用，不要出现一个比另一个用得多的情况
  - 不要出现`lock()`,`lock()`连续调用的情况，即两者之间没有释放锁`unlock()`的显示调用
  - `Condition`与`Lock`配套使用，通过 `Lock#newCondition()` 进行实例化
  - `lock()` 加锁，但不阻塞
  - `condition.await()` 释放锁,阻塞

condition.await()前需要加锁,后需要解锁

- Condition#await() 会释放lock, 线程阻塞; 直到线程中断or Condition#signal()被执行, 唤醒阻塞线程, 并重新获取lock

- Condition

- lock.newCondition() 创建条件锁

- await()

- // 使当前线程处于等待状态, 释放与Condition绑定的lock锁

// 直到 signal()方法被调用后, 被唤醒

// 唤醒后, 该线程会再次获取与条件绑定的 lock锁

- signal()

- 表示条件达成, 唤醒一个被条件阻塞的线程

- signalAll()

- 唤醒所有被条件阻塞的线程。

- 样例: ArrayBlockingQueue, LinkedBlockingQueue

- ReentrantReadWriteLock

读写锁: 读读共享, 写写互斥, 读写互斥, 写读互斥

- lock.readLock() 读锁(读读共享)

- lock() 允许多个线程同时执行lock()方法后面的代码

- lock.writeLock() 写锁

(写写互斥, 读写互斥)

- (写写互斥)writeLock().lock() 同一时间只允许一个线程执行lock()方法后面的

代码

- (读写互斥) 读锁时,不能写,

写锁时不能读

- java.util.concurrent.locks

不可重入, 默认占有许可证, 可以响应中断

LockSupport.park() 获取当前线程许可, 阻塞线程

LockSupport.unpark() 取消当前线程许可, 唤醒线程

- 底层为sun.misc.Unsafe:

//park

public native void park(boolean isAbsolute, long time);

//unpack

public native void unpark(Object var1);

- 与Object类的wait/notify机制相比, park/unpark有两个优点

- 以thread为操作对象更符合阻塞线程的直观定义

- 操作更精准, 可以准确地唤醒某一个线程 (notify随机唤醒一个线程,

notifyAll唤醒所有等待的线程), 增加了灵活性。

- sun.misc.Unsafe 提供CAS native操作方法

线程park/unpark

- Thread.sleep、Object.wait、LockSupport.park 区别

<https://www.cnblogs.com/tong-yuan/p/11768904.html>

- Thread.sleep(time)方法必须传入指定的时间, 线程将进入休眠状态, 通过jstack输出线程快照的话此时该线程的状态应该是TIMED\_WAITING, 表示休眠一段时间。

不能主动唤醒。

通过sleep方法进入休眠的线程不会释放持有的锁,会让出CPU时间片

- Object.wait() 线程的状态都将是WAITING状态  
必须获得对象上的锁后，才可以执行该对象的wait方法。否则程序会在运行时抛出IllegalMonitorStateException异常。  
Object.wait()方法需要在synchronized块中执行；  
Object.wait()不带超时的，需要另一个线程执行notify()来唤醒，但不一定继续执行后续内容；  
如果在wait()之前执行了notify()会怎样？抛出IllegalMonitorStateException异常；  
调用wait()方法后，线程进入休眠的同时，会释放持有的该对象的锁，  
- 通过LockSupport.park()方法，我们也可以让线程进入休眠。它的底层也是调用了Unsafe类的park方法，会响应中断interceptor()；  
需要使用LockSupport.unpark() 唤醒；  
LockSupport.park()不带超时的，需要另一个线程执行unpark()来唤醒，一定会继续执行后续内容；  
如果在park()之前执行了unpark()会怎样？线程不会被阻塞，直接跳过park()，继续执行后续内容；  
调用park方法进入休眠后，线程状态为WAITING  
另外，和wait方法不同，执行park进入休眠后并不会释放持有的锁。

- 队列Queue

- java.util.concurrent.BlockingQueue

- 基于ReentrantLock

- 实现(implement)

- ArrayBlockingQueue

- 基于数组的有界阻塞队列

- ArrayBlockingQueue底层是使用一个数组（Object数组）实现队列的，并且在构造ArrayBlockingQueue时需要指定容量，也就意味着底层数组一旦创建了，容量就不能改变了，因此ArrayBlockingQueue是一个容量限制的阻塞队列。  
因此，在队列全满时执行入队将会阻塞，在队列为空时出队同样将会阻塞。

- ArrayBlockingQueue的并发阻塞是通过ReentrantLock和Condition来实现的，ArrayBlockingQueue内部只有一把锁，意味着同一时刻只有一个线程能进行入队或者出队的操作。  
(可指定是否为公平锁,插入删除是否FIFO)

- lock = new ReentrantLock(fair);  
notEmpty = lock.newCondition();  
notFull = lock.newCondition();

- LinkedBlockingQueue

- 基于单向链表的有界队列

- LinkedBlockingQueue是一个基于单向链表实现的可选容量的阻塞队列。队头的元素是插入时间最长的，队尾的元素是最新插入的。新的元素将会被插入到队列的尾部。  
在于该队列至少有一个节点，头节点不含有元素  
如果在初始化时没有指定容量，那么默认使用int的最大值作为队列容量

## ■ 原理

- `LinkedBlockingQueue`中维持两把锁，一把锁用于入队`putLock`，一把锁用于出队`takeLock`，这也就意味着，同一时刻，只能有一个线程执行入队，其余执行入队的线程将会被阻塞；同时，可以有另一个线程执行出队，其余执行出队的线程将会被阻塞。换句话说，虽然入队和出队两个操作同时均只能有一个线程操作，但是可以一个入队线程和一个出队线程共同执行，也就意味着可能同时有两个线程在操作队列，那么为了维持线程安全，`LinkedBlockingQueue`使用一个`AtomicInteger`类型的变量表示当前队列中含有的元素个数，所以可以确保两个线程之间操作底层队列是线程安全的。

```
ReentrantLock takeLock = new ReentrantLock();
```

```
Condition notEmpty = takeLock.newCondition();
```

```
ReentrantLock putLock = new ReentrantLock();
```

```
Condition notFull = putLock.newCondition();
```

- `LinkedBlockingQueue`是允许两个线程同时在两端进行入队或出队的操作的，但一端同时只能有一个线程进行操作，这是通过两把锁来区分的；为了维持底部数据的统一，引入了`AtomicInteger`的一个`count`变量，表示队列中元素的个数。`count`只能在两个地方变化，一个是入队的方法（可以+1），另一个是出队的方法（可以-1），而`AtomicInteger`是原子安全的，所以也就确保了底层队列的数据同步。

## ■ `LinkedBlockingDeque`

基于双向链表的有界队列

- `LinkedBlockingDeque`是双向链表实现的双向并发阻塞队列。该阻塞队列同时支持FIFO和FILO两种操作方式，即可以从队列的头和尾同时操作(插入/删除)；并且，该阻塞队列是支持线程安全。一把锁

```
ReentrantLock lock = new ReentrantLock();
```

```
Condition notEmpty = lock.newCondition();
```

```
Condition notFull = lock.newCondition();
```

- 1、`add(E e)`：在不违反容量限制的情况下，将指定的元素插入此双端队列的末尾，返回值为`Boolean`。或抛出异常
- 2、`addFirst(E e)`：如果立即可行且不违反容量限制，则将指定的元素插入此双端队列的开头；如果当前没有空间可用，则抛出`IllegalStateException`。
- 3、`addLast(E e)`：如果立即可行且不违反容量限制，则将指定的元素插入此双端队列的末尾；如果当前没有空间可用，则抛出`IllegalStateException`。
- 包含队列的基本`offer`,`put`,`poll`,`take`，及`offerFirst`,`offerLast`(不阻塞)

putFirst,putLast(阻塞)  
pollFirst,pollLast(不阻塞)  
takeFirst,takeLast(阻塞)

- **LinkedBlockingQueue 与 ArrayBlockingQueue 比较**

- **ArrayBlockingQueue**

- 一个对象数组+一把锁+两个条件
    - 入队与出队都用同一把锁
    - 在只有入队高并发或出队高并发的情况下，因为操作数组，且不需要扩容，性能很高
    - 采用了数组，必须指定大小，即容量有限

- **LinkedBlockingQueue**

- 一个单向链表+两把锁+两个条件
    - 两把锁，一把用于入队，一把用于出队，有效的避免了入队与出队时使用一把锁带来的竞争。
    - 在入队与出队都高并发的情况下，性能比ArrayBlockingQueue高很多
    - 采用了链表，最大容量为整数最大值，可看做容量无限

- **SynchronousQueue**

没容量的无缓存阻塞队列

- SynchronousQueue没有容量，是无缓冲等待队列，是一个不存储元素的阻塞队列，会直接将任务交给消费者，必须等队列中的添加元素被消费后才能继续添加新的元素,这样一个过程是一个配对过程.
  - SynchronousQueue使用CAS实现线程的安全访问,并不依赖AQS,底层实现使用TransferQueue队列(单向链表)  
执行ThreadExecutor时核心线程数为0,最大线程数为Integer.MAX,60s超时,执行execute时,由于corePoolSize为0,所以会继续向队列offer数据, TransferQueue.transfer()直接返回null,导致出发创建非核心的工作线程,且线程执行完成后可在60s内复用线程
  - 使用公平队列时,采用TransferQueue,FIFO  
非公平时采用TransferStack
  - new SynchronousQueue(true)时,创建new eTransferQueue(),并创建虚拟头节点  
QNode h = new QNode(null, false)
  - queue =new SynchronousQueue(false)时,创建new TransfeStack(),不创建虚拟头节点  
queue.offer()时,返回null,offer失败
  - SynchronousQueue.offer()不会创建任何节点，最终返回false

- **操作**

- **入队**

- add(E e): 内部调用offer，若队列满时抛出IllegalStateException("Queue full") **【异常】**

- offer(E e): 如果队列没满, 立即返回true; 如果队列满了, 立即返回false 【不阻塞】
- put(E e): 如果队列满了, 一直阻塞, 直到队列不满或者线程被中断 【阻塞】
  - put方法总结
    - ■ LinkedBlockingQueue不允许元素为null。
    - ■ 同一时刻, 只能有一个线程执行入队操作, 因为putLock在将元素插入到队列尾部时加锁了
    - ■ 如果队列满了, 那么将会调用notFull的await()方法将该线程加入到Condition等待队列中。await()方法就会释放线程占有的锁, 这将导致之前由于被锁阻塞的入队线程将会获取到锁, 执行到while循环处, 不过可能因为由于队列仍旧是满的, 也被加入到条件队列中。
    - ■ 一旦一个出队线程取走了一个元素, 并通知了入队等待队列中可以释放线程了, 那么第一个加入到Condition队列中的将会被释放, 那么该线程将会重新获得put锁, 继而执行enqueue()方法, 将节点插入到队列的尾部
    - ■ 然后得到插入一个节点之前的元素个数, 如果队列中还有空间可以插入, 那么就通知notFull条件的等待队列中的线程。
    - ■ 通知出队线程队列为空了, 因为插入一个元素之前的个数为0, 而插入一个之后队列中的元素就从无变成了有, 就可以通知因队列为空而阻塞的出队线程了。
- offer(E e, long timeout, TimeUnit unit): 在队尾插入一个元素,, 如果队列已满, 则进入等待, 直到出现以下三种情况:
  - 【阻塞】
  - 被唤醒
  - 等待时间超时
  - 当前线程被中断
- 出队
  - remove
    - remove()方法用于删除队列中一个元素, 如果队列中不含有该元素, 则抛出异常; 有的话则删除并返回该元素(实际调用poll())。入队和出队都是只获取一个锁, 而remove()方法需要同时获得两把锁fullLock(); 其他对队列 contains(),toString()等都是fullLock()
    - 【异常】
  - poll(): 如果没有元素, 直接返回null; 如果有元素, 出队 【不阻塞】
  - take(): 如果队列空了, 一直阻塞, 直到队列不为空或者线程被中断 【阻塞】

- take方法总结
  - 当队列为空时，就加入到notEmpty(的条件等待队列中，当队列不为空时就取走一个元素，当取完发现还有元素可取时，再通知一下自己的伙伴（等待在条件队列中的线程）；最后，如果队列从满到非满，通知一下put线程。
  - poll(long timeout, TimeUnit unit): 如果队列不空，出队；如果队列已空且已经超时，返回null；如果队列已空且时间未超时，则进入等待，直到出现以下三种情况：【阻塞】  
被唤醒  
等待时间超时  
当前线程被中断
  - 检索当前队列头元素  
(检索,不remove)
    - E element(),实际调用peek(),值为空时【抛异常】
    - E peek(),值为空时,返回null【不抛异常】
  - J.U.C下主要内容: 线程池(ThreadPoolExecutor), 阻塞队列(ArrayBlockingQueue,LinkedBlockingQueue),ReentrantLock,线程同步的CountDownLatch,CircleBarrier(循环屏障),Semaphore(信号量),ConcurrentHashMap,ForkJoin

## JDK工具包

- Array 和 ArrayList  
List , LinkedList
  - Array
    - 优点：在内存中是连续的，速度较快，操作简单。
    - 缺点：定义数组时要定义其长度，不是很灵活，过长过短都会造成问题。不方便进行数据的添加、插入和移除
  - Array与ArrayList的区别
    - Array可以包含基本类型和对象类型，ArrayList只能包含对象类型。  
Array数组在存放的时候一定是同种类型的元素。ArrayList就存储Object[]对象数组
    - Array大小是固定的，ArrayList的大小是动态变化的。
    - ArrayList提供了更多的方法和特性，比如：addAll(), removeAll(), iterator()等等。
    - 对于基本类型数据，集合使用自动装箱来减少编码工作量。但是，当处理固定大小的基本数据类型的时候，这种方式相对比较慢。
  - ArrayList 内部为 Object[]
    - new ArrayList(Collection c)  
将c转换为数组保存到内部数组对象中，若c中值不为Object[] 则，Array.copyOf为Object[]
    - 优点：命名空间System.Collections下的一部分。大小是动态扩充与收缩的。  
在声明ArrayList对象时不需要指定它的长度。ArrayList继承了IList接口，可以



很方便的进行数据的添加、插入和移除。

- 缺点：当向集合插入不同类型的数据后（ArrayList将数据当作object存储），在进行数据处理时容易出现类型不匹配的错误，使用时需要进行类型转换处理，存在装箱与拆箱操作，造成性能大量损耗的现象。
- 1.8 ArrayList 创建时默认大小为0，当加入元素后,默认大小为10,其后，默认增长数组(当前数据量)的1/2大小(即扩展到原大小的1.5倍),  
每次add时的最小size为当前数据量大小+1,  
当新容量<最小大小时,新容量设置为最小大小值。  
最大容量为Integer.MAX\_VALUE;  
数据插入时机:先扩容,再插入数据  
1.7版本ArrayList在新建时创建对象数组,其他同1.8

```
■ DEFAULT_CAPACITY = 10;
public boolean add(E e) {
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e;
    return true;
}
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

- List为接口  
public interface List extends Collection
- LinkedList内部为链表Node  
每个节点存储Prev,Next元素索引  
双向链表
- ArrayList和LinkedList的区别
  - ArrayList和LinkedList都实现了List接口，他们有以下不同点：
  - 1.ArrayList是基于索引的数据接口，它的底层是数组。它可以以O(1)时间复杂度对元素进行随机访问。与此对应，LinkedList是以元素列表的形式存储它的数据，每一个元素都和它的前一个和后一个元素链接在一起，在这种情况下，查找某个元素的时间复杂度是O(n)。
    - get(i)  
ArrayList 直接访问数组element[i]  
LinkedList 需要循环访问到节点i
  - 2.相对于ArrayList，LinkedList的插入，添加，删除操作速度更快，因为当元素被添加到集合任意位置的时候，不需要像数组那样重新计算大小或者是更新索引。

- add(i,E)  
ArrayList插入元素时，将i位置的元素在数组中向后移动一位，再将E添加到i位置  
LinkedList 插入元素时，先获取到i位置的元素，  
然后添加新元素到i位置
    - 3.LinkedList比ArrayList更占内存，因为LinkedList为每一个节点存储了两个引用，一个指向前一个元素，一个指向下一个元素。
- HashMap 、  
LinkedHashMap  
和  
HashTable
  - java.util.HashMap(1.8+)
    - 非线程安全  
HashMap继承自AbstractMap(implements Map)类， implements Map。  
可以用Collections.synchronizedMap(HashMap map)方法使HashMap具有同步的能力。
    - 并发下的问题
      - (1)多线程扩容，引起的死循环问题 (1.8已解决),因为在扩容时,导致链表数据相互引用,导致死循环
      - (2)多线程put的时候可能导致元素丢失
      - (3)put非null元素后get出来的却是null
    - 无序，根据key的hashCode值存储数据  
key允许null,且只有1个null key  
null key在HashMap第0个位置
    - 数据结构: 数组+链表(Node<K,V>[] table) 或 数组+红黑树（在JDK1.8中如果链表长度大于8的时候才转换为红黑树，平常不是），链表为单向链表  
数组默认大小16,数组每个位置叫做一个Bucket  
若hash(key)相同时,获取到当前hash位置存储的链表，若链表长度  
 $\geq \text{TREEIFY\_THRESHOLD}(8)$  并且 数组长度 $>64$  ( $n = \text{tab.length}$ )  $<$   
 $\text{MIN\_TREEIFY\_CAPACITY}(64)$  时,将链表转为红黑树处理;  
当 红黑树的长度小于6时 ( $\text{UNTREEIFY\_THRESHOLD}(6)$ )，转换为链表存储。  
HashMap底层是通过链表来解决hash冲突的。
      - 为什么链表长度为8时,才转换为红黑树：
  - 随机hashCode算法下所有bin中节点的分布频率会遵循泊松分布，
  - 一个bin中链表长度达到8个元素的概率为0.00000006
  - more: less than 1 in ten million
    - 为什么链表长度为6时，由红黑树转换为链表：
  - 当链表长度为6时 查询的平均长度为  $n/2=3$   
红黑树为  $\log(6)=2.6$   
为8时： 链表  $8/2=4$   
红黑树  $\log(8)=3$

中间有个差值7可以防止链表和树之间频繁的转换

- 扩容：默认下 数组大小为16，当元素超过 $160.75=12$ 时.会扩容为 $216=32$   $Cap < 1$ ，即扩容1倍

capacity 的容量大小是 2 的 n 次幂

衡量HashMap是否进行Resize的条件如下：

$HashMap.Size \geq Capacity * LoadFactor$

- HashMap扩容：

HashMap扩容时,先创建新大小的链表数组(大小为原来的2倍,容量为2的n次幂),然后循环遍历旧链表数组,将旧链表的数据重新Hash( $e.hash \& (newCap - 1)$ )到新数组上,

1.当链表只有1个元素时,直接rehash到新数组

2.当链表为红黑树时,做数的数据转换

- 当链表多于1条数据时,进行链表处理,最终链表会在原来的位置或2倍原来的位置处保存  
1.7时的resize后元素顺序变成了返序,1.8时为正序

数据插入时机:第一次插入时,数组为空,需要先初始化数组(即先扩容),再插入数据;之后时,先插入数据再扩容

- hash值计算方式:  $(h = key.hashCode()) \wedge (h \ggg 16)$

key为null时为0

hashCode 的高 16 位不变, 低 16 位与高 16 位做一个异或。

- modCount: HashMap 在结构上被修改的次数, 结构修改是指改变HashMap中映射的次数, 或者以其他方式修改其内部结构(例如, rehash)。此字段用于使HashMap集合视图上的迭代器快速失败(fail fast)。(著名的ConcurrentModificationException便与此有关)

- 遍历过程中不能修改HashMap,否则会抛出

ConcurrentModificationException

因为  $modCount \neq expectedModCount$

- index计算:  $i = (n - 1) \& hash$

$= hash \% length$

- JDK1.8 修改

- 数据结构:

数组+链表 为

数组+红黑树

- 优化了hash算法

$h \wedge (h \ggg 16)$

- 扩容后, 元素要么是在原位置, 要么是在原位置再移动2次幂的位置, 且链表顺序不变。

- 遍历方式Iterator

- 1.7与1.8的区别

- 1.数组复制插入数据方式不同:

JDK1.7采用头插法,因为1.7中采用单链表进行纵向延伸,采用头插法会容易出现逆序且环形链表死循环问题(以及复制后丢数据问题);

JDK1.8采用单链表和红黑树,使用尾插法,能够避免出现逆序和死循环问题

新元素索引的计算方式1.7和1.8是相同的都是 $hash \& (length - 1)$

- 1.7put数据和多线程环境下链表复制死循环问题:

1.7put时,计算key hash,然后获取index

`int i = indexOf(hash, table.length);`

`int hash = hash(key);`

然后检索是否已存在数据,存在则更新,

然后创建Entry链表,创建时,先扩容,再插入数据;

扩容时,创建2倍大小的新数组,然后再将旧数组链表中的数据转移到新数组链表中;

链表数组转移时,节点从头到尾遍历,按新元素索引计算方式重新计算节点所在数组索引,然后将当前节点保存到新数组链表中,由于从头到尾遍历,所以新数组中的元素位置会出现倒排序(旧数组链表中先进入的数据会被插入到新数组链表的尾部,新链表头部为旧链表的尾部),

然后把旧数组替换为新数组(扩容,插入新数组链表时,使用头插法,导致新数组链表内节点顺序与旧数组链表顺序相反),

最后重新计算threshold阈值

其中:1.遍历链表时,首先将当前节点的next节点临时保存为next,

Entry<K,V> next = e.next;

- 然后将当前节点的next节点保存到新数组链表的位置newTable[i]:

e.next = newTable[i];

3.然后再将新数组链表位置节点替换为当前节点e:

newTable[i] = e;

4.最后将节点e设置为链表下一个节点,直到结束.

在多线程环境下,如果多个线程同时在操作resize扩容,

1,2线程都处在(假设有A-B-C 个节点)遍历到旧数组链表的B节点,新数组链表已存在A节点时,

当1线程在2操作结束后,失去cpu时间片,线程2此时开始执行1操作,获取到的节点e,以及节点e.next,这时e节点的next节点指向的是新数组链表中的元素A,然后线程2正常进行这一轮的剩余步骤,此时线程2中旧数组元素B.next为A,新数组元素A所在链表的节点为B,然后线程2进行下一轮复制,这时的e为A节点,A.next=null,接着设置A.next=新数组节点,即B节点,再将新数组链表节点设置为A,至此,A和B节点的链表形成了环表,在之后的get方法中,若匹配数据不是A或B时,会一直死循环A,B节点,最终导致CPU100%

<https://www.jianshu.com/p/4d1cad21853b>

- 1.8扩容流程:

1.8使用尾插法,将链表按原顺序保存到新的链表中,

为减少在计算index的消耗,1.8使用hash&oldCap(旧容量大小)来确定元素位置,即元素可能在原来的位置或2倍原来的位置.

流程:

1.首先计算新容量大小及新阈值,在小余Integer.MAX\_VALUE的容量情况下,新容量为旧容量的2倍;

2.循环数组,当数组节点不为空时,循环节点链表,当旧链表中只有1个节点时,直接插入新数组位置;当链表为红黑树时,转换为红黑树处理,最后遍历链表,采用头插法,将链表按hash&oldCap==0分为2个链表,分别放在新数组的原索引位置和新数组的原索引+原容量大小位置,直到完成

- 2.扩容后数据存储位置的计算方式不同:

1.7为先扩容,后插入,1.7时直接用hash&(length-1),为了减少Hash碰撞次数,所以设置扩容为2的幂次;

1.8为先插入后扩容(第一次插入时,为先扩容后插入),1.8时使用扩容前的位置+扩容的大小值(即原容量大小),减少了确定位置的计算次数

- 1.7在新增或复制数组时确认数据节点位置都是使用hash & (length-1)

1.8在新增时使用hash & (length-1),在复制时使用原位置(j)或原位置+旧容量大小位置  
newTab[j] = loHead;

```
newTab[j + oldCap] = hiHead;
```

- HashMap创建流程

- 1.7创建/插入流程:

1.创建(在创建对象时,就创建了数组)

```
HashMap map = new HashMap();
```

这一步中使用默认容量16和默认加载因子0.75来初始化HashMap(若指定的容量不是2的倍数时,会将容量转换为大于容量的最小2的幂次数),同时创建链表数组,计算阈值,阈值为初始化容量最小2幂次数\*加载因子loadFactor

```
// Find a power of 2 >= initialCapacity
```

```
int capacity = 1;
```

```
while (capacity < initialCapacity)
```

```
capacity <<= 1;
```

```
threshold = (int)Math.min(capacity * loadFactor, MAXIMUM_CAPACITY + 1);
```

2.插入

```
map.put("a",1);
```

第一步检查key是否为null,为null时单独处理,将key=null的节点放到数组[0]位置的链表中;

第二步计算key的hash(使用key.hashCode再经过多次位运算)

```
h ^= k.hashCode();
```

```
h ^= (h >>> 20) ^ (h >>> 12);
```

```
return h ^ (h >>> 7) ^ (h >>> 4);
```

和计算当前值的数组索引位置(hash & (length-1))

第三步循环当前索引的数组链表,查找是否已存在key的节点,若存在,则更新返回;

第四步插入节点:

插入节点流程有2部分,第一,先检查map容量是否>=阈值,同时对应的数组链表不为空时,开始扩容(扩容流程见:1.7与1.8的区别-1.7put数据和多线程环境下链表复制死循环问题);

第二,在当前数组链表上添加新节点,使用头插法,将新节点插入链表头部;

然后put流程结束

(简述: new HashMap()创建链表数组,put时,先计算数组索引,检查是否已存在节点,再继续添加节点,然后当容量不足时,先扩容,然后再插入新节点,新节点插入到数组链表的头部,key为null时存在数组[0]的链表中)

## ■ 删除remove()

删除时,遍历key对应索引的数组链表循环检查,并剔除匹配数据

- 1.8创建/插入流程

1.创建(创建对象时,只初始化了加载因子和阈值)

```
HashMap map = new HashMap();
```

默认阈值是初始化容量的最小2的幂次数值,即tableSizeOf(初始化容量)

2.插入

```
map.put("a",1);
```

第一步检查数组table是否为空,为空表示第一次插入值,先扩容(即初始化数组);

第二步检查key对应数组索引处是否有节点存在,若不存在,则直接插入;

第三步循环检查对应数组链表,当链表只有1个节点时,且节点key与输入key相同,则修改节点值,当链表为红黑树时,进行红黑树节点处理,反之遍历数组链表,

将新节点插入到节点尾部(尾插法),当检查到有key相同的节点时,更新节点值

第四步节点插入结束后,检查容量+1>阈值,则扩容(扩容流程见:1.7与1.8的区别-1.8扩容流程);

然后put流程结束

- 多线程下修改HashMap,会出现ConcurrentModificationException

- HashMap转同步

- `java.util.Collections.synchronizedMap(map)`

- 方法内部使用synchronized同步

- LinkedHashMap extends HashMap

- 非线程安全

- 有序

- 记录的插入顺序, 在用Iteraor遍历LinkedHashMap时, 先得到的记录肯定是先插入的。在遍历的时候会比HashMap慢。有HashMap的全部特性。

- 插入顺序(默认)

- 访问顺序

- `boolean accessOrder= false;`

- put/get已存在的Entry时, 会把Entry移动到双向链表的表尾

- 数据结构: 双向链表+HashMap

- 初始化时创建了只有头结点的双向链表

- 单独维护了1个双向链表,head ,tail

- 重写了Map的

- `newNode,newTreeNode,replacementNode,replacementTreeNode,entrySet()` 等方法

- 重新实现了Entry, LinkedEntrySet,LinkedHashIterator

- HashTable

- 线程安全

- 大部分方法线程同步

- Hashtable继承自Dictionary类.implements Map

- 保存key/value数据,

- key/value都不能为空

- 实现同HashMap

- 默认容量11, 不要求底层数组容量为2 的整数次幂

- 扩容为 $old * 2 + 1$

- 数据结构: 数组+链表

- Key不能为null,因为完全使用Key的hashCode

- hash值计算方式: `key.hashCode()`

- `Entry<K,V> tables index : (hash & 0x7FFFFFFF) % tab.length`

- 遍历方式: Enumeration

- TreeMap

- TreeMap

- extends `AbstractMap<K,V>`

- implements `NavigableMap<K,V>`

- 非线程安全

- 有序且可以进行排序
 

TreeMap能够把它保存的记录根据键排序，默认是按升序排序，也可以指定排序的比较器。当用Iterator遍历TreeMap时，得到的记录是排过序的。

TreeMap的键和值都不能为空。

保证了 key 的大小排序性

默认按key的自然序排序

可指定Comparator排序

  - 添加子节点时，若无根节点，则该节点为黑色根节点；  
若存在根节点，则循环查找新节点的存储位置；  
找到位置后，处理红黑树，默认设置节点为红色
- 数据结构：红黑数
  - 红黑树二叉树而言我们必须增加如下规则：
    - 1、每个节点都只能是红色或者黑色
    - 2、根节点是黑色
    - 3、每个叶节点（NIL节点，空节点）是黑色的。
    - 4、如果一个结点是红的，则它两个子节点都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。
    - 5、从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。
  - 红黑树的操作
    - 左旋
    - 右旋
    - 着色
  - 根节点为黑色
  - 若父节点为黑色时，子节点插入时为红色
- HashSet 和 TreeSet
  - HashSet
    - 继承于AbstractSet implements Set
    - 默认为HashMap实现，值为Map Key,Map Value为 Object
  - LinkedHashSet
    - extends HashSet implements Set  
由LinkedHashMap实现
  - TreeSet
    - 继承于AbstractSet implements NavigableSet(extends SortedSet)  
默认为TreeMap实现，值为Map Key,Map Value为 Object
- java.util.concurrent.ConcurrentHashMap
  - 支持并发的HashMap,Key和Value都不能为空;  
HashMap.capacity容量是指数组的大小;  
HashMap.size是指map中保存的元素个数(等于数组的大小),当发生hash碰撞时,一个数组的链表上有多个节点,这时,数组的某个位置可能是null;  
HashMap.threshold是指容量的阈值;  
计算扩容时,根据size和threshold判断是否需要扩容;
- jdk7

- 原理:

ConcurrentHashMap采用Segment分段锁,维护一个Segment分段数组,Segment extends ReentrantLock并实现HashMap的操作,每个Segment操作进行加锁(独占)tryLock()保证线程安全;  
Segment内实现了HashMap(数组+链表)的操作:即维护了Entry数组,threshold,put,扩容等操作;  
ConcurrentHashMap的初始容量,与concurrentLevel整除后得到的数,再取最小的大于等于该数的2的幂次的数作为每个Segment的初始容量,之后每个Segment单独扩容各自容量;  
维护的Segment数组:大小由concurrencyLevel控制,即并发数,默认大小16,理论上同时支持16个线程同时操作在不同的Segment上,该值初始化后不可更改;  
#实例化时,创建Segment数组,并初始化第0个Segment,之后Segment数组的其他对象根据第0个Segment的配置生成;  
ConcurrentHashMap key,value不允许为空,默认容量16;  
Segment.put时,先获取到锁,然后再遍历,创建(头插法)/更新节点,修改数组对象,获取数组对象时,使用Unsafe中的方法以及CAS来保证线程安全;并在更新数组链表节点前进行双倍容量大小扩容;

- ConcurrentHashMap创建:

1.校验loadFactor,initialCapacity,concurrencyLevel

- 处理concurrencyLevel,若超过MAX\_SEGMENTS( $2^{16}$ ),则设置为MAX\_SEGMENTS(最大分段数),  
默认分段数为16

- 分段大小设置为最小大于concurrencyLevel的2的幂次数,再将容量按分段数平均分配,取大于平均值的最小2的幂次数即为每段的容量数;

同时创建分片数组及第一个分片对象,第一个分片对象创建链表数组

#创建时即建立新容量大小的数组,默认容量为16,分段数默认16,分段内最小数组大小为2

#Segment extends ReentrantLock

#每个分段内的数组容量最大为 $2^{30}$ 次

- ConcurrentHashMap put(putVal)流程:

- 通过key的hashCode计算到key所在的segment片段下标,使用Unsafe.getObject()获取当前下标的segment对象,若对象不存在,则ensureSegment获取分段对象

1.1. ensureSegment获取分段对象

先通过Unsafe.getObjectVolatile从分段数组中获取当前下标对应的分片对象,若对象为空,则根据第0位置的分片对象segments[0]的容量,加载因子创建新的链表数组,及新的片段对象,并设置到分片数组中;该过程中获取分段数组中的分段对象及设置对象到数组中都是使用Unsafe的CAS方法

2.执行segment.put方法

该方法中先tryLock(),获取到锁后,再获取当前key对应当前分段链表数组的链表对象,然后循环链表,当key已存在时,根据onlyIfAbsent更新数据,当key不存在时,使用头插法,创建新节点并插入到链表头,

#然后校验是否需要扩容

if (c > threshold && tab.length < MAXIMUM\_CAPACITY)

rehash(node);

并更新对应数组的链表对象;

结束时,unlock();



- 扩容(ConcurrentHashMap.rehash(node)):  
同HashMap,创建2倍大小旧容量的新数组,重新计算threshold  
新节点使用头插法插入链表,并重新设置数组中的链表对象

- jdk8
  - 原理:  
JDK1.8的ConcurrentHashMap不再使用Segment实现,而是重新维护了Node[] table,使用sizeCtl控制初始化,resizing和下次resizing的值,使用数组+链表和数组+红黑树实现;  
其中大量使用CAS和Unsafe.getObjectVolatile来更新/获取数组节点,并使用synchronized对节点做线程安全控制;  
在treeifyBin方法中做扩容操作tryPresize();
  - 最大容量 $2^{30}$ (同HashMap)  
MAXIMUM\_CAPACITY = 1 << 30;
  - ConcurrentHashMap创建:  
创建时只设置capacity容量,不创建数据,默认容量为16  
容量大小是2的倍数
  - ConcurrentHashMap put(putVal)流程:
    - 判断key,value不为空,为空抛出NPE
    - 根据key.hashCode()计算key在table数组的下标
    - 若当前table大小为0,则先初始化table, initTable,其中利用Unsafe.CAS做同步;  
table大小不为0时,判断当前数组位置的链表是否为空,若为空,则新建节点;此处获取数组位置的链表和新增节点到链表都是使用Unsafe.CAS获取和设置;  
3.1 若当前节点的Hash为MOVEDOUBLE时,指当前HashMap正在resize(),单独处理该节点  
4.其他情况下,使用synchronized(f)对当前节点加锁,遍历当前位置的链表,使用尾插法添加新值节点,当链表长度达到8时转换为红黑树

## JDK8新特性

- Java语言新特性
  - Lambdas 与 Functional 接口
    - Lambdas
      - Lambda允许把函数作为参数传到方法中或把代码看成数据
      - Lambda 可以引用类的成员变量和局部变量, 如果这些变量不是final的话, 会被隐式转换为final  
(匿名类调用所在类的成员变量和局部变量时,需要将传入的变量转换为final,防止成员变量和局部变量在匿名类中被修改。)  
Lambda最终会动态生成匿名内部类(但不会生成.class文件,反射可以查看类信息), 类名如: LambdaTest\$\$Lambda\$1/4036432  
->(箭头)函数
      - Lambda可以返回一个值, 若Lambda的函数体只有1行的话, return可以省略。
      - java.util.function.Consumer;  
accept(T t)
      - 如: Arrays.asList("a","b","c").forEach(e->System.out.println(e));

或

```
Arrays.asList("a","b","c").forEach(e-> {  
    System.out.println(e);  
}  
);
```

- Functional

- 为了友好支持Lambda，增加了函数式接口概念  
函数式接口可以隐式转换为Lambda表达式：  
一个接口中只有1个抽象方法的接口称为函数式接口

- @FunctionalInterface 标注函数式接口

- 默认方法与静态方法不影响函数式接口的定义

- java.util.function包  
常用的函数式接口

- Consumer

接受一个输入参数，无返回值

- T apply(T t)

- Int,Long,Double,ObjInt... Consumer

- Consumer consumer = (x) ->  
System.out.println("consumer: " + x);  
consumer.accept("Hello");

- Supplier

无输入参数，返回一个结果

- T get()

- Boolean,Int,Long,Double...Supplier

- Supplier supplier = () -> "Test supplier";  
supplier.get();

- Predicate

接受一个输入参数，返回布尔值

Predicate 是一个可以指定入参类型，并返回 boolean 值的函数式接口。它内部提供了一些带有默认实现的方法，可以被用来组合一个复杂的逻辑判断 (and, or, negate)

- and(Predicate<? super T> other)

negate(Predicate<? super T> other)

- Predicate predicate = (x) -> x.length() > 0;  
predicate.test("String");

- Function

接受一个输入参数，返回一个结果

Function 函数式接口的作用是，我们可以为其提供一个原料，他给生产一个最终的产品。通过它提供的默认方法，组合,链行处理 (compose, andThen):

- R apply(T t)

- Function<Integer, String> function1 = (x) -> "result: " +

- x;
    - function1.apply(6);
  - //比较接口  
java.util.Comparator
- 接口的默认方法与静态方法
  - 接口default标识符  
default void run()
    - 如:java.lang.Iterable实现的forEach,spliterator,stream等
    - 不需要子类实现, 子类可重写
  - 接口static 标识符 声明静态方法  
static void runS()
- 方法引用  
(直接引用已有Java类或对象(实例)的方法或构造器)
  - Class::new 构造器引用
    - final Car car = Car.create(Car::new);  
final Car car2 = Car.create(() -> new Car());
  - Class::static\_method 静态方法引用
    - cars.forEach(Car::collide);  
cars2.forEach((c)->Car.collide(c));
  - Class::method 特定类的任意方法引用
    - cars.forEach(Car::repair);  
cars2.forEach((c)->c.repair());
  - instance::method 特定类的任意对象的方法引用
    - cars.forEach(Car.create(Car::new)::follow);  
cars.forEach(police::follow);  
cars2.forEach((c)->police.follow(c));
- 重复注解  
@Repeatable
  - @Repeatable(Filters.class)  
//value必须为注解数组类  
//Filters必须与Filter的注解相同  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Filters{  
Filter[] value();  
}  
@Target(ElementType.TYPE)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Filter{  
String value();  
}  
■ class.getAnnotationsByType(Filters.class)  
获取相同类型的注解,返回Filter注解数组

```
Filter[] = [@Filter1,@Filter2];
```

- `class.getDeclaredAnnotations()`

获取到的注解中若存在@Repeatable注解，则获取到的注解为注解数组，即  
`Filters(value=[@Filter1,@Filter2])`

- 更好的类型推测机制

- 扩展注解

可以为任何东西添加注解:

局部变量、泛型类、父类、接口类的实现、方法的异常

- @Target

`ElementType.TYPE_USE`

`ElementType.TYPE_PARAMETER`

- ```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.util.ArrayList;
import java.util.Collection;
public class Annotations {
    @Retention( RetentionPolicy.RUNTIME )
    @Target( { ElementType.TYPE_USE, ElementType.TYPE_PARAMETER } )
    public @interface NonEmpty {
        public static class Holder< @NonEmpty T > extends @NonEmpty object {
            public void method() throws @NonEmpty Exception [
        }
    }
    @SuppressWarnings( "unused" )
    public static void main(String[] args){
        final Holder< String > holder = new @NonEmpty Holder< string >();
        @NonEmpty Collections @NonEmpty String > strings = new ArrayListo0;
    }
}
```

- Java 类库新特性

- `java.util.Optional`

null值容器类

- `//map`对当前Optional值进行转换，`orElse`: 设置默认值  
`fullName.map(s->"Hey "+s+"!").orElse("Default");`
- `Optional.empty()` 创建空的Optional对象
  - `get()` 会抛出`NoSuchElementException`
- `Optional.of(obj)` 创建一个包含obj的Optional对象
  - `obj`不能为null,否则会抛出NPE
- `Optional.ofNullable(obj);` 创建一个包含obj的Optional对象
  - `obj`可以为null, 为null时同`Optional.empty()`
- `of.get()` 获取Optional包含的值
- `of.isPresent()` 判断是否为空

- `of.ifPresent(t-> System.out.println(t))` 若Optional包含值不为空时，提供一个Consumer
- //为空时提供回调方法(Supplier)产生默认值  
`of.orElseGet(()->"[none]")`
  - `of.orElseGet(Object::new)`
  - `orElseGet` 因为使用了Supplier,所在传值时不会计算Supplier中的值  
 所以当of不为空时，`orElseGet`中传入的值不计算
- //为空时提供默认值  
`of.orElse("Default");`
  - 因为`orElse`的参数为具体对象或值，所以在传入参数时，会计算参数结果，导致不管of是否为空，都会执行参数计算
- //为空时返回自定义异常  
`of.orElseThrow(Supplier t)`
- //值转换， 返回一个Optional对象，传入函数值为任何值  
`Optional map(Function<? super T, ? extends U> mapper)`  
`of.map(t -> t).orElseGet(()->"t")`
- //值转换， 返回一个Optional对象，传入值需为Optional对象  
`Optional flatMap(Function<? super T, Optional> mapper)`
- //值过滤， 若值不满足条件，则返回Optional.empty();  
`of.filter(t -> t != null).orElse("null");`
- `java.util.Stream`  
 Stream 是对集合（Collection）对象功能的增强，它专注于对集合对象进行各种非常便利、高效的聚合操作（aggregate operation），或者大批量数据操作 (bulk data operation)  
 简化了集合对象的处理  
 优点:  
 减少了迭代次数，也避免了存储中间结果
  - 生成Stream
    - `Stream.generate(Obj);`
    - Collection 和数组
      - `Collection.stream()`
      - `Collection.parallelStream()`
      - `Arrays.stream(T array) or Stream.of()`
    - `BufferedReader`
      - `java.io.BufferedReader.lines()`
    - 静态工厂
      - `java.util.stream.IntStream.range()`
      - `java.nio.file.Files.walk()`
    - 自己构建
      - `java.util.Spliterator`
      - `StreamSupport.stream(new Spliterator() {})`
    - 其它

- Random.ints()
  - BitSet.stream()
  - Pattern.splitAsStream(java.lang.CharSequence)
  - JarFile.stream()
- 流的操作类型分为两种：
  - Intermediate: (中间操作)一个流可以后面跟随零个或多个 intermediate 操作。其目的主要是打开流，做出某种程度的数据映射/过滤，然后返回一个新的流，交给下一个操作使用。这类操作都是惰性的 (lazy)，就是说，仅仅调用到这类方法，并没有真正开始流的遍历。
    - map、mapToInt、mapToLong、mapToDouble、flatMap、flatMapToInt、flatMapToLong、flatMapToDouble、filter、distinct、sorted、peek、limit、skip、parallel、sequential、unordered
  - Terminal: (终止操作)一个流只能有一个 terminal 操作，当这个操作执行后，流就被使用“光”了，无法再被操作。所以这必定是流的最后一个操作。Terminal 操作的执行，才会真正开始流的遍历，并且会生成一个结果，或者一个 side effect。
    - Stream.collect()
    - Stream.count()
    - Stream.min()
    - Stream.max()
    - Stream.forEach()
    - Stream.forEachOrdered()
    - Stream.reduce()
    - Stream.toArray()
    - 短路操作(short-circuiting operations)  
如果一个中间操作 (intermediate operation) 的输入是无限的，而它可以产生一个有限stream结果，那么这个中间操作是短路的 (short-circuiting)；
      - Stream.allMatch()
      - Stream.anyMatch()
      - Stream.noneMatch()
      - Stream.findAny()
      - Stream.findFirst()
  - 有状态的操作 (Stateful operations)
    - distinct() sorted() sorted() limit() skip()
  - IntStream、LongStream、DoubleStream
    - summaryStatistics  
统计  
IntStream.builder().build().summaryStatistics();
    - 为什么单独提供IntStream,LongStream,DoubleStream
  - //将Stream中的元素映射为另一种类型  
Stream.map()

- //将Stream中的元素映射为另一种类型  
 //flatMap是一种扁平化、一对多的操作，会返回一个新的Stream，因此  
 flatMap实现需要返回一个新的Stream对象  
 Stream.flatMap()  
 flatMap(e -> Stream.of(e.split(" ")))
  - Arrays.asList("Stream operations is a new API in Java 8")  
 .stream()  
 .flatMap(e -> Stream.of(e.split(" ")))  
 .forEach(System.out::println);
- //规约操作（reduction operation），也称作折叠（fold）。可以通过相同的  
 合并方法将系列的输入组合成一个结果。  
 Stream.reduce
  - //函数接口BinaryOperator accumulator称为累加器，需满足结合律，  
 其抽象方法为T apply(T t1, T t2); 有两个参数和返回值，且他们的类型相  
 同。t1为之前结合的结果，t2为当前元素，该抽象方法的实现需要将t1和  
 t2结合，并返回结果。  
 Optional reduce(BinaryOperator accumulator)
  - T reduce(T identity, BinaryOperator accumulator) 跟1) 相似，只不过  
 需要制定计算时的初始值，
- Stream.parallelStream()  
 与  
 Stream.stream() 实现区别
  - parallelStream  
 并行流
    - stream.collect()执行并行的条件
      - isParallel() 指定为并行操作  
 同时  
 collector对象支持并行操作  
 同时  
 (非排序 || collector对象指定非排序)  
 isParallel()  
 &&  
 (collector.characteristics().contains(Collector.Characteristic  
 s.CONCURRENT))  
 && (!isOrdered() ||  
 ollector.characteristics().contains(Collector.Characteristics.  
 UNORDERED))
    - 使用ForkJoinTask实现并行
  - stream  
 串行流
- 流的基本特点
  - 集合是对一组特定类型的元素值序列提供的接口 是数据结构,提供了元素  
 的存取
  - 流也是对一组特定类型元素值序列提供的接口,在于计算,提供了对元素序

列的操作计算方式 比如 filter map等

- 流只能运行一次
- 流由源 0个或者多个中间操作以及结束操作组成
- 流操作的方法基本上是函数式接口的实例
- 流的中间操作是惰性的并不会立即执行 这更有利于内部迭代的优化
- 流借助于它内部迭代特性提供了声明式的编程方式 更简洁
- 中间操作本身会返回一个流,可以将多个操作复合叠加,形成一个更大的流水线
- 流分为顺序和并行两种方式
- Stream的每个操作被抽象为1个stage,所有操作组成了一个双向链表,这些Stream对象以双向链表的形式组织在一起, 构成整个流水线, 由于每个Stage都记录了前一个Stage和本次的操作以及回调函数, 依靠这种结构就能建立起对数据源的所有操作
- 每个stage被1个Sink包裹, 为每个操作具体的行为操作, 也叫做回调
  - 一个sink有两种状态,初始/激活
  - 开始时是初始状态,begin 激活, end使之回到初始状态,可以重复利用
  - accept只能在激活状态使用
  - Sink用于协调相邻的stage之间的数据调用
  - 通过begin end accept方法 以及cancellationRequested短路标志位来控制处理流程,对数据进行管控
- Sink.begin(size)
- Sink.end()
- Sink.cancellationRequested()
- Sink.accept
  - void begin(long size) 开始遍历元素之前调用该方法, 通知Sink做好准备。
  - void end() 所有元素遍历完成之后调用, 通知Sink没有更多的元素了。
  - boolean cancellationRequested() 是否可以结束操作, 可以让短路操作尽早结束。
  - void accept(T t) 遍历元素时调用, 接受一个待处理元素, 并对元素进行处理。Stage把自己包含的操作和回调方法封装到该方法里, 前一个Stage只需要调用当前Stage.accept(T t)方法就行了。
- Date/Time API(JSR 310)
- java.time
  - Clock clock = Clock.systemUTC();
  - //基于Clock实现,只返回ISO-8601且无时区信息的日期部分
  - LocalDate localDate = LocalDate.now();
  - //2019-12-12
  - //基于Clock实现,只返回ISO-8601且无时区信息的时间部分
  - LocalTime localTime = LocalTime.now();
  - //11:24:44.153
  - //基于Clock实现,只返回ISO-8601且无时区信息的日期+时间部分
  - LocalDateTime localDateTime = LocalDateTime.now();
  - //2019-12-12T11:24:44.153



- JavaScript引擎Nashorn
- java.util.Base64
  - Base64.getEncoder().encode(byte[])
  - Base64.getDecoder().decode(String)
- 并行(parallel)数组
  - Arrays.parallelSort()
- 并发(Concurrency)
  - ConcurrentHashMap新增方法支持聚合操作
- Java 编译器新特性
  - 参数名字  
javac -parameters启用  
Parameter.getName()获取
    - 不使用 -parameter 编译后获取的参数为  
arg0  
使用后获取为args
- Java虚拟机新特性
  - PermSpace (永久代)移除, 新增MetaSpace  
-XX:MetaSpaceSize  
-XX:MaxMetaSpaceSize
- 新增Java工具
  - Nashorn引擎:jjs
    - //接收参数为js文件,返回执行的结果  
jjs file.js
  - 类依赖分析器: jdeps
    - 接收.class文件,目录,jar作为输入, 结果输出到控制台

## 流式/响应式编程

- Reactor(io.projectreactor.reactor-core,JDK1.8+,Spring Webflux集成,reactor-netty)
  - Reactor是基于Reactive Stream(响应式流)规范,在JVM上构建的非阻塞式应用程序
    - reactor-core: Reactor 是完全无阻塞(full non-blocking)的,可提供有效的需求管理,可直接与JavaAPI交互: CompletableFuture,Stream和Duration
    - 输入的[0 | 1 | N]序列: Reactor 提供2种响应式(reactive)和可组合(composable)的API: Flux [N] 和Mono [0 | 1],它们广泛的实现了Reactive Extensions
    - 非阻塞IO: Reactor非常适合微服务,可为Http(包括Websocket),TCP,UDP提供背压就绪(backpressure-ready)的网络引擎(network engines),即reactor-netty
  - Reactor3参考指南:<https://projectreactor.io/docs/core/release/reference/>
    - 入门(Getting Started)
      - Reactor Core在JDK1.8+运行
      - reactor版本控制:  
Reactor3使用 BOM (Bill of Materials物料清单) model模式管理版本;  
Artifacts版本:MAJOR.MINOR.PATCH-QUALIFIER

BOM版本: YYYY.MINOR.PATCH-QUALIFIER

MAJOR: Reactor大版本号

MINOR: Reactor小版本号,从0开始,每个发布周递增

.PATCH: 基于0的数字,随服务版本递增

-QUALIFIER: 文字限定词,GA没有-QUALIFIER;

每个发布周期都被赋予一个代号,与以前的基于代号的方案相一致,可用于更非正式地引用它(例如在讨论,博客文章等中)。

代号代表传统上的MAJOR.MINOR号。它们(大多数)来自元素周期表,按字母升序排列。

- MVN安装(MVN支持BOM):

```
io.projectreactor
reactor-bom
2020.0.4
pom
import
```

```
io.projectreactor
reactor-core
```

```
io.projectreactor
reactor-test
test
```

- Reactor 采用观察者模式,对数据进行订阅,同时使用生产者/消费者消息队列(MessageQueue)模型处理数据,基本流程(以Flux.create().subscribe()举例):

- 创建序列Flux: Flux.join()/Flux.create()
- 处理FluxSink回调(有SynchronousSink,MonoSink等)  
当在2中sink.next时,BufferAsyncSink中的queue会queue.offer()数据到队列中,然后触发drain()方法,进行处理数据,期间会触发调用subscriber.onNext()方法,直至数据处理完成,等待下次next或其他的drain()调用
- 添加操作链(filter,map等)和定义事件回调(onError,onComplete等)
- 调用订阅方法,并指派订阅对象.subscribe();
- 调用subscribe方法后,开始创建sink(createSink,默认backpressure策略为BUFFER,创建一个默认256大小的无界异步队列(BufferAsyncSink),new SpscLinkedArrayQueue<>(256)),  
sink创建完成后,触发Subscriber.onSubscribe()方法,进入到  
LambdaSubscriber#onSubscribe()方法,并执行

subscription.request(Long.MAX\_VALUE);实际调用BaseSink.request()方法,调用时,会触发subscribe中的consumer,然后再执行onRequestedFromDownstream(),继续执行drain(),并在onNext()方法中调用subscribe()中consumer.accept()方法(期间会调用FluxPeekFuseable的相关方法);

简要流程:flux.create(sinkConsumer)->subscribe(Subscriber)[subscribe中做2件事: 1.

创建Sink,2: 执行SinkConsumer]->createSink[FluxCreate.subscribe]->sink.next()

[sinkConsumer]->queue.offer(t),drain()->drain()调用

Flux.onNext().onNext().onNext()...->subscriberConsumer[Subscriber]-

>sink.complete()或subscriber.dispose()[subscribe()方法执行完成后,若存在dispose方法,则会继续执行dispose()方法]->结束

- 响应式编程简介(Introduction to Reactive Programming)

- Reactor是响应式编程范例(Reactive Programming paradigm)的实现;

响应式编程是一种异步编程范例, 涉及数据流和变化的传播。这意味着可以通过所采用的编程语言轻松表达静态(例如数组)或动态(例如事件发射器)数据流。

响应式编程范例通常以面向对象的语言表示, 作为Observer设计模式的扩展。

- JVM提供了2种异步编程模型:

- 回调(Callback),异步方法没有返回值,但有个额外的callback参数(匿名内部类或lambda)
- Future,异步方法立即返回Future异步过程计算一个T值, 但是Future对象包装了对它的访问。该值不是立即可用的, 并且可以轮询该对象, 直到该值可用为止。例如, ExecutorService正在运行的Callable任务使用Future对象。

- 背压(Backpressure):上游传播的信号也用于实现背压, 我们在组装流水线中将其描述为当工作站的处理速度比上游工作站慢时, 沿生产线向上发送的反馈信号

- Reactor核心功能(Reactor Core Features )

- Reactor的主要部件是Reactor core,专注于实现Reactive Stream 规范,适用于JDK1.8+

- reactor.core.publisher.Flux: 表示包含0...N项的异步响应序列;

Flux是Publisher代表0到N个发出项目的异步序列的标准, 可以选择以完成信号或错误终止。如无流规范, 这三种类型的信号转换为呼叫到下游用户的onNext, onComplete和onError方法。

- 示例:

//创建Flux对象

```
Flux seq1 = Flux.just("foo", "bar", "foobar");
```

//创建List

```
List iterable = Arrays.asList("foo", "bar", "foobar");
```

//通过list创建Flux对象

```
Flux seq2 = Flux.fromIterable(iterable);
```

- reactor.core.publisher.Mono: 表示包含0...1项的异步响应序列;

Mono是通过信号Publisher发出最多一项然后以 信号结束(成功, 有或没有值)或仅发出单个信号(失败)的专家。onNext,onComplete,onError

- 示例:

//创建空的Mono对象

```
Mono noData = Mono.empty();
```

```
Mono data = Mono.just("foo");
```

```
Flux numbersFromFiveToSeven = Flux.range(5, 3);
```

- Subscript订阅示例:

//订阅并触发序列。

```

subscribe();
//消费每个生产的值
subscribe(Consumer<? super T> consumer);
//消费每个生产的值,并指定错误处理的消费者
subscribe(Consumer<? super T> consumer,
    Consumer<? super Throwable> errorConsumer);
//消费每个生产的值,并指定错误处理的消费者和完成的回调
subscribe(Consumer<? super T> consumer,
    Consumer<? super Throwable> errorConsumer, Runnable completeConsumer);
//消费每个生产的值,并指定错误处理的消费者,完成的回调和订阅结果
subscribe(Consumer<? super T> consumer,
    Consumer<? super Throwable> errorConsumer, Runnable completeConsumer,
    Consumer<? super Subscription> subscriptionConsumer);

```

- Disposable(一次性的,可自由支配的):

所有这些基于lambda的变体subscribe()都具有Disposable返回类型。在这种情况下,该Disposable接口表示可以通过调用其方法来取消订阅的事实dispose()。

对于Flux或Mono,取消表示信号源应停止产生元素。但是,并不能保证立即执行:某些源可能会产生如此快的元素,以至于甚至在接收到取消指令之前它们也可以完成;

该类Disposable中提供了一些实用程序Disposables。在其中:

Disposables.swap():创建一个Disposable包装器,使您可以自动取消和替换混凝土Disposable;

另一个有趣的实用程序是Disposables.composite(...):通过此组合,您可以收集多个Disposable(例如,与服务调用关联的多个进行中的请求),并在以后一次处理所有这些请求。

- Swap swap = Disposables.swap();

swap.update(disposable1);

包装一个Disposable对象,并处理

- Disposables.composite(disposable1,disposable2)

可一次处理多个disposable

- Subscribe的2种方式

- 1.Lambdas

- flux.take(1).subscribe(consumer->{

```

System.out.println("take consumer:"+consumer.toString());
});

```

- 2.BaseSubscriber(Lambdas的替代品)

可自定义控制订阅处理过程

- 示例:

//创建自定义订阅者,实现BaseSubscriber

```

SampleSubscriber ss = new SampleSubscriber();

```

```

Flux ints = Flux.range(1, 4);

```

```

ints.subscribe(ss);

```

//自定义订阅者实现类

```

package io.projectreactor.samples;

```

```

import org.reactivestreams.Subscription;

```

```

import reactor.core.publisher.BaseSubscriber;

```

```

public class SampleSubscriber extends BaseSubscriber {

```

```

public void hookOnSubscribe(Subscription subscription) {
    System.out.println("Subscribed");
    request(1);
}
public void hookOnNext(T value) {
    System.out.println(value);
    request(1);
}
}

```

- 关于背压和重塑请求 (On Backpressure and Ways to Reshape Requests), 第一个请求在订阅时来自最终的订阅者, 然而最直接的订阅方式都会立即触发一个无限的请求长最大值: Long.MAX\_VALUE

- subscribe() 及大多数基于 lambda 的变体 (Consumer 除外)
- block(), blockFirst() and blockLast()
- iterating over a tolerable() or toStream()
- APIs (Mono/Flux)
- 创建序列
  - 同步 (Synchronous): generate: Flux.generate() 同步生成序列

Flux.generate(Consumer<SynchronousSink> generator), 其中 sink 为 SynchronousSink,

是一个同步, 且为一对一的队列, 每个 sink 回调中 next() 只能调用一次

- 示例:

// 基于状态的例子 generate

```
Flux flux = Flux.generate(
```

```
// 设置状态为0
```

```
() -> 0,
```

```
(state, sink) -> {
```

```
// 使用状态来选择要发出的信号
```

```
sink.next("3 x " + state + " = " + 3*state);
```

```
// 设置停止状态
```

```
if (state == 10) sink.complete();
```

```
// 返回在下一次调用中使用的新状态 (除非序列在此调用中终止)。
```

```
return state + 1;
```

```
});
```

- 异步和多线程 (Asynchronous and Multi-threaded: create): create:

```
FLux.create(Consumer<? super FluxSink> emitter);
```

create 是程序化创建序列的一种更高级的方式, 它可以每轮多次 emit, 甚至是多线程 emit, 它暴露了 FluxSink

- Mono.create(Consumer<MonoSink> callback)

Mono 中也有 create 生成器, 但不允许多个 emit, 它会在第一个信号后丢弃其余的所有信号

- 示例:

假设使用了基于侦听器的 API。它按块处理数据并有两个事件: (1) 数据块已准备就绪, 并且 (2) 处理完成 (终端事件), 如 MyEventListener 接口所示:

```

interface MyEventListener {
    void onDataChunk(List chunk);
    void processComplete();
}

```

```
}
```

可以用来create将其桥接到Flux:

```
Flux bridge = Flux.create(sink -> {  
    myEventProcessor.register(  
        new MyEventListener() {  
            public void onDataChunk(List chunk) {  
                for(String s : chunk) {  
                    //可以多线程调用next方法  
                    sink.next(s);  
                }  
            }  
        })  
    public void processComplete() {  
        sink.complete();  
    }  
});  
});
```

- 异步和单线程( Asynchronous but single-threaded):push:

```
Flux.push(Consumer<? super FluxSink> emitter);
```

push是generate和create之间的中间地带, 适合于处理来自单个生产者的事件。它类似于create, 因为它也可以是异步的, 并且可以使用create支持的任何溢出策略来管理背压。但是, 一次只能有一个生成线程调用next、complete或error。

- 示例:

```
Flux bridge = Flux.push(sink -> {  
    myEventProcessor.register(  
        new SingleThreadEventListener() {  
            public void onDataChunk(List chunk) {  
                for(String s : chunk) {  
                    //同一时间只能有1个线程调用next方法  
                    sink.next(s);  
                }  
            }  
        })  
    public void processComplete() {  
        sink.complete();  
    }  
    public void processError(Throwable e) {  
        sink.error(e);  
    }  
});  
});
```

- 混合推/拉模型(A hybrid push/pull model):

- 示例:

```
Flux bridge = Flux.create(sink -> {  
    myMessageProcessor.register(  
        new MyMessageListener() {  
            public void onMessage(List messages) {  
                for(String s : messages) {
```

```

        sink.next(s);
    }
}
});
sink.onRequest(n -> {
    List messages = myMessageProcessor.getHistory(n);
    for(String s : messages) {
        sink.next(s);
    }
});
});

```

- push()或create()后的清理:

清理有2种回调 onDispose and onCancel:

onDispose可用于在Flux 完成， 错误排除或取消操作时执行清理。

onCancel可用于在使用进行清理之前执行特定于取消的任何操作onDispose。

onCancel 首先调用， 仅用于取消信号。

onDispose 为完成， 错误或取消信号而调用。

- 示例:

```

Flux bridge = Flux.create(sink -> {
    sink.onRequest(n -> channel.poll(n))
        .onCancel(() -> channel.cancel())
        .onDispose(() -> channel.close())
});

```

- handle:

```

Mono.create((callback)->{}).handle((obj,synchronousSink)->{});

```

```

Flux.create((emitter)->{}).handle((obj,sink)->{});

```

handle方法有点不同：它是一个实例方法，这意味着它被链接到一个现有的源上（就像常见的操作符一样）。它存在于Mono和Flux中。

handle()是对每次sink.next()后的进一步处理

- 示例:

```

Flux.create((emitter)->{
    for (int i = 0; i < 10; i++) {
        System.out.println("emit:"+i);
        emitter.next(i);
    }
    emitter.complete();
}).handle((obj,sink)->{
    System.out.println("obj:"+obj);
}).subscribe();

```

执行结果:

emit:0

obj:0

emit:1

obj:1

emit:2

obj:2

emit:3  
obj:3  
emit:4  
obj:4  
emit:5  
obj:5  
emit:6  
obj:6  
emit:7  
obj:7  
emit:8  
obj:8  
emit:9  
obj:9

- 线程和调度程序(Threading and Schedulers):

Flux或Mono并不一定意味着它在专用线程中运行。相反,大多数操作符继续在前一个操作符执行的线程中工作。除非指定,否则最顶层的操作符(源)本身在进行subscribe()调用的线程上运行

- Schedulers类具有访问以下执行上下文的静态方法

Schedulers部分实现依赖ScheduledExecutorService

- Schedulers.immediate(): 无执行上下文,在处理时,将直接执行提交的Runnable,有效地在当前线程上运行它们(可以被视为“空对象”或无操作调度器)。

- Schedulers.single(): 单一的,可重复使用的线程,此方法对所有调用方重用同一线程,直到调度程序被释放。

- Schedulers.single().createWorker().schedule();

???

- Schedulers.elastic():无限弹性线程池,存在隐藏背压问题和导致线程过多的趋势,不建议使用

- Schedulers.boundedElastic(): 有界弹性线程池,与它的前身elastic()一样,它根据需要创建新的工作池,并重用空闲的工作池。空闲时间过长(默认值为60秒)的工作池也会被释放。与它的elastic()前身不同,它对可以创建的备份线程数有一个上限(默认值是CPU内核数x 10)。在达到上限后提交的多达10万个任务将排队,并在线程变为可用时重新调度(当使用延迟进行调度时,延迟将在线程变为可用时开始)。对于I/O阻塞工作,这是一个更好的选择。Schedulers.boundedElastic()是一种方便的方法,可以为阻塞进程提供自己的线程,这样它就不会占用其他资源。

- Schedulers.parallel(): 固定线程数线程池,线程数量为CPU核数

- Schedulers.fromExecutorService(ExecutorService);将ExecutorService转换为Schedulers

- Schedulers.newXXX创建新的线程池实例,以上不带new的方法创建的线程池都是被缓存的线程池,由以下属性缓存:

```
static AtomicReference<Schedulers.CachedScheduler> CACHED_ELASTIC = new AtomicReference();
```

```
static AtomicReference<Schedulers.CachedScheduler> CACHED_BOUNDED_ELASTIC = new AtomicReference();
```

```
static AtomicReference<Schedulers.CachedScheduler> CACHED_PARALLEL = new AtomicReference();
```



```
static AtomicReference<Schedulers.CachedScheduler> CACHED_SINGLE = new AtomicReference();
```

- 不带new的xxx方法cache:

```
//SINGLE_SUPPLIER
```

```
static final Supplier SINGLE_SUPPLIER = () -> {return newSingle("single", true);};
```

```
//single方法
```

```
public static Scheduler single() {
```

```
    return cache(CACHED_SINGLE, "single", SINGLE_SUPPLIER);
```

```
}
```

```
//cache方法
```

```
static Schedulers.CachedScheduler
```

```
cache(AtomicReference<Schedulers.CachedScheduler> reference, String key,
```

```
Supplier supplier) {
```

```
    Schedulers.CachedScheduler s = (Schedulers.CachedScheduler)reference.get();
```

```
    if (s != null) {
```

```
        return s;
```

```
    } else {
```

```
        s = new Schedulers.CachedScheduler(key, (Scheduler)supplier.get());
```

```
        if (reference.compareAndSet((Object)null, s)) {
```

```
            return s;
```

```
        } else {
```

```
            s._dispose();
```

```
            return (Schedulers.CachedScheduler)reference.get();
```

```
        }
```

```
    }
```

```
}
```

- 切换上下文或调度器(switching the execution context (or Scheduler) in a reactive chain):

publishOn, subscribeOn;

publishOn在链中的位置很重要，而subscribeOn的位置则不重要

- publishOn:

publishOn的应用方式与任何其他操作相同，位于用户链的中间，它接收来自上游的信号，并在下游重放这些信号，同时对来自关联调度器的worker执行回调。因此，它会影响后续操作符的执行位置（直到另一个publishOn链接进来）；

除非它们在特定的调度程序上工作，否则发布后的操作符将继续在同一线程上执行；

- 示例:

```
//S1:创建4个线程的线程池Scheduler实例
```

```
Scheduler s = Schedulers.newParallel("parallel-scheduler", 4);
```

```
// 第一个M1:map运行在线程P5上
```

```
final Flux flux = Flux
```

```
.range(1, 2)
```

```
.map(i -> 10 + i) //M1
```

```
//publishOn在从S1中选取的线程上切换整个序列。
```

```
.publishOn(s)
```

```
//第二个M2:map运行在S1的线程中
```

```
.map(i -> "value " + i); //M2
```

```
//P5:
new Thread(() -> flux.subscribe(System.out::println));
运行结果:
parallel-scheduler-1,value: T1-11
parallel-scheduler-1,value: T1-12
parallel-scheduler-1,value: T1-13
end
```

- subscribeOn:

subscribeOn应用于订阅进程, 当反向链被构造时;因此, 无论将subscribeOn放置在链中的何处, 它都会影响源emmit的上下文。但是, 这并不影响对publishOn的后续调用行为; 它们仍然会切换后面部分链的执行上下文。

- 示例:

```
Scheduler s = Schedulers.newParallel("parallel-scheduler", 4);
final Flux flux = Flux
.range(1, 2)
//运行在s中的线程中
.map(i -> 10 + i)
//设置消费者从s中取线程,影响消费者执行线程,放在任何位置都会影响
.subscribeOn(s)
//运行在s中的线程中
.map(i -> "value " + i);
new Thread(() -> flux.subscribe(System.out::println));
运行结果:
parallel-scheduler-3,value: parallel-scheduler-3-11
parallel-scheduler-3,value: parallel-scheduler-3-12
parallel-scheduler-3,value: parallel-scheduler-3-13
end
```

- 错误处理(Handling Errors):

在ReactiveStream 中错误(error)是终止事件(terminal event),一旦发送错误,会停止序列(原始序列不会继续执行),并沿着操作符链(与错误相关定义的操作符链)传播到最后一步,即自定义的订阅者及其onError方法;

此类错误时应用级别的错误,需要始终定义Subscriber.onError()方法,若不定义,将抛出UnsupportedOperationException异常,可以使用

Exceptions.isErrorCallbackNotImplemented 检测并触发onError未定义错误

- 示例:

```
Flux.just(1, 2, 0)
.map(i -> "100 / " + i + " = " + (100 / i)) //this triggers an error with 0
//在onErrorReturn之前定义,会触发该回调,在onErrorReturn后定义,则不会触发
回调
.doOnError((e)->{
    System.out.println("doOnError");
})
//发送错误时,处理错误,并设置返回值
.onErrorReturn("Divided by zero :(")
.subscribe(t-> System.out.println("subscribe:"+t),
//定义onErrorReturn后,subscribe的errorConsumer不会被触发
```

```
(e)-> System.out.println("subscribe error:"+e.getMessage()); // error handling
```

example

执行结果:

subscribe:100 / 1 = 100

subscribe:100 / 2 = 50

doOnError

subscribe:Divided by zero :(

- 错误处理操作方式(Error Handling Operators)

所有的错误处理都是等价的

- 捕获并返回静态值(Catch and return a static default value.)

- 1. Flux onErrorReturn(T fallbackValue)

当发生错误时返回指定值

- 2. Flux onErrorReturn(Predicate<? super Throwable> predicate, T fallbackValue)

当发生错误并满足断言时,返回指定值

- 3. Flux onErrorReturn(Class type, T fallbackValue)

当发生错误并且错误类型为type时,返回指定值

- 捕获并使用fallback方法执行替代路径(Catch and execute an alternative path with a fallback method.)

- 1. Flux onErrorResume(Function<? super Throwable, ? extends Publisher<? extends T>> fallback)

当发生错误时,通过错误处理,返回新的Publisher

- 2. Flux onErrorResume(Predicate<? super Throwable> predicate, Function<? super Throwable, ? extends Publisher<? extends T>> fallback)

当发生错误并且满足断言时,通过错误处理,返回新的Publisher

- 3. Flux onErrorResume(Class type, Function<? super E, ? extends Publisher<? extends T>> fallback)

当发生错误且错误类型为type时,通过错误处理,返回新的Publisher

- 捕获并动态计算返回值(Catch and dynamically compute a fallback value.)

- 示例:

//使用onErrorResume处理,并封装处理返回值

```
erroringFlux.onErrorResume(error -> Mono.just(
```

```
    MyWrapper.fromError(error)
```

```
));
```

- 捕获,包装BusinessException,再抛出(Catch, wrap to a BusinessException, and re-throw.)

- 捕获,记录错误日志,再抛出(Catch, log an error-specific message, and re-throw.)

- 使用finally代码块清理资源或使用try-with-resource清理资源 (Use the finally block to clean up resources or a Java 7 “try-with-resource” construct.)

- Reactive finally: doFinally,最终执行(同finally):

```
Flux.just("").doFinally()
```

- 示例:

```
Stats stats = new Stats();
```

```
LongAdder statsCancel = new LongAdder();
```

```

Flux flux =
Flux.just("foo", "bar")
.doOnSubscribe(s -> stats.startTimer())
.doFinally(type -> {
    stats.stopTimerAndRecordTiming();
    if (type == SignalType.CANCEL)
        statsCancel.increment();
})
.take(1);

```

- Reactive try-with-resource: using()

- 示例:

```

Flux flux =
Flux.using(
//生产资源
    () -> disposableInstance,
//处理资源
    disposable -> Flux.just(disposable.toString()),
//释放资源
    Disposable::dispose
);

```

- 重试(Retry):

retry允许重试产生错误的序列,但它通过重新订阅上游流量来工作,并非原序列,原序列依然是终止的;

重试会重新发起订阅

- 示例:

```

Flux.interval(Duration.ofMillis(250))
    .map(input -> {
        if (input < 3) return "tick " + input;
        throw new RuntimeException("boom");
    })
    .retry(1)
    .elapsed()
    .subscribe(System.out::println, System.err::println);

```

Thread.sleep(2100);

执行结果:

[258,tick 0]

[250,tick 1]

[250,tick 2]

[507,tick 0]

[253,tick 1]

[249,tick 2]

java.lang.RuntimeException: boom

- Debug调试

- 1.toDebug.subscribe(System.out::println, Throwable::printStackTrace);

打印错误栈

- 2.激活debug模式(Activating Debug Mode - aka tracebacks)

```
Hooks.onOperatorDebug();
```

- 3. 生产就绪的全局调试(Production-ready Global Debugging):

### 3.1 Maven添加reactor-tools

```
io.projectreactor
```

```
reactor-tools
```

### 3.2 初始化debug代理

```
public static void main(String[] args) {  
    ReactorDebugAgent.init();  
    SpringApplication.run(Application.class, args);  
}
```

或在非main开头使用时,重新处理已存在的类:

```
ReactorDebugAgent.init();  
ReactorDebugAgent.processExistingClasses();
```

- 4. Flux.log()打印日志信息

- Reactor运算符(Reactor Operators)

- transform: Flux.transform()

transform操作符允许将操作符链的一部分封装到函数中。该函数在装配时应用于原始操作符链, 以使用封装的操作符对其进行扩充。这样做将对序列的所有订户应用相同的操作, 基本上等同于直接链接操作符。

- 示例:

```
Function<Flux, Flux> filterAndMap =  
f -> f.filter(color -> !color.equals("orange"))  
    .map(String::toUpperCase);  
Flux.fromIterable(Arrays.asList("blue", "green", "orange", "purple"))  
    .doOnNext(System.out::println)  
    .transform(filterAndMap)  
    .subscribe(d -> System.out.println("Subscriber to Transformed MapAndFilter: "+d));
```

- transformDeferred: Flux.transformDeferred()

transformDeferred操作符类似于transform, 还允许在函数中封装操作符。主要的区别在于, 该函数以每个订户为基础应用于原始序列。这意味着函数实际上可以为每个订阅生成一个不同的操作符链(通过维护一些状态);

- 示例:

```
AtomicInteger ai = new AtomicInteger();  
Function<Flux, Flux> filterAndMap = f -> {  
    if (ai.incrementAndGet() == 1) {  
        return f.filter(color -> !color.equals("orange"))  
            .map(String::toUpperCase);  
    }  
    return f.filter(color -> !color.equals("purple"))  
        .map(String::toUpperCase);  
};  
Flux composedFlux =  
Flux.fromIterable(Arrays.asList("blue", "green", "orange", "purple"))  
    .doOnNext(System.out::println)
```

```
.transformDeferred(filterAndMap);
composedFlux.subscribe(d -> System.out.println("Subscriber 1 to Composed
MapAndFilter :"+d));
composedFlux.subscribe(d -> System.out.println("Subscriber 2 to Composed
MapAndFilter: "+d));
```

- 其他操作符,如take,just,fromArray,concat等见:

<https://projectreactor.io/docs/core/release/reference/#which-operator>

- 热与冷(Hot versus Cold):

到目前为止,我们认为所有的Flux (和Mono) 都是相同的: 它们都表示一个异步的数据序列, 实际上, 有两大类发布者: hot和cold。

前面的描述(Flux和Mono)适用于冷酷的publisher家族。它们为每个订阅重新生成数据。

如果没有创建订阅, 则永远不会生成数据。

另一方面, hot publisher并不依赖于任何数量的Subscriber。他们可能会立即开始发布数据, 并在新的订阅者进来时继续这样做 (在这种情况下, 订阅者只会看到在其订阅之后发出的新元素)。对于热门出版商来说, 在你订阅之前确实会发生一些事情。

- hot 操作:

Flux.just(...)

Mono.just(...)

hot操作转换为cold操作:

Flux.defer():

Flux.defer(()->Flux.just(""))

- cold 操作:

Flux.generate()

Flux.create()

Mono.create()

cold操作转换为hot操作:

share()/replay():

Flux.just("").share().replay();

- 高级功能和概念(Advanced Features and Concepts)

- RxJava2(JDK1.6+,主要用于Android开发)

## 其他

- 异常淹没,返回值栈
  - 当Java程序的方法中有异常处理块的时候, 执行引擎可能需要处理多个返回值, 这时候, 执行引擎会将处理到的返回值, 压入到返回值栈中。
  - 不建议在finally中进行return, 因为当有异常发生且catch块中又抛出新的异常, 会淹没异常
  - try-catch中若有finally不提供catch块(catch可多个),且finally中return时,异常会被淹没
- //若为null抛出NPE  
Objects.requireNonNull(obj)
- Hash冲突解决方法
  - 开放地址法

- 线型探测法:
 

当冲突发生后, 直接去下一个位置找是否存在没用的位置, 例如2位置发生冲突, 然后去下一位置3查找, 如果3也被占用, 去找4, 直到问题解决

  - 问题: 这样就会导致落在区间内的关键字Key要进行多次探测才能找到合适的位置, 并且还会继续增大这个连续区间, 使探测时间变得更长, 这样的现象被称为“一次聚集 (primary clustering)”, 也就是说越后面的数, 如果发生hash冲突, 探测的时间越长, 因为前面的数都已经将很多可用区域占了。
- 平方探测法:
 

当冲突发生后, 当直接每次增长 $i$ 的2 倍, 即  $2 \text{ (hash值)} + (-) i^2$
- 再哈希法(双哈希法):
 

多个不同的Hash函数, 当发生冲突时, 使用第二个, 第三个, ..., 等哈希函数计算地址, 直到无冲突。虽然不易发生聚集, 但是增加了计算时间。
- 链地址法:
 

链地址法的基本思想是: 每个哈希表节点都有一个next指针, 多个哈希表节点可以用next指针构成一个单向链表, 被分配到同一个索引上的多个节点可以用这个单向链表连接起来

  - HashMap 使用该方法解决hash冲突
- 建立公共溢出区:
 

将哈希表分为基本表和溢出表两部分, 凡是和基本表发生冲突的元素, 一律填入溢出表
- clone实现方式
  - 浅拷贝: `java.lang.Object.clone()`

对象实现Cloneable接口并重写Object类中的clone()方法  
对象中的封装类型数据在对象clone以后引用的同一个对象
  - 深拷贝:
- 对象实现Serializable接口, 通过对象的序列化和反序列化实现克隆
- 对象实现Cloneable接口并重写Object类中的clone()方法, 手动拷贝, 将封装数据类型copy一次赋值
- equals实现原则  
`java.lang.Object`
  - 自反性 reflexive
    - 对于任何非null的引用值x, `x.equals(x)`必须返回true。  
`x.equals(x)` 必须为true
  - 对称性 symmetric
    - 对于任何非null的引用值x和y, 当且仅当`y.equals(x)`返回true时, `x.equals(y)`必须返回true  
`x.equals(y)`的返回要和`x.equals(y)`的返回保持一致
  - 传递性 transitive
    - 对于任何非null的引用值x,y和z, 如果`x.equals(y)`返回true, 并且`y.equals(z)`返回true, 那么`x.equals(z)`返回true  
`x.equals(y)==true` , `y.equals(z)==true` , 则 `x.equals(z) == true`

- 一致性 consistent
  - 对于任何非null的引用值x和y，只要equals的比较操作在对象中所用的信息没有被修改，多次调用x.equals(y)就会一致地返回
- 任何非null引用值x ,x.equals(null) 必须返回false

## DB


---

### NoSQL(Not Only SQL)

NoSQL，泛指非关系型的数据库

- Redis  
(Key/Value)
  - Redis: 内存Key-Value数据库  
<https://redis.io/>  
<http://www.redis.cn/>
    - Redis数据全部保存在内存中  
只有save后会持久化到磁盘
    - Redis key是二进制安全的，可以使用任何二进制序列作为key
    - 空字符串也是有效key
    - 规则
      - key 不要太长， 不仅因为消耗内存，而且在数据中查找这类键值的计算成本很高。
      - key 不要太短
      - 最好使用一种模式,如 user:1000:password按:分割的模式
  - redis能够快速执行
    - 1.绝大部分请求是纯粹的内存操作（非常快速）
    - 2.采用单线程，避免了不必要的上下文切换和竞争条件
    - 3.非阻塞IO - IO多路复用
      - IO多路复用中有三种方式：select,poll,epoll。需要注意的是，select,poll是线程不安全的，epoll是线程安全的
  - redis的内部实现:  
redis内部实现采用epoll，采用了epoll+自己实现的简单的事件框架。epoll中的读、写、关闭、连接都转化成了事件，然后利用epoll的多路复用特性;
- 基础数据类型
  - Strings 字符串
    - 内容长度不能超过512M(最大大小取决于物理内存大小)
    - SET 设置值: set mykey myvalue  
SET key value [EX seconds] [PX milliseconds] [NX|XX]
    - GET 获取值: get mykey
    - SETNX key value 当key不存在时设置值
    - SETEX key seconds value 设置key有效期，单位s



- PSETEX key milliseconds value 设置key有效期,单位ms
- SET key val NX [EX seconds] [PX milliseconds] 当key不存在时设置值:  
set mykey newval nx
- SET key val XX [EX seconds] [PX milliseconds] 当key存在时设置值:  
set mykey newval xx
- INCR原子递增
  - set counter 100  
incr counter
  - INCR 命令将字符串值解析成整型, 将其加一, 最后将结果保存为新的字符串值,
  - 类似的命令有INCRBY, DECR 和 DECRBY。
- INCRBY原子递增指定值
  - SET mykey "10"  
INCRBY mykey 5  
(integer 15)
- DECR, DECRBY 原子递减
- GETSET 设置新值并返回旧值
- MSET批量设置多个keyvalue
  - mset a 10 b 20
- MGET 批量获取多个key的值
  - mget a b  
(  
1) "10"  
2) "20"  
3) "30"  
)
- STRLEN key 获取key value的长度
- APPEND key value 在value 尾部追加value
- GETRANGE key start end 获取指定范围的字符串
- SETRANGE key offset value 重写指定位置开始的字符串为新字符串
- Hashes 散列
  - 由field和关联的value组成的map。field和value都是字符串的  
hash的域数量没有限制(除内存外)
  - HSET key field val 设置HASH 值
  - HMSET mykey field1 val1 key2 field2 val2 ... 设置多个HASH field  
value值
    - hmset user:1000 username antirez birthyear 1977 verified 1
  - HSETNX key field val 当field不存在时设置HASH field, 存在时不处理
  - HMGET mykey field1 field2 .. 获取多个HASH key的值
-  hmget user:1000 username birthyear no-such-field

- 1) "antirez"
- 2) "1977"
- > 3) (nil)
- >
- > - HGET mykey field1 获取HASH field的值
  - hget user:1000 username  
"antirez"
  - HGETALL mykey 获取所有HASH field/value对
- redis> HSET myhash field1 "Hello"  
(integer) 1  
redis> HSET myhash field2 "World"  
(integer) 1  
redis> HGETALL myhash  
1) "field1"  
2) "Hello"  
3) "field2"  
4) "World"  
- HEXISTS mykey field 获取HASH field 是否存在
  - HEXISTS myhash field1  
(integer) 0
    - HDEL mykey field1 field2 ... 删除HASH 多个field
- redis> HDEL myhash field2  
(integer) 0  
- HINCRBY key field increment 原子递增(increment)HASH field值
  - redis> HSET myhash field 5  
(integer) 1  
redis> HINCRBY myhash field 1  
(integer) 6
    - HINCRBYFLOAT key field increment 原子递增浮点数 HASH field值
    - HKEYS key 返回所有HASH field名称
  - HLEN key 获取HASH field数量
  - HSCAN key cursor [MATCH pattern] [COUNT count] 同SCAN
  - HSTRLEN key field 获取HASH field字符串长度
  - HVALS key 获取所有HASH values
- redis> HSET myhash field1 "Hello"  
(integer) 1  
redis> HSET myhash field2 "World"  
(integer) 1  
redis> HVALS myhash  
1) "Hello"  
2) "World"  
  - Lists 列表

- 按插入顺序排序的字符串元素的集合，由链表实现
- LPUSH mykey val1 val2 ... 向list头部添加多个元素，返回list 长度
- RPUSH mykey val1 val2 ... 向list尾部添加多个元素，返回list 长度
- LRANGE mykey startIdx stopIdx 从list中取出一定范围的元素
  - LRANGE 带有两个索引，一定范围的第一个和最后一个元素。这两个索引都可以为负来告知Redis从尾部开始计数，因此-1表示最后一个元素，-2表示list中的倒数第二个元素，以此类推
- LINDEX key index 获取list中某个索引的元素值
- LPOP mykey 从头部删除并返回删除的元素
- RPOP mykey 从尾部删除并返回删除的元素
- POP 空list时返回null
- LTRIM mykey startIdx stopIdx 把list从左边截取指定长度。
  - 从0位起截取到2索引位，3个元素： ltrim mykey 0 2
- list上的阻塞操作(可实现生产者，消费者队列)
  - BLPOP mylist1 mylist2 ... TIMEOUT 阻塞式从头部删除元素
    - TIMEOUT 为指定阻塞时间，若为0时，表示一直阻塞
  - BRPOP mylist1 mylist2 ... TIMEOUT 阻塞式从尾部删除元素
  - 多批量回复(multi-bulk-reply): 具体来说:  
 当没有元素可以被弹出时返回一个 nil 的多批量值，并且 timeout 过期。  
 当有元素弹出时会返回一个双元素的多批量值，其中第一个元素是弹出元素的 key，第二个元素是 value。  
 (
 

```
redis> RPUSH list1 a b c
(integer) 3
redis> BRPOP list1 list2 0
1) "list1"
2) "c"
)
```
  - RPOPLPUSH source destination
  - 原子性地返回并移除存储在 source 的列表的最后一个元素（列表尾部元素），并把该元素放入存储在 destination 的列表的第一个元素位置（列表头部）。  
 返回值：  
 bulk-string-reply: 被移除和放入的元素
    - 例如：假设 source 存储着列表 a,b,c， destination存储着列表 x,y,z。执行 RPOPLPUSH 得到的结果是 source 保存着列表 a,b，而 destination 保存着列表 c,x,y,z。
  - BRPOPLPUSH
  - LLEN mylist 获取list长度
- LREM key count element 删除列表中的指定元素

- `count > 0`: 删除等于element从头到尾移动的元素。  
`count < 0`: 删除等于element从尾到头移动的元素。  
`count = 0`: 删除所有等于的元素element。
  - `redis> RPush mylist "hello"`  
`(integer) 1`  
`redis> RPush mylist "hello"`  
`(integer) 2`  
`redis> RPush mylist "foo"`  
`(integer) 3`  
`redis> RPush mylist "hello"`  
`(integer) 4`  
`redis> LRem mylist -2 "hello"`  
`(integer) 2`  
`redis> LRange mylist 0 -1`  
`1) "hello"`  
`2) "foo"`  
`redis>`
  - 例如, `LRem list -2 "hello"`将删除"hello"存储在中的列表中最后两个出现的 list。
- `LSet key index element` 设置列表中某个位置的元素
- `redis> RPush mylist "one"`  
`(integer) 1`  
`redis> RPush mylist "two"`  
`(integer) 2`  
`redis> RPush mylist "three"`  
`(integer) 3`  
`redis> LSet mylist 0 "four"`  
`"OK"`  
`redis> LSet mylist -2 "five"`  
`"OK"`  
`redis> LRange mylist 0 -1`  
`1) "four"`  
`2) "five"`  
`3) "three"`  
`redis>`
- Sets 集合
  - 不重复且无序的字符串元素的集合。
  - `SADD key value1 ,value2 ...` 添加新元素到set中
  - `SMEMBERS key` 获取set中的所有元素
    - `redis> smembers myset1`
    - 数据量大时, 生产环境慎用, 该操作会导致服务器阻塞  
使用SSCAN替代
    - `SISMEMBER key value` 检测元素是否存在, 1:是 0:否
  - `SINTER key [key ...]` 返回从所有给定集合的交集得到的集合成员
    - `key1 = {a,b,c,d}`

- key2 = {c}
  - key3 = {a,c,e}
  - SINTER key1 key2 key3 = {c}
  - SINTERSTORE destination key [key ...] 获取给定集合的交集并保存到 destination
- SDIFF key [key ...] 返回给定集合的差集
  - key1 = {a,b,c,d}
  - key2 = {c}
  - key3 = {a,c,e}
  - SDIFF key1 key2 key3 = {b,d}
  - SDIFFSTORE destination key [key ...] 获取给定集合的差集并保存到 destination中
- SUNION key [key ...] 获取给定集合的合集
  - key1 = {a,b,c,d}
  - key2 = {c}
  - key3 = {a,c,e}
  - SUNION key1 key2 key3 = {a,b,c,d,e}
- SUNIONSTORE destination key [key ...] 获取给定集合的合集并保存到destination中
  - SCARD key 获取集合中的元素个数
  - SPOP key [count] 随机删除集合中的1个或多个元素，并返回(count参数始于3.2版本)
  - redis> spop myset 1
- SREM key member [member ...] 删除集合中的多个元素，返回从集合中删除的元素数量
  - redis> SREM myset "four"
  - (integer) 0
  - SRANDMEMBER key [count] 从集合中随机获取指定数量的元素（不删除集合中的元素）
- SMOVE source destination member 将集合中的元素移动到另一个集合中，返回元素是否移动成功，1：是，0：否
  - redis> SMOVE myset myotherset "two"
  - (integer) 0
- SSCAN
- Sorted Sets 有序集合
  - 类似Sets,但是每个字符串元素都关联到一个叫score浮动数值（floating number value）。里面的元素总是通过score进行着排序，所以不同的是，它是可以检索的一系列元素。
  - 排序规则
    - 如果A和B是两个具有不同分数的元素，那么如果A.score是> B.score，则A>B。
    - 如果A和B的分数完全相同，那么如果A字符串在字典上大于B字符串，则A>B。A和B字符串不能相等，因为排序集仅具有唯一元素。

- ZADD key [NX|XX] [CH] [INCR] score member [score member ...] 将具有指定分数的成员添加到有序集合中，可指定多个分数/成员对。返回添加到有序集合中的元素数量，不含更新分数的现有元素。  
+inf和-inf值也是有效值
  - XX: 只更新已存在元素
  - NX: 元素不存在时，创建新元素
- ZRANGE key start stop [WITHSCORES] 返回有序列表中指定范围的元素，如果得分相同，将按字典排序。带有WITHSCORES 时同时返回每个元素的分数
- redis> ZRANGE myzset 0 1 WITHSCORES
  - 1) "one"
  - 2) "1"
  - 3) "two"
  - 4) "2"
  - ZREVRANGE key start stop [WITHSCORES] 返回有序集key中，指定区间内的成员。其中成员的位置按score值递减(从大到小)来排列。具有相同score值的成员按字典序的反序排列。
- ZCARD key 返回有序列表的基数（元素数）
  - redis> ZCARD myzset  
(integer) 2
- ZCOUNT key min max 返回有序集合中的元素数量，元素得分介于min和max间，默认包含min max的分数。  
(min (max 时不包含值为min max分数的元素)
- redis> ZADD myzset 1 "one"  
(integer) 1  
redis> ZADD myzset 2 "two"  
(integer) 1  
redis> ZADD myzset 3 "three"  
(integer) 1  
redis> ZCOUNT myzset -inf +inf  
(integer) 3  
redis> ZCOUNT myzset (1 3  
(integer) 2  
redis>
  - redis> zcount myzset (1 (3  
(integer) 1
  - ZINCRBY key increment member 递增元素在有序集合中的分数
- ZSCORE key member 返回有序集合中元素的分数
- redis> ZADD myzset 1 "one"  
(integer) 1  
redis> ZSCORE myzset "one"  
"1"  
redis>
  - ZREM key member [member ...] 删除有序集合中多个元素
- 等: <https://redis.io/commands/zunionstore>

- Bitmaps (Bit arrays)
  - simply bitmaps: 通过特殊的命令，你可以将 String 值当作一系列 bits 处理：可以设置和清除单独的 bits，数出所有设为 1 的 bits 的数量，找到最前的被设为 1 或 0 的 bit，等等
- hyperloglogs
  - 被用于估计一个 set 中元素数量的概率性的数据结构
  - PFADD key element [element ...]
- 命令
  - KEYS pattern 列出所有匹配的key
    - 生产环境慎用，该操作会导致服务器阻塞  
使用SCAN替代
  - EXISTS mykey 查询key是否存在，返回1/0
  - DEL mykey 删除key，返回1/0
- TYPE mykey 返回key对应的存储类型
  - string
  - hash
  - list
  - set
  - zset
  - EXPIRE mykey secondVal 设置key有效期，精度可以使用毫秒或秒，默认：s
    - 设置key5s 超时 expire key 5
  - TTL mykey 获取key剩余有效时间
- key的自动创建和删除
  - 推入元素之前创建空的 list，或者在 list 没有元素时删除它。在 list 为空时删除 key，并在用户试图添加元素（比如通过 LPUSH）而键不存在时创建空 list，是 Redis 的职责。
    - 三条规则来概括
      - 当我们向一个聚合数据类型中添加元素时，如果目标键不存在，就在添加元素前创建空的聚合数据类型。
      - 当我们从聚合数据类型中移除元素时，如果值仍然是空的，键自动被销毁。
      - 对一个空的 key 调用一个只读的命令，比如 LLEN（返回 list 的长度），或者一个删除元素的命令，将总是产生同样的结果。该结果和对一个空的聚合类型做同个操作的结果是一样的。
  - SCAN
    - SCAN 命令用于迭代当前数据库中的key集合。  
SSCAN 命令用于迭代SET集合中的元素。  
HSCAN 命令用于迭代Hash类型中的键值对。  
ZSCAN 命令用于迭代SortSet集合中的元素和元素对应的分值
    - SCAN 命令用于迭代当前数据库中的key集合。
      - SCAN cursor [MATCH pattern] [COUNT count]

- SCAN命令是一个基于游标的迭代器。这意味着命令每次被调用都需要使用上一次这个调用返回的游标作为该次调用的游标参数，以此来延续之前的迭代过程  
当SCAN命令的游标参数被设置为 0 时， 服务器将开始一次新的迭代，而当服务器向用户返回值为 0 的游标时， 表示迭代已结束。
- SCAN命令的返回值 是一个包含两个元素的数组， 第一个数组元素是用于进行下一次迭代的新游标， 而第二个数组元素则是一个数组， 这个数组中包含了所有被迭代的元素。  
在第二次调用 SCAN 命令时， 命令返回了游标 0， 这表示迭代已经结束， 整个数据集已经被完整遍历过了。  
full iteration： 以 0 作为游标开始一次新的迭代， 一直调用 SCAN 命令， 直到命令返回游标 0， 我们称这个过程为一次完整遍历。
- Scan命令的保证（SCAN命令以及其他增量式迭代命令）
  - 从完整遍历开始直到完整遍历结束期间， 一直存在于数据集内的所有元素都会被完整遍历返回； 这意味着， 如果有一个元素， 它从遍历开始直到遍历结束期间都存在于被遍历的数据集中， 那么 SCAN 命令总会在某次迭代中将这个元素返回给用户。
  - 同样， 如果一个元素在开始遍历之前被移出集合， 并且在遍历开始直到遍历结束期间都没有再加入， 那么在遍历返回的元素集中就不会出现该元素。
  - SCAN命令每次执行返回的元素数量
    - SCAN增量式迭代命令并不保证每次执行都返回某个给定数量的元素,甚至可能会返回零个元素， 但只要命令返回的游标不是 0， 应用程序就不应该将迭代视作结束。
    - 默认COUNT=10， 即最大返回10条记录
  - cursor 游标
    - 在开始一个新的迭代时， 游标必须为 0。  
增量式迭代命令在执行之后返回的， 用于延续迭代过程的游标。
- SCAN, SSCAN, HSCAN 和 ZSCAN 命令都返回一个包含两个元素的 multi-bulk 回复： 回复的第一个元素是字符串表示的无符号 64 位整数（游标）， 回复的第二个元素是另一个 multi-bulk 回复， 包含了本次被迭代的元素。  
SCAN 命令返回的每个元素都是一个key。  
SSCAN 命令返回的每个元素都是一个集合成员。  
HSCAN 命令返回的每个元素都是一个键值对， 一个键值对由一个键和一个值组成。  
ZSCAN命令返回的每个元素都是一个有序集合元素， 一个有序集合元素由一个成员（member）和一个分值（score）组成。
  - SSCAN 命令用于迭代SET集合中的元素。
  - HSCAN 命令用于迭代Hash类型中的键值对。
    - ZSCAN 命令用于迭代SortSet集合中的元素和元素对应的分值
- 其他功能
  - Geospatial 地理空间半径查询



- GEOADD key longitude latitude member [longitude latitude member ...]
  - 存储在ZSET中
  - 将指定的地理空间位置（纬度、经度、名称）添加到指定的key中。这些数据将会存储到sorted set这样的目的是为了更方便使用GEORADIUS或者GEORADIUSBYMEMBER命令对数据进行半径查询等操作。
  - 返回添加到ZSET中的元素数目，不包含更新score的元素
    - redis> GEOADD Sicily 13.361389 38.115556 "Palermo" 15.087269 37.502669 "Catania" (integer) 2
    - redis> GEODIST Sicily Palermo Catania "166274.15156960039"
    - redis> GEORADIUS Sicily 15 37 100 km 1) "Catania"
    - redis> GEORADIUS Sicily 15 37 200 km 1) "Palermo" 2) "Catania"
    - redis>
- GEODIST key member1 member2 [unit]
  - 返回两个给定位置之间的距离。
  - 如果两个位置之间的其中一个不存在，那么命令返回空值。
  - 指定单位的参数 unit 必须是以下单位的其中一个：
    - m 表示单位为米。
    - km 表示单位为千米。
    - mi 表示单位为英里。
    - ft 表示单位为英尺。
  - 如果用户没有显式地指定单位参数，那么 GEODIST 默认使用米作为单位。
  - redis> GEOADD Sicily 13.361389 38.115556 "Palermo" 15.087269 37.502669 "Catania" (integer) 2
  - redis> GEODIST Sicily Palermo Catania "166274.15156960039"
  - redis> GEODIST Sicily Palermo Catania km "166.27415156960038"
  - redis> GEODIST Sicily Palermo Catania mi "103.31822459492736"
  - redis> GEODIST Sicily Foo Bar (nil)
  - redis>
  - 计算出的距离会以双精度浮点数的形式被返回。如果给定的位置元素不存在，那么命令返回空值。
- GEOHASH key member [member ...]
  - 返回一个或多个位置元素的 Geohash 表示。

通常使用表示位置的元素使用不同的技术，使用Geohash位置52点整数编码。由于编码和解码过程中所使用的初始最小和最大坐标不同，编码的编码也不同于标准。此命令返回一个标准的Geohash，在维基百科和geohash.org网站都有相关描述

```
- redis> GEOADD Sicily 13.361389 38.115556 "Palermo" 15.087269
37.502669 "Catania"
(integer) 2
redis> GEOHASH Sicily Palermo Catania
1) "sqc8b49rny0"
2) "sqdtr74hyu0"
redis>
```

- GEOPOS key member [member ...]
  - 从key里返回所有给定位置元素的位置（经度和纬度）。
    - redis> GEOADD Sicily 13.361389 38.115556 "Palermo" 15.087269 37.502669 "Catania"  
(integer) 2  
redis> GEOPOS Sicily Palermo Catania NonExisting  
1) 1) "13.361389338970184"  
2) "38.115556395496299"  
2) 1) "15.087267458438873"  
2) "37.50266842333162"  
3) (nil)  
redis>
  - array-reply, 具体的:  
GEOPOS 命令返回一个数组，数组中的每个项都由两个元素组成：第一个元素为给定位置元素的经度，而第二个元素则为给定位置元素的纬度。  
当给定的位置元素不存在时，对应的数组项为空值。
- GEORADIUS key longitude latitude radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count] [ASC|DESC]
  - 以给定的经纬度为中心，返回键包含的位置元素当中，与中心的距离不超过给定最大距离的所有位置元素。  
范围可以使用以下其中一个单位：  
m 表示单位为米。  
km 表示单位为千米。  
mi 表示单位为英里。  
ft 表示单位为英尺。  
在给定以下可选项时，命令会返回额外的信息：  
WITHDIST: 在返回位置元素的同时，将位置元素与中心之间的距离也一并返回。距离的单位和用户给定的范围单位保持一致。  
WITHCOORD: 将位置元素的经度和纬度也一并返回。  
WITHHASH: 以 52 位有符号整数的形式，返回位置元素经过原始 geohash 编码的有序集合分值。这个选项主要用于底层应用或者调试，实际中的作用并不大。  
命令默认返回未排序的位置元素。通过以下两个参数，用户可以指

定被返回位置元素的排序方式：

ASC: 根据中心的位置，按照从近到远的方式返回位置元素。

DESC: 根据中心的位置，按照从远到近的方式返回位置元素。

```
■ redis> GEOADD Sicily 13.361389 38.115556 "Palermo"
15.087269 37.502669 "Catania"
(integer) 2
redis> GEORADIUS Sicily 15 37 200 km WITHDIST
1) 1) "Palermo"
2) "190.4424"
2) 1) "Catania"
2) "56.4413"
redis> GEORADIUS Sicily 15 37 200 km WITHCOORD
1) 1) "Palermo"
2) 1) "13.361389338970184"
2) "38.115556395496299"
2) 1) "Catania"
2) 1) "15.087267458438873"
2) "37.50266842333162"
redis> GEORADIUS Sicily 15 37 200 km WITHDIST
WITHCOORD
1) 1) "Palermo"
2) "190.4424"
3) 1) "13.361389338970184"
2) "38.115556395496299"
2) 1) "Catania"
2) "56.4413"
3) 1) "15.087267458438873"
2) "37.50266842333162"
redis>
```

■ 返回：

bulk-string-reply, 具体的：

在没有给定任何 WITH 选项的情况下，命令只会返回一个像 ["New York","Milan","Paris"] 这样的线性（linear）列表。

在指定了 WITHCOORD、WITHDIST、WITHHASH 等选项的情况下，命令返回一个二层嵌套数组，内层的每个子数组就表示一个元素。

在返回嵌套数组时，子数组的第一个元素总是位置元素的名字。至于额外的信息，则会作为子数组的后续元素，按照以下顺序被返回：

以浮点数格式返回的中心与位置元素之间的距离，单位与用户指定范围时的单位一致。

geohash 整数。

由两个元素组成的坐标，分别为经度和纬度。

举个例子，GEORADIUS Sicily 15 37 200 km WITHCOORD WITHDIST 这样的命令返回的每个子数组都是类似以下格式的：  
["Palermo","190.4424",

["13.361389338970184","38.115556395496299"]]

- GEORADIUSBYMEMBER key member radius m|km|ft|mi  
[WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count]
- 这个命令和 GEORADIUS 命令一样，都可以找出位于指定范围内的元素，但是 GEORADIUSBYMEMBER 的中心点是由给定的位置元素决定的，而不是像 GEORADIUS 那样，使用输入的经度和纬度来决定中心点  
指定成员的位置被用作查询的中心。  
关于 GEORADIUSBYMEMBER 命令的更多信息，请参考 GEORADIUS 命令的文档。

```
redis> GEOADD Sicily 13.583333 37.316667 "Agrigento"
(integer) 1
redis> GEOADD Sicily 13.361389 38.115556 "Palermo"
15.087269 37.502669 "Catania"
(integer) 2
redis> GEORADIUSBYMEMBER Sicily Agrigento 100 km
1) "Agrigento"
2) "Palermo"
redis>
```

- 可用业务场景: 附近的人，距离位置

- Lua Scripting LUA脚本

- Transactions 事务

- MULTI 、 EXEC 、 DISCARD 和 WATCH 是 Redis 事务相关的命令
- 事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。
- 事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。
- EXEC 命令负责触发并执行事务中的所有命令
- 如果客户端在使用 MULTI 开启了一个事务之后，却因为断线而没有成功执行 EXEC ，那么事务中的所有命令都不会被执行。
- 另一方面，如果客户端成功在开启事务之后执行 EXEC ，那么事务中的所有命令都会被执行。

- 事务中的错误

- 事务在执行 EXEC 之前，入队的命令可能会出错。比如说，命令可能会产生语法错误（参数数量错误，参数名错误，等等），或者其他更严重的错误，比如内存不足（如果服务器使用 maxmemory 设置了最大内存限制的话）。
- 命令可能在 EXEC 调用之后失败。举个例子，事务中的命令可能处理了错误类型的键，比如将列表命令用在了字符串键上面，诸如此类。
- 对于发生在 EXEC 执行之前的错误，客户端以前的做法是检查命令入队所得的返回值：如果命令入队时返回 QUEUED ，那么入队成功；否则，就是入队失败。如果有命令在入队时失败，那么大部分客户端都会停止并取消这个事务。  
不过，从 Redis 2.6.5 开始，服务器会对命令入队失败的情况

进行记录，并在客户端调用 EXEC 命令时，拒绝执行并自动放弃这个事务。

- MULTI 命令开启一个事务，总是返回OK
  - MULTI 执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当 EXEC命令被调用时，所有队列中的命令才会被执行。

- - MULTI
  - OK

INCR foo

QUEUED

INCR bar

QUEUED

EXEC

1) (integer) 1

2) (integer) 1

- DISCARD 客户端可以清空事务队列，并放弃执行事务

-> SET foo 1

OK

MULTI

OK

INCR foo

QUEUED

DISCARD

OK

GET foo

"1"

- EXEC 执行命令队列中的所有命令，

返回一个数组，数组中的每个元素都是执行事务中的命令所产生的回复。其中，回复元素的先后顺序和命令发送的先后顺序一致。

- WATCH 命令可以为 Redis 事务提供 check-and-set (CAS) 行为。

- 被 WATCH 的键会被监视，并会发觉这些键是否被改动过了。如果有至少一个被监视的键在 EXEC 执行之前被修改了，那么整个事务都会被取消，EXEC 返回nil-reply来表示事务已经失败。

- Persistence 磁盘持久化

- RDB持久化方式能够在指定的时间间隔能对数据进行快照存储。

- AOF持久化方式记录每次对服务器写的操作,当服务器重启的时候会重新执行这些命令来恢复原始的数据,AOF命令以redis协议追加保存每次写的操作到文件末尾.Redis还能对AOF文件进行后台重写,使得AOF文件的体积不至于过大。

- 如果你只希望你的数据在服务器运行的时候存在,你也可以不使用任何持久化方式。

- 你也可以同时开启两种持久化方式,在这种情况下,当redis重启的时候会优先载入AOF文件来恢复原始的数据,因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整。

- RDB和AOF持久化

- RDB(Redis Database)

对数据进行快照存储

## 适用于数据备份

### - RDB的优点

- RDB是一个非常紧凑的文件,它保存了某个时间点的数据集,非常适用于数据集的备份,比如你可以在每个小时保存一下过去24小时内的数据,同时每天保存过去30天的数据,这样即使出了问题你也可以根据需求恢复到不同版本的数据集。

- RDB是一个紧凑的单一文件,很方便传送到另一个远端数据中心或者亚马逊的S3（可能加密），非常适用于灾难恢复。

- RDB在保存RDB文件时父进程唯一需要做的就是fork出一个子进程,接下来的工作全部由子进程来做，父进程不需要再做其他IO操作，所以RDB持久化方式可以最大化redis的性能。

- 与AOF相比,在恢复大的数据集的时候，RDB方式会更快一些。

### - RDB的缺点

- 如果你希望在redis意外停止工作（例如电源中断）的情况下丢失的数据最少的话，那么RDB不适合你.虽然你可以配置不同的save时间点(例如每隔5分钟并且对数据集有100个写的操作),是Redis要完整的保存整个数据集是一个比较繁重的工作,你通常会每隔5分钟或者更久做一次完整的保存,万一在Redis意外宕机,你可能会丢失几分钟的数据。

- RDB 需要经常fork子进程来保存数据集到硬盘上,当数据集比较大的时候,fork的过程是非常耗时的,可能会导致Redis在一些毫秒级内不能响应客户端的请求.如果数据集巨大并且CPU性能不是很好的情况下,这种情况会持续1秒,AOF也需要fork,但是你可以调节重写日志文件的频率来提高数据集的耐久度。

### - AOF(Append-only file, AOF)

配置：appendonly yes

通过追加命令到文件末尾，恢复数据时重新执行AOF文件命令来达到数据恢复

### - AOF优点

- 使用AOF 会让你的Redis更加耐久: 你可以使用不同的fsync策略: 无fsync, 每秒fsync,每次写的时候fsync.使用默认的每秒fsync策略,Redis的性能依然很好(fsync是由后台线程进行处理的,主线程会尽力处理客户端请求),一旦出现故障，你最多丢失1秒的数据。

- AOF文件是一个只进行追加的日志文件,所以不需要写入seek,即使由于某些原因(磁盘空间已满，写的过程中宕机等等)未执行完整的写入命令,你也可使用redis-check-aof工具修复这些问题。

- Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写：重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。

- AOF 文件有序地保存了对数据库执行的所有写入操作，这些写入操作以 Redis 协议的格式保存，因此 AOF 文件的内容非常容易被别人读懂，对文件进行分析（parse）也很轻松。导出（export）AOF 文件也非常简单：举个例子，如果你不小心执行了 FLUSHALL 命令，但只要 AOF 文件未被重写，那么只要停止服务器，移除 AOF 文件末尾的 FLUSHALL 命令，并重启 Redis，就可以将数据集恢复到 FLUSHALL 执行之前的状态。

### - AOF缺点

- 对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。

- 根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB 。在一般情况下，每秒 fsync 的性能依然非常高，而关闭 fsync 可以让 AOF 的速度和 RDB 一样快，即使在高负荷之下也是如此。不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）。

- fsync 3 种方式

- 每次有新命令追加到 AOF 文件时就执行一次 fsync：非常慢，也非常安全

- 每秒 fsync 一次：足够快（和使用 RDB 持久化差不多），并且在故障时只会丢失 1 秒钟的数据。

- 从不 fsync：将数据交给操作系统来处理。更快，也更不安全的选择。

- 推荐（并且也是默认）的措施为每秒 fsync 一次，这种 fsync 策略可以兼顾速度和安全性。

- LRU 缓存回收算法

- Redis 支持 LRU，LFU (4.0+)

- Maxmemory 配置指令

- maxmemory 配置指令用于配置 Redis 存储数据时指定限制的内存大小。

- 例如为了配置内存限制为 100mb，以下的指令可以放在 redis.conf 文件中：

maxmemory 100mb

- 设置 maxmemory 为 0 代表没有内存限制。对于 64 位的系统这是个默认值，对于 32 位的系统默认内存限制为 3GB。

- maxmemory-policy 配置（8 种，LRU：5 种，LFU：2 种，返回错误：1 种）

- LRU 是 Least Recently Used 的缩写，即最近最少使用算法

LFU：Least frequently used 最近最少使用频率算法

- 如果没有键满足回收的前提条件的话，策略 volatile-lru, volatile-random 以及 volatile-ttl 就和 noeviction 差不多了。

- noeviction: 返回错误当内存限制达到并且客户端尝试执行会让更多内存被使用的命令（大部分的写入指令，但 DEL 和几个例外）

- allkeys-lru: 尝试回收最久没使用的键（LRU），使得新添加的数据有空间存放。

- volatile-lru: 尝试回收最久没使用的键（LRU），首先从设置了过期时间的键集合中驱逐最久没有使用的键，使得新添加的数据有空间存放。

- allkeys-random: 回收随机的键使得新添加的数据有空间存放。

- volatile-random: 回收随机的键使得新添加的数据有空间存放，但仅限于在过期集合的键。

- volatile-ttl: 回收在过期集合的键，并且优先回收存活时间（TTL）较短的键，使得新添加的数据有空间存放。

- allkeys-lfu: 从所有键中驱逐使用频率最少的键

- volatile-lfu: 从所有配置了过期时间的键中驱逐使用频率最少的键

- Replication 主从复制

<http://www.redis.cn/topics/replication>

- 复制系统运行需要 3 个主要机制

- 当一个 master 实例和一个 slave 实例连接正常时，master 会发送一连串的命令流来保持对 slave 的更新，以便于将自身数据集的改变复制给 slave，：包括客户端的写入、key 的过期或被逐出等等。（4.0+ sub-slave 也从 master 同步复制流）

- 当 master 和 slave 之间的连接断开之后，因为网络问题、或者是主从意识到连接超时，slave 重新连接上 master 并会尝试进行部分重同步：这意味着它会尝试只获取在断开连接期间内丢失的命令流。

- 当无法进行部分重同步时，slave 会请求进行全量重同步。这会涉及到一个更复杂的过程，例如 master 需要创建所有数据的快照，将之发送给 slave，之后在数据集更改时持续发送命令流到 slave。

- Redis使用默认的异步复制，其特点是低延迟和高性能，是绝大多数 Redis 用例的自然复制模式。但是，从 Redis 服务器会异步地确认其从主 Redis 服务器周期接收到的数据量。

- Redis 复制的非常重要的事实

- Redis为异步复制,slave 和 master 之间异步地确认处理的数据量

- 一个Master可以有多个slave

- slave 可以接受其他 slave 的连接。除了多个 slave 可以连接到同一个 master 之外，slave 之间也可以像层叠状的结构（cascading-like structure）连接到其他 slave。自 Redis 4.0 起，所有的 sub-slave 将会从 master 收到完全一样的复制流。

- Redis 复制在 master 侧是非阻塞的。这意味着 master 在一个或多个 slave 进行初次同步或者是部分重同步时，可以继续处理查询请求。

- 复制在 slave 侧大部分也是非阻塞的。当 slave 进行初次同步时，它可以使用旧数据集处理查询请求，假设你在 redis.conf 中配置了让 Redis 这样做的话。否则，你可以配置如果复制流断开，Redis slave 会返回一个 error 给客户端。但是，在初次同步之后，旧数据集必须被删除，同时加载新的数据集。slave 在这个短暂的时间窗口内（如果数据集很大，会持续较长时间），会阻塞到来的连接请求。自 Redis 4.0 开始，可以配置 Redis 使删除旧数据集的操作在另一个不同的线程中进行，但是，加载新数据集的操作依然需要在主线程中进行并且会阻塞 slave。

- 复制既可以被用在可伸缩性，以便只读查询可以有多个 slave 进行（例如 O(N) 复杂度的慢操作可以被下放到 slave），或者仅用于数据安全。

- 可以使用复制来避免 master 将全部数据集写入磁盘造成的开销：一种典型的技术是配置你的 master Redis.conf 以避免对磁盘进行持久化，然后连接一个 slave，其配置为不定期保存或是启用 AOF。但是，这个设置必须小心处理，因为重新启动的 master 程序将从一个空数据集开始：如果一个 slave 试图与它同步，那么这个 slave 也会被清空。

- 当 master 关闭持久化时，复制的安全性

- 在使用 Redis 复制功能时的设置中，强烈建议在 master 和在 slave 中启用持久化。当不可能启用时，例如由于非常慢的磁盘性能而导致的延迟问题，应该配置实例来避免重置后自动重启。

- 当 Redis Sentinel 被用于高可用并且 master 关闭持久化，这时如果允许自动重启进程也是很危险的。例如，master 可以重启的足够快以致于 Sentinel 没有探测到故障，因此上述的故障模式也会发生。

任何时候数据安全性都是很重要的，所以如果 master 使用复制功能的同时未配置持久化，那么自动重启进程这项应该被禁用。

- Redis复制工作原理

- 每一个 Redis master 都有一个 replication ID：这是一个较大的伪随机字符串，标记了一个给定的数据集。每个 master 也持有一个偏移量，master 将自己产生的复制流发送给 slave 时，发送多少字节的数据，自身的偏移量就会增加多少，目的是当有新的操作修改自己的数据集时，它可以以此更新 slave 的状态。复制偏移量即使在没有一个 slave 连接到 master 时，也会自增，所以基本上每一对给定的 Replication ID, offset

都会标识一个 master 数据集的确切版本。

- 当 slave 连接到 master 时，它们使用 PSYNC 命令来发送它们记录的旧的



master replication ID 和它们至今为止处理的偏移量。通过这种方式，master 能够仅发送 slave 所需的增量部分。但是如果 master 的缓冲区中没有足够的命令积压缓冲记录，或者如果 slave 引用了不再知道的历史记录（replication ID），则会转而进行一个全量重同步：在这种情况下，slave 会得到一个完整的数据集副本，从头开始(即,发送 SYNC到master,master执行bgsave生成RDB文件,然后同步给slave,再将缓冲区的命令流同步给slave)。

- 全量同步的工作细节

- 1.master 开启一个后台保存进程，以便于生产一个 RDB 文件。同时它开始缓存所有从客户端接收到的新的写入命令(新进程来保存RDB数据)

- 2.当后台保存完成时，master 将数据集文件传输给 slave，slave 将之保存在磁盘上，然后加载文件到内存。再然后 master 会发送所有缓冲的命令发给 slave。这个过程以指令流的形式完成并且和 Redis 协议本身的格式相同。

- 如果 master 收到了多个 slave 要求同步的请求，它会执行一个单独的后台保存，以便于为多个 slave 服务；

事实上 SYNC 是一个旧协议，在新的 Redis 实例中已经不再被使用，但是其仍然向后兼容：但它不允许部分重同步，所以现在 PSYNC 被用来替代 SYNC。

- 无需磁盘参与的复制：

正常情况下，一个全量重同步要求在磁盘上创建一个 RDB 文件，然后将它从磁盘加载进内存，然后 slave 以此进行数据同步。

如果磁盘性能很低的话，这对 master 是一个压力很大的操作。Redis 2.8.18 是第一个支持无磁盘复制的版本。在此设置中，子进程直接发送 RDB 文件给 slave，无需使用磁盘作为中间储存介质。

- 配置基本的 Redis 复制功能是很简单的：只需要将以下内容加进 slave 的配置文件：slaveof 192.168.1.1 6379

- 只读性质的 slave:

自从 Redis 2.6 之后，slave 支持只读模式且默认开启。redis.conf 文件中的 slave-read-only 变量控制这个行为，且可以在运行时使用 CONFIG SET 来随时开启或者关闭。

只读模式下的 slave 将会拒绝所有写入命令，因此实践中不可能由于某种出错而将数据写入 slave。但这并不意味着该特性旨在将一个 slave 实例暴露到 Internet，或者更广泛地说，将之暴露在存在不可信客户端的网络，因为像 DEBUG 或者 CONFIG 这样的管理员命令仍在启用。但是，在 redis.conf 文件中使用 rename-command 指令可以禁用上述管理员命令以提高只读实例的安全性。

- Redis 复制如何处理 key 的过期:

Redis 的过期机制可以限制 key 的生存时间。此功能取决于 Redis 实例计算时间的能力，但是，即使使用 Lua 脚本更改了这些 key，Redis slaves 也能正确地复制具有过期时间的 key。

Redis 使用三种主要的技术使过期的 key 的复制能够正确工作：

- slave 不会让 key 过期，而是等待 master 让 key 过期。当一个 master 让一个 key 到期（或由于 LRU 算法将之驱逐）时，它会合成一个 DEL 命令并传输到所有的 slave。

- 但是，由于这是 master 驱动的关键过期行为，master 无法及时提供 DEL 命令，所以有时候 slave 的内存中仍然存在逻辑上已过期的 key。为了处理这个问题，slave 使用它的逻辑时钟以报告只有在不违反数据集的一致性的读取操作（从主机的新命令到达）中才存在 key。用这种方法，slave 避免报告逻辑过期的 key 仍然存在。在实际应用中，使用 slave 程序进行缩放的 HTML 碎片缓存，将避免返回已经比期

望的时间更早的数据项。

- 在Lua脚本执行期间，不执行任何 key 过期操作。当一个Lua脚本运行时，从概念上讲，master 中的时间是被冻结的，这样脚本运行的时候，一个给定的键要么存在要么不存在。这可以防止 key 在脚本中间过期，保证将相同的脚本发送到 slave，从而在二者的数据集中产生相同的效果。

- 高可用性: Sentinel(哨兵)，集群(Cluster)

- redis集群模式为：

- 主从(哨兵)

- 2.多主从(哨兵)+数据分片

- 主从+从

- redis-cli

- 连接redis服务：redis-cli.exe -h 127.0.0.1 -p 6379
    - select [index]：切换到指定的数据库，数据库索引号 index 用数字值指定，以 0 作为起始索引值，最新版本支持0-15个数据库。集群环境只有1个库
    - auth [password]：简单密码认证
    - time：返回当前服务器时间，unix时间戳
    - client list：返回所有连接到服务器的客户端和统计数据
    - client kill [ip:port]：关闭地址为ip:port的客户端
    - save：将数据同步保存到磁盘
    - bgsave：将数据异步保存到磁盘
    - lastsave：返回上次成功将数据保存到磁盘的Unix时间戳
    - shutdown [nosave|save]：保存数据并关闭服务
    - slaveof [master\_ip:port]：设置主从

- 集群

- 高可用性: Sentinel(哨兵，主从)，集群(Cluster)

- 集群(Cluster)下的数据分片

- 最大分片数为 $16384(2^{14})$ 个

- 即最大节点主机数为 $16384(2^{14})$ 个

- <https://www.cnblogs.com/yueerya/articles/11507037.html>

- Redis集群有 $16384(2^{14})$ 个Hash槽(即分片)，  
每个Key通过CRC16校验后对 $16384(2^{14})$ 取模来决定数据存放的槽
      - 比如当前有三个节点 那么：  
节点 A 包含 0 到 5500号哈希槽。  
节点 B 包含5501 到 11000 号哈希槽。  
节点 C 包含11001 到 16384号哈希槽。
      - $HASH\_SLOT = CRC16(客户端key) \bmod 16384$
      - CRC16算法产生的hash值有16bit,可以产生的值在0~65535之间
      - 在redis节点发送心跳包时需要把所有的槽放到这个心跳包里，以便让节点知道当前集群信息， $16384=16k$ ，在发送心跳包时使用char进行bitmap压缩后是2k ( $2 * 8(8\text{ bit}) * 1024(1k) = 2K$ )，也就是说使用2k的空间创建了16k的槽数。  
 $65535=65k$ ，压缩后就是8k ( $8 * 8(8\text{ bit}) * 1024(1k) = 8K$ )，也就是说需要需要8k的心跳包。

- Redis Cluster原理

node1和node2首先进行握手meet，知道彼此的存在

握手成功后，两个节点会定期发送ping/pong消息，交换数据信息(消息头，消息体)

消息头里面有个字段：unsigned char myslots[CLUSTER\_SLOTS/8]，每一位代表一个槽，如果该位是1，代表该槽属于这个节点

消息体中会携带一定数量的其他节点的信息，大约占集群节点总数的十分之一，至少是3个节点的信息。节点数量越多，消息体内容越大。

每秒都在发送ping消息。每秒随机选取5个节点，找出最久没有通信的节点发送ping消息。

每100毫秒都会扫描本地节点列表，如果发现节点最近一次接受pong消息的时间大于cluster-node-timeout/2,则立即发送ping消息

redis集群的主节点数量基本不可能超过1000个，超过的话可能会导致网络拥堵。

- 为什么集群节点使用16384个槽(而不是CRC16的 $2^{16}-1=65535$ )

(<https://github.com/redis/redis/issues/2576>)

- 正常的心跳包中包含节点全部的配置信息，传输时可以使用bitmap2K传输16K的槽数据,但需要使用8K传输65K的槽数据
  - 由于设计折中，Redis集群不太可能扩展到1000个以上的节点
  - 槽位越小,节点少的情况下,传输压缩率更高
- Redis节点配置信息中，Hash槽是通过bitmap保存的，在传输过程中会对bitmap进行压缩,如果bitmap的填充率slots/N(N表示节点数)很高的话,bitmap压缩率就很低

- 主从模式(可读写分离,或叫哨兵模式,用于主从热备)

- 主从模式为从节点复制主节点数据,主节点宕机以后,从节点自动转为主节点
- 哨兵模式基于主从复制,哨兵通常实现了自动的故障恢复,主节点故障后可从多个从节点中选取主节点,并将其他从节点从新主节点中同步数据
- 哨兵模式即1主多从,在非集群环境下,可对从库继续做从库高可用
- 哨兵模式下可配置主库只写,读都走从库
- 模式redis-proxy、master、replica、HA等几个角色。在读写分离实例中，新增read-only replica角色来承担读流量，replica作为热备不提供服务，架构上保持对现有集群规格的兼容性。redis-proxy按权重将读写请求转发到master或者某个read-only replica上；HA负责监控DB节点的健康状态，异常时发起主从切换或重搭read-only replica，并更新路由。

- 星型复制:

星型复制就是将所有的read-only replica直接和master保持同步，每个read-only replica之间相互独立，任何一个节点异常不影响到其他节点，同时因为复制链比较短，read-only replica上的复制延迟比较小。

Redis是单进程单线程模型，主从之间的数据复制也在主线程中处理，read-only replica数量越多，数据同步对master的CPU消耗就越严重，集群的写入性能会随着read-only replica的增加而降低。此外，星型架构会让master的出口带宽随着read-only replica的增加而成倍增长。

Master上较高的CPU和网络负载会抵消掉星型复制延迟较低的优势，因

此，星型复制架构会带来比较严重的扩展问题，整个集群的性能会受限于master。

- 链式复制:

链式复制将所有的read-only replica组织成一个复制链，如下图所示，master只需要将数据同步给replica和复制链上的第一个read-only replica。

链式复制解决了星型复制的扩展问题，理论上可以无限增加read-only replica的数量，随着节点的增加整个集群的性能也可以基本上呈线性增长。

链式复制的架构下，复制链越长，复制链末端的read-only replica和master之间的同步延迟就越大，考虑到读写分离主要使用在对一致性要求不高的场景下，这个缺点一般可以接受。但是如果复制链中的某个节点异常，会导致下游的所有节点数据都会大幅滞后。更加严重的是这可能带来全量同步，并且全量同步将一直传递到复制链的末端，这会对服务带来一定的影响。为了解决这个问题，读写分离的Redis都使用阿里云优化后的binlog复制版本，最大程度的降低全量同步的概率。

- 配置

- Redis 的配置文件位于 Redis 安装目录下，文件名为 redis.conf(Windows 名为 redis.windows.conf)。
- 通过 CONFIG 命令查看或设置配置项

- 调优

- 数据持久化配置
- 高可用配置(集群)

- 生产问题(缓存穿透)

- 缓存雪崩

- 原因:

- 多个热点Key在高并发下同时失效，导致并发请求落到后端数据库，导致后端数据库压力甚至宕机
- Redis 数据库崩溃，导致缓存服务失效
  - 解决方案:
- 不同热点Key 设置不同的超时时间，避免同时失效
- 设置热点Key数据永不过期
- Redis服务高可用: 设置主从及分布式部署
- 服务限流及降级，控制服务请求流量，避免压垮后端数据库
  - 缓存穿透
  - 原因:
- 请求先从Redis中未查询到缓存，到数据库中也未查询到缓存，高并发下，导致大量无效请求
  - 解决方案:
- 在代码层面，控制无效Redis Key
- 查询数据库后，将空结果缓存到Redis中，防止同一Key 频繁请求数据库
  - 缓存击穿
  - 原因:

- 高并发请求下, 某个Redis Key过期瞬间, 大量请求落到后端数据库
  - 解决方案:
- 设置热点Redis Key永不过期
- 缓存失效后, 使用分布式锁来控制查询, 只有1个请求来查询数据和更新缓存, 其他请求等待或直接返回无数据
  - 其他问题
    - 1.内存设置过大,RDB,AOF fork的子进程持久化数据时间长,导致主服务不可用,可适当减小内存或关闭主节点持久化及自动重启,在从节点做持久化
    - 2.一主多从导致复制风暴,某一时间点,所有从节点对主节点进行同步(主从数据同步都是异步)时,网络占用带宽大,导致主节点不可用,可减少主节点的从节点,再基于从节点做slave
    - 3.AOF持久化导致Redis阻塞,AOF将指令按照顺序写入日志文件,会先写到os cache,然后每s用fsync刷新到磁盘上,当磁盘性能较差时,fsync阻塞,导致主节点缓慢
- MongoDB  
(Document)
  - 分布式文件存储的文档数据库:
 

MongoDB 是一个基于分布式文件存储的数据库。由 C++ 语言编写。旨在为 WEB 应用提供可扩展的高性能数据存储解决方案。

MongoDB 是一个介于关系数据库和非关系数据库之间的产品, 是非关系数据库当中功能最丰富, 最像关系数据库的。

存储特性与内部原理:

<https://blog.csdn.net/gaozhigang/article/details/79241044>

Mongo操作数据时, 会先在内存中处理, 然后通过预写事务日志刷新到磁盘, 最终数据是保存在磁盘文件上的, Index 会存在内存中(部分索引), 当内存不足时, 会从文件里查询。

wiredTiger 引擎: 3.0+ ,使用预写事务日志journal, 来确保数据的持久性。

MMAPv1引擎(<3.2默认): mongodb原生的存储引擎, 比较简单, 直接使用系统级的内存映射文件机制 (memory mapped files)

    - 所有的write请求都基于“文档级别”的lock, 因此多个客户端可以同时更新一个collection中的不同文档, 这种更细颗粒度的lock, 可以支撑更高的读写负载和并发量。

wiredTiger每隔60秒(默认) 或者待写入的数据达到2G时, mongodb将对journal文件提交一个checkpoint (检测点, 将内存中的数据变更flush到磁盘中的数据文件中, 并做一个标记点, 表示此前的数据表示已经持久存储在了数据文件中, 此后的数据变更存在于内存和journal日志)。

对于write操作, 首先被持久写入journal, 然后在内存中保存变更数据, 条件满足后提交一个新的检测点, 即检测点之前的数据只是在journal中持久存储, 但并没有在mongodb的数据文件中持久化, 延迟持久化可以提升磁盘效率, 如果在提交checkpoint之前, mongodb异常退出, 此后再次启动可以根据journal日志恢复数据。

journal日志默认每个100毫秒同步磁盘一次, 每100M数据生成一个新的journal文件, journal默认使用了snappy压缩, 检测点创建后, 此前的journal日志即可清除。
  - MMAPV1在lock的并发级别上, 支持到collection级别, 所以对于同一个collection同时只能有一个write操作执行。

MMAPv1 为了确保数据的安全性, mongodb将所有的变更操作写入journal

并间歇性的持久到磁盘上，对于实际数据文件将延迟写入，和wiredTiger一样journal也是用于数据恢复。

所有的记录在磁盘上连续存储，当一个document尺寸变大时，mongodb需要重新分配一个新的记录（旧的record标记删除，新的record在文件尾部重新分配空间），这意味着mongodb同时还需要更新此文档的索引（指向新的record的offset），与in-place update相比，将消耗更多的时间和存储开支。由此可见，如果你的mongodb的使用场景中有大量的这种update，那么或许MMAPv1引擎并不太适合，同时也反映出如果document没有索引，是无法保证document在read中的顺序（即自然顺序）。

- 一个mongodb中可以建立多个数据库。

MongoDB的默认数据库为"db"，该数据库存储在data目录中。

MongoDB的单个实例可以容纳多个独立的数据库，每一个都有自己的集合和权限，不同的数据库也放置在不同的文件中。

- 适用场景:适合Bson及嵌套类型的数据,及不确定字段数的场景

- 数据库名(DB)

- 数据库名可以是满足以下条件的任意UTF-8字符串
- 不能是空字符串 ("")。
- 不得含有' ' (空格)、.、\$、/、\和\0 (空字符)。
- 应全部小写。
- 最多64字节。
- 保留名
  - admin: 从权限的角度来看，这是"root"数据库。要是将一个用户添加到这个数据库，这个用户自动继承所有数据库的权限。一些特定的服务器端命令也只能从这个数据库运行，比如列出所有的数据库或者关闭服务器。
  - local: 这个数据永远不会被复制，可以用来存储限于本地单台服务器的任意集合
  - config: 当Mongo用于分片设置时，config数据库在内部使用，用于保存分片的相关信息。

- 文档(Document)

- 文档是一组键值(key-value)对(即 BSON)。MongoDB 的文档不需要设置相同的字段，并且相同的字段不需要相同的数据类型
- 文档中的键/值对是有序的。
- 文档中的值不仅可以是在双引号里面的字符串，还可以是其他几种数据类型（甚至可以是整个嵌入的文档）。
- MongoDB区分类型和大小写。
- MongoDB的文档不能有重复的键。
- 文档的键是字符串。除了少数例外情况，键可以使用任意UTF-8字符。
- 命名规范
  - 键不能含有\0 (空字符)。这个字符用来表示键的结尾。
  - .和\$有特别的意义，只有在特定环境下才能使用。

- 以下划线"\_"开头的键是保留的(不是严格要求的)。
- 集合(Collection)
  - 命名规范
    - 集合名不能是空字符串""。
    - 集合名不能含有\0字符（空字符），这个字符表示集合名的结尾。
    - 集合名不能以"system."开头，这是为系统集合保留的前缀。
    - 用户创建的集合名字不能含有保留字符。有些驱动程序的确支持在集合名里面包含，这是因为某些系统生成的集合中包含该字符。除非你要访问这种系统创建的集合，否则千万不要在名字里出现\$。
  - capped collections(固定大小集合)
    - db.createCollection("mycoll", {capped:true, size:100000})
- 元数据
  - 数据库的信息是存储在集合中。它们使用了系统的命名空间：  
dbname.system.\*
    - dbname.system.namespaces 列出所有名字空间。
    - dbname.system.indexes 列出所有索引。
    - dbname.system.profile 包含数据库概要(profile)信息。
    - dbname.system.users 列出所有可访问数据库的用户。
    - dbname.local.sources 包含复制对端（slave）的服务器信息和状态。
- 基础数据类型
  - 数据存储为一个文档BSON(Binary JSON)，类似于JSON对象
  - ObjectId：对象 ID。用于创建文档的 ID。可以很快的去生成和排序，包含 12 bytes。  
 ObjectId 是一个12字节 BSON 类型数据，有以下格式：  
 前4个字节表示时间戳  
 接下来的3个字节是机器标识码  
 紧接的两个字节由进程id组成（PID）  
 最后三个字节是随机数。
    - 生成ObjectId(): ObjectId() | new ObjectId()

```
var newObject = ObjectId()
newObject.getTimestamp()
ISODate("2017-11-25T07:21:10Z")
  - 获取ObjectId字符串: ObjectId().str
newObject.str
5a1919e63df83ce79df8b38f
  - String: 字符串, UTF-8编码
  - Integer: 整型数值
  - Boolean: 布尔型
  - Double: 双精度浮点型
  - Min/Max keys: 将一个值与BSON元素的最低值和最高值相对比
  - Array: 用于将数据或列表或多个值存储为1个键
  - Timestamp: 时间戳, 记录文件档修改或添加的具体时间
  - Object: 用于内嵌文档
```



- Null: 用于创建空值
- Symbol: 符号。基本上等同于String，不同的是，一般用于采用特殊符号类型的语言
- Date: 日期。用于UNIX时间格式存储当前时间或日期。
  - > var mydate1 = new Date() //格林尼治时间

mydate1

ISODate("2018-03-04T14:58:51.233Z")

typeof mydate1

object

- > var mydate2 = ISODate() //格林尼治时间

mydate2

ISODate("2018-03-04T15:00:45.479Z")

typeof mydate2

object

- > var mydate1str = mydate1.toString()

mydate1str

Sun Mar 04 2018 14:58:51 GMT+0000 (UTC)

typeof mydate1str

string

- > Date()

Sun Mar 04 2018 15:02:59 GMT+0000 (UTC)

- Binary Data: 二进制数据
- Code: 代码类型，用于在文档中存储JavaScript代码
- Regular expression: 正则表达式

#### ■ 连接

- mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]]
  - <https://docs.mongodb.com/manual/reference/connection-string/>
  - mongodb:// 这是固定的格式，必须要指定。
  - username:password@ 可选项，如果设置，在连接数据库服务器之后，驱动都会尝试登陆这个数据库
  - host1 必须的指定至少一个host, host1 是这个URI唯一要填写的。它指定了要连接服务器的地址。如果要连接复制集，请指定多个主机地址。
  - port 可选的指定端口，如果不填，默认为27017
  - /database 如果指定username:password@，连接并验证登陆指定数据库。若不指定，默认打开 test 数据库。
  - ?options 是连接选项。如果不使用/database，则前面需要加上/。所有连接选项都是键值对name=value，键值对之间通过&或; (分号) 隔开
    - replicaSet=name 验证replica set的名称。  
Impliesconnect=replicaSet.
    - slaveOk=true|false
      - true:在connect=direct模式下，驱动会连接第一台机器，即使这台服务器不是主。在connect=replicaSet模式下，驱动会



发送所有的写请求到主并且把读取操作分布在其他从服务器。

- false: 在 connect=direct模式下, 驱动会自动找寻主服务器. 在connect=replicaSet 模式下, 驱动仅仅连接主服务器, 并且所有的读写命令都连接到主服务器。
- tls=true | false  
ssl=true | false
  - 是否使用SSL连接
- safe=true | false
  - true: 在执行更新操作之后, 驱动都会发送getLastError命令来确保更新成功。(还要参考 wtimeoutMS).
  - false: 在每次更新之后, 驱动不会发送getLastError来确保更新成功。
- w=n 驱动添加 { w : n } 到getLastError命令. 应用于safe=true。
- wtimeoutMS=ms 驱动添加 { wtimeout : ms } 到 getlasterror 命令. 应用于 safe=true.
- fsync=true | false
  - true: 驱动添加 { fsync : true } 到 getlasterror 命令.应用于 safe=true.
  - false: 驱动不会添加到getLastError命令中。
- journal=true | false 如果设置为 true, 同步到 journal (在提交到数据库前写入到实体中). 应用于 safe=true
- connectTimeoutMS=ms 可以打开连接的时间。
- socketTimeoutMS=ms 发送和接受sockets的时间。
- 连接实例
  - 使用用户名fred, 密码foobar登录localhost的baz数据库。  
mongodb://fred:foobar@localhost/baz
  - 连接 replica pair, 服务器1为example1.com服务器2为example2。  
mongodb://example1.com:27017,example2.com:27017
  - 连接 replica set 三台服务器, 写入操作应用在主服务器 并且分布查询到从服务器。  
mongodb://host1,host2,host3/?slaveOk=true
    - 连接 replica set 三台服务器, 写入操作应用在主服务器 并且分布查询到从服务器。
  - 直接连接第一个服务器, 无论是replica set一部分或者主服务器或者从服务器。  
mongodb://host1,host2,host3/?connect=direct;slaveOk=true
  - 以安全模式连接到replica set, 并且等待至少两个复制服务器成功写入, 超时时间设置为2秒。  
mongodb://host1,host2,host3/?  
safe=true;w=2;wtimeoutMS=2000

- 基本概念

- database: 数据库
- collection: 数据库表/集合
- document: 数据记录行/文档
- field: 数据字段/域
- index: 索引
- primary key: 主键, Mongo自动将\_id(ObjectId)字段设置为主键

- 用法

- DB相关命令

- show dbs 显示所有数据的列表。

- ```
show dbs
local 0.078GB
test 0.078GB
```

- db 显示当前数据库对象或集合。

- use <database\_name> 连接到一个指定的数据库(若不存在则在插入文档后创建)。

- ```
use local
switched to db local
db
local
```

- db.dropDatabase() 删除数据库

- Collection相关命令

- show collections | show tables | db.getCollectionNames() 查看所有集合

- db.createCollection(name, options) 创建集合

- 集合可以不显示创建, 插入文档时自动创建

- capped 布尔 (可选)

- 如果为 true, 则创建固定集合。固定集合是指有着固定大小的集合, 当达到最大值时, 它会自动覆盖最早的文档。

- 当该值为 true 时, 必须指定 size 参数。

- 判断是否为固定集合

- db.cappedLogCollection.isCapped()

- 返序返回集合内容

- db.cappedLogCollection.find().sort({\$natural:-1})

- autoIndexId 布尔 (可选)

- 从3.2版开始不推荐使用。

- 如为 false, 不在 \_id 字段创建索引。默认为 true。

- size 数值 (可选)

- 为固定集合指定一个最大值, 以千字节计 (KB)。

- 如果 capped 为 true, 也需要指定该字段。

- max 数值 (可选) 指定固定集合中包含文档的最大数量。

- db.<collection\_name>.drop() 删除集合
- Document相关命令
  - 使用.操作符操作 嵌入式文档中的字段(Object 或Array)
  - db.col.find().pretty() 格式化显示结果
  - db.collection.find(query, projection) 查询文档
    - query：可选，使用查询操作符指定查询条件
    - projection：可选，使用投影操作符指定返回的键。查询时返回文档中所有键值，只需省略该参数即可（默认省略）。1返回,0:不返回

```

- >db.collection.findOne(<query>},{field1:1|0})
- db.collection.findOne(query, projection) 查询文档，只返回一个文档
- where 条件操作符
  - 等于 {<key>:<value>}
    - db.col.find({"by":"菜鸟教程"}).pretty()
  - 小于 {<key>:{$lt:<value>}}
    - db.col.find({"likes":{$lt:50}}).pretty()
  - 小于或等于 {<key>:{$lte:<value>}}
    - db.col.find({"likes":{$lte:50}}).pretty()
  - 大于 {<key>:{$gt:<value>}}
    - db.col.find({"likes":{$gt:50}}).pretty()
  - 大于或等于 {<key>:{$gte:<value>}}
    - db.col.find({"likes":{$gte:50}}).pretty()
  - 不等于 {<key>:{$ne:<value>}}
    - db.col.find({"likes":{$ne:50}}).pretty()
- or
  - $or: [ {key1: value1}, {key2:value2}]
- $type 操作符
-
https://docs.mongodb.com/manual/reference/operator/query/type/index.htm
1
  类型，值核对表
  - db.col.find({"title" : {$type : 2}})

```

或

- ```

db.col.find({"title" : {$type : 'string'}})
- limit 读取指定数量的数据记录
  - db.collection.find().limit(NUMBER)
- skip 跳过指定数量的数据
  - db.collection.find().limit(NUMBER).skip(NUMBER)
- sort 排序( 1:升序, -1:降序)
  - db.collection.find().sort({field1:1,field2:-1})
- $in $nin
- 正则表达式 $regex
  - 注意事项
- 正则表达式中使用变量。一定要使用eval将组合的字符串进行转换，不能直接将字符

```

串拼接后传入给表达式。否则没有报错信息，只是结果为空！

```
var name=eval("/" + 变量值key +"/i");
```

- 模糊查询包含title关键词, 且不区分大小写

```
title:eval("/"+title+"/i") // 等同于 title:{$regex:title,$option:"$i"}
```

- regex操作符
  - {:\$regex:/pattern/, \$options:""}
  - {:\$regex:'pattern',\$options:""}
  - {:\$regex:/pattern/}
- 正则表达式对象
  - {:/pattern/}
- options
  - 包括i, m, x以及s四个选项
    - i 忽略大小写, {{\$regex/pattern/i}}, 设置i选项后, 模式中的字母会进行大小写不敏感匹配。
    - m 多行匹配模式, {{\$regex/pattern/\$options:'m'}}, m选项会更改^和\$元字符的默认行为, 分别使用与行的开头和结尾匹配, 而不是与输入字符串的开头和结尾匹配。
    - x 忽略非转义的空白字符, {{\$regex:/pattern/\$options:'x'}}, 设置x选项后, 正则表达式中的非转义的空白字符将被忽略, 同时井号(#)被解释为注释的开头注, 只能显式位于option选项中。
    - s 单行匹配模式{{\$regex:/pattern/\$options:'s'}}, 设置s选项后, 会改变模式中的点号(.)元字符的默认行为, 它会匹配所有字符, 包括换行符(\n), 只能显式位于option选项中。
  - 使用\$regex操作符时, 需要注意下面几个问题:
    - i, m, x, s可以组合使用, 例如:{name:{\$regex:/jk/\$options:"si"}}
    - 在设置索引的字段上进行正则匹配可以提高查询速度, 而且当正则表达式使用的是前缀表达式时, 查询速度会进一步提高, 例如:{name:{\$regex:/^joe/}}
  - \$regex与正则表达式对象的区别:
    - 在\$in操作符中只能使用正则表达式对象, 例如:{name:{\$in:/^joe/i,/^jack/}}
    - 在使用隐式的\$and操作符中, 只能使用\$regex, 例如:{name:{\$regex:/^jo/i, \$nin:['john']}}
  - 当option选项中包含X或S选项时, 只能使用\$regex, 例如:{name:{\$regex:/m.line/\$options:"si"}}

```
- db.<collection_name>.insert(document) 插入文档
- db.<collection_name>.insertOne() 插入1个文档
- db.collection.insertOne(
,
{
writeConcern:
}
)
- 可以将数据定义为一个变量, 再插入
- > document=({title: 'var obj', likes: 100});

{ "title" : "var obj", "likes" : 100 }
db.col.insert(document)
WriteResult({ "nInserted" : 1 })
- db.collection.insertMany() 插入多个文档
```

```

- db.collection.insertMany(
[ <document 1> , <document 2> , ... ],
  {
    writeConcern: ,
ordered:
  }
)

```

- document: 要写入的文档。

- writeConcern: 写入策略, 默认为 1, 即要求确认写操作, 0 是不要求。

- ordered: 指定是否按顺序写入, 默认 true, 按顺序写入。

#### ■ 更新文档

```

db.collection.update(
,
,
{
  upsert: ,
  multi: ,
writeConcern:
}
)

```

#### ■ db.collection.updateOne(, , )

```
db.collection.updateMany(, , )
```

```
db.collection.replaceOne(, , )
```

- query: update的查询条件, 类似sql update查询内where后面的。

- update: update的对象和一些更新的操作符 (如\$, \$inc...) 等, 也可以理解为sql update查询内set后面的

- \$inc 更新操作符

- \$currentDate 设置字段为当前日期

- db.customers.updateOne(

```

{ _id: 1 },
{
  $currentDate: {
    lastModified: true,
    "cancellation.date": { $type: "timestamp" }
  },
  $set: {
    "cancellation.reason": "user request",
    status: "D"
  }
}
)

```

- \$inc 为字段增加指定值

- { \$inc: { : , : , ... } }

- \$min 当指定值小于当前值时, 将值更新为指定值

- { \$min: { : , ... } }

- \$max 当指定值大于当前值时, 将值更新为指定值

- { \$max: { : , ... } }
- \$mul 当前值与指定值相乘，若当前字段不存在，则用0相乘

```
- { "_id" : 1, "item" : "ABC", "price" :
  NumberDecimal("10.99"), "qty" : 25 }
```

```
db.products.update(
  { id: 1 },
  { $mul: { price: NumberDecimal("1.25"), qty: 2 } }
)
  { "id" : 1, "item" : "ABC", "price" : NumberDecimal("13.7375"), "qty" : 50 }
    - $rename 重命名字段名
      - { $rename: { : , : , ... } }
    - $setOnInsert 当upsert=1切为插入数据时，插入指定数据
    - db.collection.update(
,
{ $setOnInsert: { : , ... } },
{ upsert: true }
)
    - $unset 删除字段
      - { $unset: { : "" , ... } }
    - array 更新操作符
    - $ 更新指定数组中的指定元素为新元素，而无需指定具体索引
.$:val
  $.field:val
    - db.collection.update(
{ : value ... },
{ : { ". $" : value } }
)
    - >db.students.insert([
  { "id" : 1, "grades" : [ 85, 80, 80 ] },
  { "id" : 2, "grades" : [ 88, 90, 92 ] },
  { "_id" : 3, "grades" : [ 85, 100, 90 ] }
])
db.students.updateOne(
  { _id: 1, grades: 80 },
  { $set: { "grades.$" : 82 } }
)

{ "id" : 1, "grades" : [ 85, 82, 80 ] }
{ "id" : 2, "grades" : [ 88, 90, 92 ] }
  { "_id" : 3, "grades" : [ 85, 100, 90 ] }
    - db.collection.update(
{ },
{ : { "array.$field" : value } }
)
    - {
```

```

    _id: 4,
    grades: [
      { grade: 80, mean: 75, std: 8 },
      { grade: 85, mean: 90, std: 5 },
      { grade: 85, mean: 85, std: 8 }
    ]
  }
  db.students.updateOne(
    { _id: 4, "grades.grade": 85 },
    { $set: { "grades.$.std" : 6 } }
  )

{
  "id": 4,
  "grades": [
    { "grade": 80, "mean": 75, "std": 8 },
    { "grade": 85, "mean": 90, "std": 6 },
    { "grade": 85, "mean": 85, "std": 8 }
  ]
}

- db.students.updateOne(
{
  _id: 5,
  grades: { $elemMatch: { grade: { $lte: 90 }, mean: { $gt: 80 } } }
},
{ $set: { "grades.$.std" : 6 } }
)

- [] 更新字段中的所有元素
.$[]:val
- { { ".$[]" : value } }
db.collection.updateMany(
{ },
{ : { ".$[]" : value } }
)

- [] 按条件更新数组中的部分数据
- db.collection.updateMany(
{ },
{ : { ".$[]" : value } },
{ arrayFilters: [ { : } ] }
)

- { "id" : 1, "grades" : [ 95, 92, 90 ] }
{ "_id" : 2, "grades" : [ 98, 100, 102 ] }

{ "_id" : 3, "grades" : [ 95, 110, 100 ] }
db.students.update(
{ },
{ $set: { "grades.$[element]" : 100 } },

```

```

    { multi: true,
      arrayFilters: [ { "element": { $gte: 100 } } ]
    }
  )

{ "id" : 1, "grades" : [ 95, 92, 90 ] }
{ "id" : 2, "grades" : [ 98, 100, 100 ] }
  { "_id" : 3, "grades" : [ 95, 100, 100 ] }
    - $addToSet 将元素添加到数组中(若元素中已存在, 则不插入)
    - db.collection.update(
      { },
      { $addToSet: { field: val } }
    )
      - db.foo.update(
        { _id: 1 },
        { $addToSet: { colors: "c" } }
      )
        - $pop 删除数组中的第一个或最后一个元素
        - db.collection.update( { },
          { $pop: { field: -1 | 1 } } )
          - $pull 删除数组中指定条件的元素, 或完全匹配的1个元素
          - { $pull: { : <value | condition>, : <value | condition>, ... } }
          - {
            _id: 1,
            fruits: [ "apples", "pears", "oranges", "grapes", "bananas" ],
            vegetables: [ "carrots", "celery", "squash", "carrots" ]
          }
          {
            _id: 2,
            fruits: [ "plums", "kiwis", "oranges", "bananas", "apples" ],
            vegetables: [ "broccoli", "zucchini", "carrots", "onions" ]
          }
        }
      db.stores.update(
        { },
        { $pull: { fruits: { $in: [ "apples", "oranges" ] }, vegetables: "carrots" } },
        { multi: true }
      )

{
  "id" : 1,
  "fruits" : [ "pears", "grapes", "bananas" ],
  "vegetables" : [ "celery", "squash" ]
}

{
  "id" : 2,
  "fruits" : [ "plums", "kiwis", "bananas" ],
  "vegetables" : [ "broccoli", "zucchini", "onions" ]
}

```



```
}
```

- \$pullAll 从数组中删除多个指定元素

- { \$pullAll: { : [ , ... ], ... } }

- \$push 将元素添加到数组中

- { \$push: { : , ... } }

- { \$push: { : { : , ... }, ... } }

- 添加多个元素到数组中

```
db.students.update(
```

```
{ id: 5 },
```

```
{
```

```
  $push: {
```

```
    quizzes: {
```

```
      $each: [ { wk: 5, score: 8 }, { wk: 6, score: 7 }, { wk: 7, score: 6 } ],
```

```
      $sort: { score: -1 }
```

```
    }
```

```
  }
```

```
}
```

```
)
```

- 修饰符

- \$each 将多个值添加到数组中, 用于\$addToSet,\$push

- { \$addToSet: { : { \$each: [ , ... ] } } }

- { \$push: { : { \$each: [ , ... ] } } }

- \$position(NUMBER), 与\$each配合使用 将元素插入到指定位置, 用于

\$push, 下标从0开始

- {

```
  $push: {
```

```
    : {
```

```
      $each: [ , , ... ],
```

```
      $position:
```

```
    }
```

```
}
```

```
}
```

- \$slice(NUMBER) 将数组截取为指定数量的大小

, 用于\$push

- {

```
  $push: {
```

```
    : {
```

```
      $each: [ , , ... ],
```

```
      $slice:
```

```
    }
```

```
}
```

```
}
```

- \$sort 对数组进行排序, 与\$each配合使用, 用于\$push

- {

```
  $push: {
```

```
    : {
```

```

    $each: [ , , ... ],
    $sort:
  }
}
}

```

- `upsert`: 可选, 这个参数的意思是, 如果不存在`update`的记录, 是否插入`objNew`, `true`为插入, 默认是`false`, 不插入。

- `multi`: 可选, `mongodb` 默认是`false`, 只更新找到的第一条记录, 如果这个参数为`true`, 就把按条件查出来多条记录全部更新。

- `writeConcern`: 可选, 抛出异常的级别

- 实例

- 更新匹配到的所有记录

```
db.col.update({'title':'MongoDB 教程'},{$set: {'title':'MongoDB'}},{multi:true})
```

- 全部更新

```
db.col.update( { "count" : { $gt : 3 } } , { $set : { "test2" : "OK" } }, false, true );
```

- 删除文档

```
db.collection.remove(
```

```

,
{
  justOne: ,
  writeConcern:
}
)

```

- `query`: (可选) 删除的文档的条件。

- `justOne`: (可选) 如果设为`true` 或 `1`, 则只删除一个文档, 如果不设置该参数, 或使用默认值`false`, 则删除所有匹配条件的文档。

- `writeConcern`: (可选) 抛出异常的级别。

- `index`索引相关命令

- `db.collection.createIndex(keys, options)` 创建索引

- `options`

- `name` string

索引的名称。如果未指定, `MongoDB`的通过连接索引的字段名和排序顺序生成一个索引名称。

- `background` Boolean

建索引过程会阻塞其它数据库操作, `background`可指定以后台方式创建索引, 即增加"`background`" 可选参数。"`background`" 默认值为`false`。

- `unique` Boolean

建立的索引是否唯一。指定为`true`创建唯一索引。默认值为`false`。

- `sparse` Boolean

对文档中不存在的字段数据不启用索引; 这个参数需要特别注意, 如果设置为`true`的话, 在索引字段中不会查询出不包含对应字段的文档。默认值为`false`。

- `expireAfterSeconds` integer

指定一个以秒为单位的数值, 完成 TTL 设定, 设定集合的生存时间。

- `v` index version

索引的版本号。默认的索引版本取决于`mongod`创建索引时运行的版本。

- `weights` document

索引权重值，数值在 1 到 99,999 之间，表示该索引相对于其他索引字段的得分权重。

- default\_language string

对于文本索引，该参数决定了停用词及词干和词器的规则的列表。默认为英语

- language\_override string

对于文本索引，该参数指定了包含在文档中的字段名，语言覆盖默认的language，默认值为language。

- db.collection.getIndexes() 查询集合索引

- db.col.dropIndex("indexName") 删除指定索引

- db.collection.dropIndexes() 删除所有索引

- db.collection.totalIndexSize() 查看集合索引大小

- 全文索引

- MongoDB 在 2.6 版本以后是默认开启全文检索的，如果你使用之前的版本，你需要使用以下代码来启用全文检索：

- >db.adminCommand({setParameter:true,textSearchEnabled:true})

或者使用命令：

```
mongod --setParameter textSearchEnabled=true
```

- 创建全文索引

- db.collection.createIndex({post\_text:"text"})

- 查询全文索引

- db.collection.find({\$text:{\$search:"runoob"}})

- aggregate 聚合操作

- db.collection.aggregate(AGGREGATE\_OPERATION)

- AGGREGATE\_OPERATION

- \$project：修改输入文档的结构。可以用来重命名、增加或删除域，也可以用于创建计算结果以及嵌套文档。

- 结果中就只剩下title和author2个字段了（id默认被包含）

```
db.article.aggregate(
```

```
{ $project : {
```

```
  id : 0 ,
```

```
  title : 1 ,
```

```
  author : 1
```

```
});
```

- \$match：用于过滤数据，只输出符合条件的文档。\$match使用MongoDB的标准查询操作。

- \$match用于获取分数大于70小于或等于90记录，然后将符合条件的记录送到下一阶段\$group管道操作符进行处理。

```
db.articles.aggregate( [
```

```
{ $match : { score : { $gt : 70, $lte : 90 } } },
```

```
{ $group: { _id: null,
```

```
  count: { $sum: 1 } } }
```

```
]);
```

- \$limit：用来限制MongoDB聚合管道返回的文档数。

- \$skip：在聚合管道中跳过指定数量的文档，并返回余下的文档。

- \$unwind：将文档中的某一个数组类型字段拆分成多条，每条包含数组中的一个值。

- \$group：将集中的文档分组，可用于统计结果。

- \$sort: 将输入文档排序后输出。
- \$geoNear: 输出接近某一地理位置的有序文档。
- MongoDB中的关系
  - 1. 嵌入文档
  - 2类数据维护在1条数据中

```
{
  "id": ObjectId("52ffc33cd85242f436000001"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin",
  "address": [
    {
      "building": "22 A, Indiana Apt",
      "pincode": 123456,
      "city": "Los Angeles",
      "state": "California"
    },
    {
      "building": "170 A, Acropolis Apt",
      "pincode": 456789,
      "city": "Chicago",
      "state": "Illinois"
    }
  ]
}
```

- 2. 引用式关系
  - 两次查询，第一次查询用户地址的对象id (ObjectId)，第二次通过查询的id获取用户的详细地址信息。

```
{
  "id": ObjectId("52ffc33cd85242f436000001"),
  "contact": "987654321",
  "dob": "01-01-1991",
  "name": "Tom Benzamin",
  "address_ids": [
    ObjectId("52ffc4a5d85242602e000000"),
    ObjectId("52ffc4a5d85242602e000001")
  ]
}
```

- 3. 数据库应用DBRef
  - 指定集合，一个文档从多个集合引用文档，我们应该使用 DBRefs。
  - 形式: { \$ref: , \$id: , \$db: }
    - \$ref: 集合名称
    - \$id: 引用的id
    - \$db: 数据库名称, 可选参数
  - {
 

```
"id": ObjectId("53402597d852426020000002"),
          "address": {
```

```
"$ref": "address_home",
"$id": ObjectId("534009e4d852427820000002"),
"$db": "runoob",
"contact": "987654321",
"dob": "01-01-1991",
"name": "Tom Benzamin"
}
```

```
- >var user = db.users.findOne({"name":"Tom Benzamin"})
```

```
var dbRef = user.address
db[dbRef.$ref].findOne({"_id":(dbRef.$id)})
```

#### ■ 集群 (ReplicaSet副本集)

##### ■ 原理

- mongodb的复制至少需要两个节点。其中一个主节点，负责处理客户端请求，其余的都是从节点，负责复制主节点上的数据。

mongodb各个节点常见的搭配方式为：一主一从、一主多从。

主节点记录在其上的所有操作oplog，从节点定期轮询主节点获取这些操作，然后对自己的数据副本执行这些操作，从而保证从节点的数据与主节点一致。

副本集在主机宕机后，副本会接管主节点成为主节点，不会出现宕机的情况。

##### ■ 设置

- - mongod --port "PORT" --dbpath "YOUR\_DB\_DATA\_PATH" --replSet "REPLICA\_SET\_INSTANCE\_NAME"
  - mongod --port 27017 --dbpath "D:\set up\mongodb\data" --replSet rs0
- - 登录client,执行 rs.initiate()来启动一个新的副本集
  - rs.conf()来查看副本集的配置
  - rs.status() 查看副本集状态
  - db.isMaster() 判断服务是否是主节点
- - rs.add()方法来添加副本集的成员
  - rs.add(HOST\_NAME:PORT)
  - rs.add("mongodb1.net:27017")

#### ■ 集群(Shard分片)

- <https://www.runoob.com/mongodb/mongodb-sharding.html>
- 监控
  - mongostat
    - mongostat是mongodb自带的状态检测工具，在命令行下使用。它会间隔固定时间获取mongodb的当前运行状态，并输出。
  - mongotop
    - mongotop也是mongodb下的一个内置工具，mongotop提供了一个方法，用来跟踪一个MongoDB的实例，查看哪些大量的时间花费在读取和写入数据。mongotop提供每个集合的水平统计数据。默认情况下，mongotop返回值的每一秒。

#### ■ 查询分析

- MongoDB 查询分析可以确保我们所建立的索引是否有效，是查询语句性能分析的重要工具。

MongoDB 查询分析常用函数有：explain() 和 hint()。

```
- explain()
  - db.collection.find().explain()
    - {
      "cursor": "BtreeCursor gender_1_user_name_1",
      "isMultiKey": false,
      "n": 1,
      "nscannedObjects": 0,
      "nscanned": 1,
      "nscannedObjectsAllPlans": 0,
      "nscannedAllPlans": 1,
      "scanAndOrder": false,
      "indexOnly": true,
      "nYields": 0,
      "nChunkSkips": 0,
      "millis": 0,
      "indexBounds": {
        "gender": [
          [
            "M",
            "M"
          ]
        ],
        "user_name": [
          [
            {
              "$minElement": 1
            },
            {
              "$maxElement": 1
            }
          ]
        ]
      }
    }
```

```
}  
}
```

- 返回字段说明

- indexOnly: 字段为 true , 表示我们使用了索引。

- cursor: 因为这个查询使用了索引, MongoDB 中索引存储在B树结构中, 所以这也是使用了 BtreeCursor 类型的游标。如果没有使用索引, 游标的类型是 BasicCursor。这个键还会给出你所使用的索引的名称, 你通过这个名称可以查看当前数据库下的system.indexes集合(系统自动创建, 由于存储索引信息, 这个稍微会提到)来得到索引的详细信息。

- n: 当前查询返回的文档数量。

- nscanned/nscannedObjects: 表明当前这次查询一共扫描了集合中多少个文档, 我们的目的是, 让这个数值和返回文档的数量越接近越好。

- millis: 当前查询所需时间, 毫秒数。

- indexBounds: 当前查询具体使用的索引。

- hint()

使用 hint 来强制 MongoDB 使用一个指定的索引

- db.collection.find().hint({field1:1,field2:1})

■ Mongo中的原子操作

■ mongodb不支持事务, 但是mongodb提供了许多原子操作, 比如文档的保存, 修改, 删除等, 都是原子操作

■ 原子操作命令

- \$set
- \$unset
- \$inc
- \$push
- \$pushAll
- \$pull
- \$addToSet
- \$pop
- \$rename
- \$bit

■ 高级索引

■ 查询时, 无需指定索引顺序, mongo会自动优化

■ 索引数组字段

■ 在数组中创建索引, 默认对数组中的每个字段依次建立索引

■ 索引子文档字段

■ db.collection.createIndex({"obj1.field1":1,"obj2.field2":1})

■ 索引限制

■ 额外开销: 每个索引占用一定存储空间在, 进行插入, 更新和删除操作时也需要对索引进行操作

■ 内存(RAM)使用

■ 由于索引是存储在内存(RAM)中, 应该确保该索引的大小不超过内存的限制

- 查询限制(索引不生效)
  - 正则表达式及非操作符, 如 \$nin, \$not, 等。
  - 算术运算符, 如 \$mod, 等。
  - \$where 子句
- 最大范围
  - 集合中索引不能超过64个
  - 索引名的长度不能超过128个字符
  - 一个复合索引最多可以有31个字段
- MapReduce

- Map-Reduce是一种计算模型, 简单的说就是将大批量的工作(数据)分解(MAP)执行, 然后再将结果合并成最终结果(REDUCE)。
- 基本语法:

```
db.collection.mapReduce(
function() {emit(key,value);}, //map 函数
function(key,values) {return reduceFunction}, //reduce 函数
{
  out: collection,
  query: document,
  sort: document,
  limit: number
}
)<.find()查询返回结果>
```

- 使用 MapReduce 要实现两个函数 Map 函数和 Reduce 函数,Map 函数调用 emit(key, value), 遍历 collection 中所有的记录, 将 key 与 value 传递给 Reduce 函数进行处理。

Map 函数必须调用 emit(key, value) 返回键值对。

- 实例:

```
db.posts.mapReduce(
function() { emit(this.user_name,1); },
function(key, values) {return Array.sum(values)},
{
  query:{status:"active"},
  out:"post_total"
}
)
```

- 参数说明:

- map : 映射函数 (生成键值对序列,作为 reduce 函数参数)。
- reduce 统计函数, reduce函数的任务就是将key-values变成key-value把values数组变成一个单一的值value。
- out 统计结果存放集合 (不指定则使用临时集合,在客户端断开后自动删除)。
- 创建临时集合

```
out: { inline: 1 }
```

- 设置了 {inline:1} 将不会创建集合, 整个 Map/Reduce 的操作将会在内存中进行。



注意，这个选项只有在结果集单个文档大小在16MB限制范围内时才有效。

- query 一个筛选条件，只有满足条件的文档才会调用map函数。（query。limit，sort可以随意组合）

- sort 和limit结合的sort排序参数（也是在发往map函数前给文档排序），可以优化分组机制

- limit 发往map函数的文档数量的上限（要是没有limit，单独使用sort的用处不大）

- 以下实例在集合 orders 中查找 status:"A" 的数据，并根据 cust\_id 来分组，并计算 amount 的总和。

- 返回值说明

- {  
 "result": "post\_total",  
 "timeMillis": 23,  
 "counts": {  
 "input": 5,  
 "emit": 5,  
 "reduce": 1,  
 "output": 2  
 },  
 "ok": 1  
}

- result: 储存结果的collection的名字,这是个临时集合，MapReduce的连接关闭后自动就被删除了。

- timeMillis: 执行花费的时间，毫秒为单位

- input: 满足条件被发送到map函数的文档个数

- emit: 在map函数中emit被调用的次数，也就是所有集合中的数据总量

- output: 结果集合中的文档个数（count对调试非常有帮助）

- ok: 是否成功，成功为1

- err: 如果失败，这里可以有失败原因，不过从经验上来看，原因比较模糊，作用不大

## ■ 自增

- Mongo没有自增功能，但可使用Javascript函数实现自增序列

- ```
function getNextSequenceValue(sequenceName){  
  var sequenceDocument = db.counters.findAndModify(  
    {  
      query:{id: sequenceName },  
      update: {$inc:{sequence_value:1}},  
      "new":true  
    });  
  return sequenceDocument.sequence_value;  
}  
  
db.products.insert({  
  "id":getNextSequenceValue("productid"),  
  "product_name":"Samsung S3",  
  "category":"mobiles"})
```

- //调用db.system.js.insert()给系统添加自定义函数，函数的语法格式和JS一致
 

```
db.system.js.insert({
  "id": "getNextValue",
  value:function(colName) {
    var res = db.counters.findAndModify({
      query: {"id": colName},
      update: {$inc: {current_value: 1}},
    });
    return res.current_value;
  }
})
db.eval('getNextValue("id")')
```

- 调优

- Memcache
- Neo4j  
(Graph)
- DB区别与选型

## RDMS

- Oracle
  - 基础数据类型
  - 用法
  - 集群
  - 配置
  - 索引
  - 调优
  - 默认事务隔离级别: READ COMMITED
- Mysql
  - 基础数据类型
  - 用法
  - 集群
  - MySQL关注点
    - 逻辑架构和查询过程
    - 索引及数据存储结构
    - 创建索引及索引优化
    - MySQL事务隔离级别及其表现
  - 逻辑架构
    - 客户端层: 通过客户端访问协议连接服务
    - 核心服务层
 

包括查询解析、分析、优化、缓存、内置函数(比如：时间、数学、加密等函数)。所有的跨存储引擎的功能也在这一层实现：存储过程、触发器、视图等

- 连接/线程处理
    - 查询缓存
    - SQL解析器
    - 优化器
    - 存储引擎:提供对引擎操作的API接口
  - MySQL查询过程
    - 1.客户端通过客户端/服务器通信协议连接数据库
 

客户端/服务器通信协议为"半双工":在任一时刻,只能有一段向另一端发送数据,或接收数据,一旦发送消息,接收端只有接收到完整消息才能进行响应  
可设置max\_allowed\_packet参数配置最大允许客户端的数据包大小

      - --查询当前最大数据包配置
 

```
SELECT @@max_allowed_packet;
SHOW VARIABLES LIKE 'max_allowed_packet';
```
    - 根据客户端发送SQL, 查询缓存
 

若命中缓存,则返回结果集;  
只有和缓存完全一致的SQL才能命中缓存(多任何字符都不能命中,如空格,注释等);  
MySQL将缓存放在一个引用表(类似HashMap),通过一个哈希值索引(哈希值通过查询本身、当前要查询的数据库、客户端协议版本号等一些可能影响结果的信息计算得来)  
缓存失效: 当相关表的表(数据或结构)发生变化,则与该表相关的所有缓存数据将失效

[https://mp.weixin.qq.com/s?\\_biz=MzAxMTkwODIyNA==&mid=2247492357&idx=1&sn=fa951ebb3760705242c5f87216297f6a&source=41#wechat\\_redirect](https://mp.weixin.qq.com/s?_biz=MzAxMTkwODIyNA==&mid=2247492357&idx=1&sn=fa951ebb3760705242c5f87216297f6a&source=41#wechat_redirect)
    - 如果查询中包含任何用户自定义函数、存储函数、用户变量、临时表、mysql库中的系统表, 其查询结果都不会被缓存。比如函数NOW()或者CURRENT\_DATE()会因为不同的查询时间, 返回不同的查询结果, 再比如包含CURRENT\_USER或者CONNECTION\_ID()的查询语句会因为不同的用户而返回不同的结果, 将这样的查询结果缓存起来没有任何的意义。
    - 解析器进行语法解析生成解析树,然后预处理,生成新的解析树
  - 4.使用查询优化器对解析树进行优化
  - 5.生成执行计划,查询执行引擎(通过存储引擎暴露的API查询)
- 索引
  - InnoDB 聚集索引: 聚集索引和具体数据存放在B+树的叶子节点上,所以每个表必须会有1个聚集索引,当有主键时,主键为聚集索引;没有主键时,取第一个不为空的字段为聚集索引;反之,以数据库生成的rowid为聚集索引;  
普通索引:除主键之外的索引即普通索引,普通索引的叶子节点存放的是键值和列对应的主键值;  
聚集索引和普通索引是2个独立的索引树
  - 1.<https://mp.weixin.qq.com/s/y0pjtNUZhOW2ZBOy4m-xsA>
  - 2.[https://mp.weixin.qq.com/s/MoIVZjVAIXRj48\\_SreN4MA](https://mp.weixin.qq.com/s/MoIVZjVAIXRj48_SreN4MA)
  - 3.[https://mp.weixin.qq.com/s?\\_biz=MzAxMTkwODIyNA==&mid=2247492357&idx=1&sn=fa951ebb3760705242c5f87216297f6a&source=41#wecha](https://mp.weixin.qq.com/s?_biz=MzAxMTkwODIyNA==&mid=2247492357&idx=1&sn=fa951ebb3760705242c5f87216297f6a&source=41#wecha)

## [t\\_redirect](#)

- 覆盖索引(InnoDB)
  - SQL 查询时,若查询条件中满足索引(innoDB为聚集索引/普通索引,MyISAM为普通索引)列查询,则进行普通索引查询,当SQL查询字段都是该索引的字段值时,可直接返回,不需要再使用对应的主键值去聚集索引中查询其他字段值,这种情况就是索引覆盖
- 回表(InnoDB)
  - 当SQL查询时,查询的结果字段不在索引列中包含时,就需要从普通索引中找到数据对应的主键,再用主键到聚集索引中查询其他字段数据,这种情况下会查询2次索引树,这种情况就是回表
- 稀疏索引
  - 被索引的数据必须是有序的
  - 索引项为一段数据的起始值,该项对应一组连续的数据,一般为与下一个索引项之间的数据
    - 如:2个索引文件  
00001: 中保存了00001-00010的数据  
00011:文件中保存了00011-00020的数据
  - Kafka为稀疏索引
- B+树(B树:Balance tree,一棵平衡二叉树,且节点(含子节点和叶子节点)可以保存多份数据;B+树:在B树的基础上优化而来,B+树的子节点不保存数据,只保存键值和指向叶子节点数据的指针,叶子节点保存键值和数据,节点的记录值通过双线链表连接);

MySQL为什么使用B+树,而不使用B树:

由于普通平衡二叉树(AVL树)节点只能存放1组数据,所以在此基础上发展了B树,B树可以在节点上存放多组数据,且节点和叶子节点都可以存放数据;B+树为节点只存放键值和索引,只有叶子节点存放数据;

由于MySQL数据存储的基本单位是页(默认大小为16K),当采用B树时,所有子节点都可以存放数据,则每页存放的数据量会减少,同时页的数量会增多,就会生成一棵很高很宽的树,同时查询数据时,需要把页内数据加载到内存中处理,读取页时会进行一次磁盘IO操作,由此,会进行大量磁盘IO操作,查询效率会大大降低;

当采用B+树时,B+树的子节点不存具体数据,只存键值和子节点指针,叶子节点存放具体数据,页大小固定的情况下,子节点可以存放更多的键值,而对应的索引树会更低,更宽,进行磁盘IO的情况会更少,因此查询效率会更高

  - 1.<https://mp.weixin.qq.com/s/BWlkrHiB-uP6fDnsxtKU0Q>
  - 2.[https://mp.weixin.qq.com/s/MoIVZjVAIXRj48\\_SreN4MA](https://mp.weixin.qq.com/s/MoIVZjVAIXRj48_SreN4MA)
  - MySQL的数据存储基本单位为页(磁盘基本单位为磁道512B,操作系统基本单位为块4K),默认大小16K
    - 查询页大小
    - show global variables like '%page%';
    - innodb\_page\_size
  - 联合索引

- 联合索引的生效原则是 从前往后依次使用生效，如果中间某个索引没有使用，那么断点前面的索引部分起作用，断点后面的索引没有起作用；
  - 1.对于where条件字段顺序,若where条件里包含联合索引的字段,则会自动优化
  - 2.若联合索引中间字段存在范围值,则也算断点,断点字段后的字段无法应用索引
  - 3.存在orderBy的查询,只有在where条件中包含联合索引前缀字段,且不存在断点字段,orderBy索引才能生效
- where a=3 and b=45 and c=5 .... 这种三个索引顺序使用中间没有断点，全部发挥作用；  
 where a=3 and c=5... 这种情况下b就是断点，a发挥了效果，c没有效果  
 where b=3 and c=4... 这种情况下a就是断点，在a后面的索引都没有发挥作用，这种写法联合索引没有发挥任何效果；  
 where b=45 and a=3 and c=5 .... 这个跟第一个一样，全部发挥作用，abc只要用上了就行，跟写的顺序无关
- 对于复合索引：Mysql从左到右的使用索引中的字段，一个查询可以只使用索引中的一部份，但只能是最左侧部分。例如索引是key index (a,b,c) 。可以支持a | a,b| a,b,c 3种组合进行查找，但不支持 b,c进行查找 .当最左侧字段是常量引用时，索引就十分有效。
- 联合索引情况
  - (0) select \* from mytable where a=3 and b=5 and c=4;  
 abc三个索引都在where条件里面用到了，而且都发挥了作用
  - (1) select \* from mytable where c=4 and b=6 and a=3;  
 这条语句列出来只想说明 mysql没有那么笨，where里面的条件顺序在查询之前会被mysql自动优化，效果跟上一句一样
  - (2) select \* from mytable where a=3 and c=7;  
 a用到索引，b没有用，所以c是没有用到索引效果的
  - (3) select \* from mytable where a=3 and b>7 and c=3;  
 a用到了，b也用到了，c没有用到，这个地方b是范围值，也算断点，只不过自身用到了索引
  - (4) select \* from mytable where b=3 and c=4;  
 因为a索引没有使用，所以这里 bc都没有用上索引效果
  - (5) select \* from mytable where a>4 and b=7 and c=9;  
 a用到了 b没有使用，c没有使用
  - (6) select \* from mytable where a=3 order by b;  
 a用到了索引，b在结果排序中也用到了索引的效果，前面说了，a下面任意一段的b是排好序的
  - (7) select \* from mytable where a=3 order by c;  
 a用到了索引，但是这个地方c没有发挥排序效果，因为中间断点了，使用 explain 可以看到 filesort
  - (8) select \* from mytable where b=3 order by a;  
 b没有用到索引，排序中a也没有发挥索引效果
- 索引失效条件
  - 1.不在索引列上做任何操作(计算,函数,类型转换),会导致全表扫描

- 2.存储引擎不能使用索引范围条件右边的列
- 3.尽量使用索引覆盖的列,减少select \*
- 4.不使用!=,<>,会导致全表扫描

■ is null,is not null 不走索引,导致索引失效

6.like 以左通配符开头会导致全表扫描,如like '%a',可使用右通配符

- 索引类型

- select\_type

- 1.simple 简单select(不使用union或子查询)

2.primary 最外面的select

3.union union中的第二个或后面的select语句

4.dependent union union中的第二个或后面的select语句,取决于外面的查询

5.union result union的结果。

6.subquery 子查询中的第一个select

7.dependent subquery 子查询中的第一个select,取决于外面的查询

8.derived 导出表的select(from子句的子查询)

- Extra与type

- Distinct:一旦MYSQL找到了与行相联合匹配的行,就不再搜索了

Not exists: MYSQL优化了LEFT JOIN,一旦它找到了匹配LEFT JOIN标准的行,就不再搜索了

Range checked for each Record(index map:#):没有找到理想的索引,因此对于从前面表中来的每一个行组合,MYSQL检查使用哪个索引,并用它来从表中返回行。这是使用索引的最慢的连接之一

Using filesort: 看到这个的时候,查询就需要优化了。MYSQL需要进行额外的步骤来发现如何对返回的行排序。它根据连接类型以及存储排序键值和匹配条件的全部行的行指针来排序全部行

Using index: 列数据是从仅仅使用了索引中的信息而没有读取实际的行动的表返回的,这发生在对表的全部的请求列都是同一个索引的部分的时候

Using temporary 看到这个的时候,查询需要优化了。这里,MYSQL需要创建一个临时表来存储结果,这通常发生在对不同的列集进行ORDER BY上,而不是GROUP BY上

Where used 使用了WHERE从句来限制哪些行将与下一张表匹配或者是返回给用户。如果不想返回表中的全部行,并且连接类型ALL或index,这就会发生,或者是查询有问题不同连接类型的解释(按照效率高低的顺序排序

其中type:

如果是Only index,这意味着信息只用索引树中的信息检索出的,这比扫描整个表要快。

如果是where used,就是使用上了where限制。

如果是impossible where 表示用不着where,一般就是没查出来啥。

如果此信息显示Using filesort或者Using temporary的话会很吃力,WHERE和ORDER BY的索引经常无法兼顾,如果按照WHERE来确定索引,那么在ORDER BY时,就必然会引起Using filesort,这就要看是先过滤再排序划算,还是先排序再过滤划算。

- ref:

system 表只有一行: system表。这是const连接类型的特殊情况

const:表中的一个记录的最大值能够匹配这个查询(索引可以是主键或惟一索引)。因为只有一行,这个值实际就是常数,因为MYSQL先读这个值然后把它当做常数来对待

eq\_ref:在连接中,MYSQL在查询使用了索引为主键或惟一键;

ref:这个连接类型只有在查询使用了不是惟一或主键的键或者是这些类型的部分(比如,利用最左边前缀)时发生

range:这个连接类型使用索引返回一个范围中的行,比如使用>或<查找东西时发生的情况

+

index: 这个连接类型对前面的表中的每一个记录联合进行完全扫描(比ALL更好,因为索引一般小于表数据),索引表全表扫描

ALL:这个连接类型对于前面的每一个记录联合进行完全扫描,这一般比较糟糕,应该尽量避免

#### - 创建索引技巧

- 1.选择高纬度的列创建索引,数据列中 不重复值 出现的个数,这个数量越高,维度就越高

- 2.对 where,on,group by,order by 中出现的列使用索引

- 3.对较小的数据列使用索引,这样会使索引文件更小,同时内存中也可以装载更多的索引键

- 4.为较长的字符串使用前缀索引

- ALTER TABLE myIndex ADD INDEX name\_city\_age

(vc\_Name(10),vc\_City,i\_Age);

- 5.不要过多的创建索引,索引需要额外的磁盘空间来保存,且每次增删改都需要操作索引

- 6.使用组合索引可减少索引大小,速度优先于单列索引

#### - 大表加索引

- 1. (影子拷贝)复制旧表表结构,在新表上添加好索引,导出旧表数据,load旧表数据到新表,再继续脚本同步旧表数据,待同步完成以后,rename新表名为旧表(需处理同步过程中旧表更新和删除的数据,可使用触发器,保存更新数据到其他表)

- 2.MySQL5.6在线无锁加索引

ALTER TABLE tbl\_name ADD PRIMARY KEY (column), ALGORITHM=INPLACE,  
LOCK=NONE;

- 3.使用第三方工具:

Percona online schema change

Facebook OSC

#### ■ 配置

#### ■ 调优

##### ■ 索引调优

#### ■ MySQL Binlog格式

■ statement 基于修改的SQL同步,记录的是修改SQL语句(MySQL5.0(含)前唯一的模式)

■ row 基于修改的数据行同步,记录的是每行实际数据的变更,MySQL5.1新增模式

■ mixed statement/row混合模式

#### ■ 事务隔离级别(tx\_isolation)

■ MySQL默认事务隔离级别:REPEATABLE READ (可重复读)

■ MySQL默认事务隔离级别为REPEATABLE READ与MySQL主从同步有关,在MySQL5.0前只有statement binlog同步机制,而在这种机制下,使用read committed会导致主从数据不一致:

如在A,B2个事务中,A事务先开启事务,插入数据,B事务再开启事务,然后A事务中删除该范围内的数据,然后B事务插入在该范围内的数据,B事务先提

交,A事务再提交;此时主库数据正常,从库查询无数据,因为statement先保存的是B事务的修改SQL,然后才保存A事务SQL,导致A在B后执行,删除了B事务创建的数据,所以导致主从数据不一致;

而REPEATABLE READ模式下引入了Gap间隙锁,在修改时,会锁住范围内的数据,能够避免A事务最后执行;

所以使用REPEATABLE READ作为默认隔离级别;

■ 解决主从不同步有2种方式:

- 使用REPEATABLE READ,加入间隙锁
- Binlog使用row行模式,由于MySQL5.1引入的row模式,所以由于历史原因,将REPEATABLE READ作为默认隔离级别
  - READ UNCOMMITTED(读未提交)
    - 一个事务可以读取到另一个尚未提交事务的修改数据,会产生脏读,不使用
    - 可产生脏读,不可重复读,幻读
  - READ COMMITTED(读已提交)
    - 一个事务可以读取到另一个已提交事务的修改数据
    - 当前读,可产生不可重复读,幻读,

因为READ COMMITTED是行锁(RC隔离级别保证对读取到的记录加锁 (记录锁)),不能阻止其他行的插入;

快照读可避免产生不可重复读,幻读.

- REPEATABLE READ(可重复读)
  - 一个事务内,多次执行同一个查询,查询结果相同
  - 当前读,可避免产生不可重复读,幻读,

因为REPEATABLE READ 使用Gap间隙锁,会锁定查询数据范围内的数据,保证其他事务不能操作这些数据;间隙锁不互斥.

- 1.在RR隔离级别下, 存在间隙锁, 导致出现死锁的几率比RC大的多!

2.在RR隔离级别下, 条件列未命中索引会锁表! 而在RC隔离级别下, 只锁行

- SERIALIZABLE(串行读)
  - 查询顺序执行,每个查询都会加锁,并发效率低
  - 从MVCC并发控制退化为基于锁的并发控制。不区别快照读与当前读,所有的读操作均为当前读,读加读锁(S锁),写加写锁(X锁)。

Serializable隔离级别下,读写冲突,因此并发度急剧下降,在MySQL/InnoDB下不建议使用

- //查询设置:

```
select @@tx_isolation;
SELECT @@session.tx_isolation;
SELECT @@global.tx_isolation;
//设置
```

```
set tx_isolation='read-committed';
```

- 较优配置: 隔离级别使用RC,Binlog使用row

■ 事务隔离级别下的现象

■ 脏读(Dirty Read)

■ 一个事务可以读取另一个尚未提交事务的修改数据

■ 不可重复读(nonrepeatable read)

■ 在同一个事务中, 同一个查询在T1时间读取某一行, 在T2时间重新读取这一行时候, 这一行的数据已经发生修改, 可能被更新了 (update) ,



也可能被删除了 (delete) ;与幻读区别: 不可重复读强调数据的update更新操作

- 幻读(phantom read)

- 在同一事务中, 同一查询多次进行时候, 由于其他插入操作 (insert) 的事务提交, 导致每次返回不同的结果集。强调数据的插入操作

- MVCC(Multi-Version Concurrency Control)多版本并发控制

- MySQL在已提交读(READ COMMITTD)和可重复读(REPEATABLE READ)这两种隔离级别下的事务对于SELECT操作会访问版本链中的记录的过程

- 已提交读和可重复读的区别就在于它们生成ReadView的策略不同。

- 已提交读隔离级别下的事务在每次查询的开始都会生成一个独立的ReadView

- 可重复读隔离级别则在第一次读的时候生成一个ReadView, 之后的读都复用之前的ReadView

- 在MVCC并发控制中, 读操作可以分成两类: 快照读 (snapshot read)与当前读 (current read):

快照读, 读取的是记录的可见版本 (有可能是历史版本), 不用加锁;

当前读, 读取的是记录的最新版本, 并且, 当前读返回的记录, 都会加上锁, 保证其他事务不会再并发修改这条记录

- MySQL InnoDB

快照读: 简单的select操作, 属于快照读, 不加锁(可避免不可重复读,幻读):

select \* from where ?

当前读: 特殊的读操作, 插入/更新/删除操作, 属于当前读, 需要加锁:

select \* from table where ? lock in share mode;

select \* from table where ? for update;

insert into table values (...);

update table set ? where ?;

delete from table where ?;

- 数据库读写分离,分库分表

- PostgreSQL

- 基础数据类型

- 用法

- 集群

- 索引

- 配置

- 调优

- PostgreSQL与MySQL选型比较

- DB区别与选型

## LDAP

## GraphDB

- Neo4j

## Spring

---

### Spring基础

- Spring核心概念
  - 架构分析  
(Spring框架是一个分层架构, 包含大约20个模块)
    - Core Container  
Core和Beans模块是 框架的基础部分, 提供IoC(控制反转)和DI(依赖注入)特性, 基础概念是 BeanFactory
    - Core模块(4.3.12.RELEASE)  
主要包含Spring框架基本的核心工具类
      - Core模块下的包org.springframework
        - asm Spring对于ASM的重新打包(带有特定Spring补丁)
        - cglib Spring对于CGLIB 的重新打包(带有特定Spring补丁)  
<https://docs.spring.io/spring-framework/docs/current/java-doc-api/org/springframework/cglib/package-summary.html>
        - core Core模块核心包
          - core.annotation  
注解, 元注解以及具有属性覆盖的合并批注的核心支持包  
如@AliasFor @Order等注解及注解操作
          - core.convert  
类型转换系统API
          - core.env  
Spring的环境抽象包括bean定义配置文件和分层属性源支持。  
如:Environment,PropertiesSource,CommandLineArgs等接口和类
          - core.io  
整个框架中使用的资源相关包
            - org.springframework.core.io.Resource  
extends InputStreamSource  
配置文件转为资源接口(配置文件的封装)

- org.springframework.core.io.AbstractResource  
抽象资源类
  - ByteArrayResource  
Byte数组资源类
  - ClassPathResource  
基于Class Path实现的资源类  
不支持jar中文件
  - FileSystemResource  
基于java.io.File文件系统资源类
  - PathResource  
基于java.nio.file.Path的资源类
  - InputStreamResource  
基于输入流的资源类
  - DescriptiveResource  
简单实现保存资源的描述,不是一个真正可读的资源
  - VfsResource  
基于jboss Virtual File System的资源类
  - AbstractFileResolvingResource  
将URLs 转换为File 引用的抽象基类资源类  
如: UrlResource,ClassPathResource
  - org.springframework.beans.factory.support.BeanDefinitionResource  
org.springframework.beans.factory.config.BeanDefinition 的包装类
- org.springframework.core.io.WritableResource  
扩展Resource接口以支持写入
- org.springframework.core.io.support.EncodedResource extends InputStreamSource  
绑定资源描述符和指定编码或为读取的资源设置编码
- org.springframework.core.io.ResourceLoader  
资源加载策略接口
  - DefaultResourceLoader  
默认资源加载类
  - org.springframework.core.io.support.ResourcePatternResolver  
处理一个位置模式(location pattern)到一个资源对象  
如:模式为classpath\*: 的路径
    - org.springframework.context.ApplicationContext

- `core.serializer`  
Spring的序列化程序接口和实现的根包。
  - `core.style`  
支持将样式值设置为字符串，以ToStringCreator作为中心类。
  - `core.task`  
这个包定义了Spring的核心TaskExecutor抽象，并提供SyncTaskExecutor和SimpleAsyncTaskExecutor实现。
  - `core.type`  
类型自省的核心支持包。
  - `MessageSource` 消息处理策略接口,用于处理消息国际化  
`messageSource.getMessage(code,defaultMsg,local)`  
 Spring provides two out-of-the-box implementations for production:  
`ResourceBundleMessageSource`, built on top of the standard `ResourceBundle`  
`ReloadableResourceBundleMessageSource`, being able to reload message definitions without restarting the VM.  
 Spring提供了两种开箱即用的实现，一种是标准实现，一种是运行时可重新加载。  
 默认使用bean名称为messageSource的单例Bean
    - `ResourceBundleMessageSource` 默认实现
    - `ReloadableResourceBundleMessageSource`
  - `lang` 具有语言级语义的常见注解: `Nullable` , `UsesJava7`, `UsesJava8`等
  - `objenesis` Spring对Objenesis 3.0的重新打包 （带有SpringObjenesis入口点）
  - `util` 其他应用程序工具类包
- Beans模块  
所有应用都要用到的，它包含访问配置文件、创建和管理bean以及进行Inversion of Control/Dependency Injection(Ioc/DI)操作相关的所有类
  - Beans模块下的包`org.springframework.beans`
    - `annotation`  
用于Java 5注释的bean样式处理的支持包
    - `factory`  
实现Spring的轻量级控制反转（IoC/DI）容器的核心包
      - 接口
        - (interface)`org.springframework.beans.factory.Aware`  
标记超级接口，用于指示bean有资格通过回调样式方法由Spring容器通知特定框架对象。

用于容器加载通知的特定回调接口,如  
EnvironmentAware,ApplicationContextAware

- (interface)org.springframework.beans.factory.BeanClassLoaderAware  
允许bean知道bean的回调 class loader; 也就是说, 当前bean工厂使用的类加载器来加载bean类。
- BeanFactoryAware  
希望了解其所有权的bean将实现的接口 BeanFactory。
- BeanNameAware  
由想要在bean工厂中知道其bean名称的bean实现的接口。
- org.springframework.beans.factory.BeanFactory  
用于访问Spring bean容器的根接口
- HierarchicalBeanFactory  
由beanFactory实现的子接口, 可以是层次结构的一部分。
  - !!!org.springframework.beans.factory.support.AbstractBeanFactory  
BeanFactory的抽象化接口, 提供ConfigurableBeanFactory的全部能力, 可以用于BeanFactory获取后端资源类提供单例缓存  
Bean实例化核心类  
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory:实现默认bean创建的抽象BeanFactory超类,具有RootBeanDefinition类指定的全部功能
  - Bean实例创建及初始化(初始化过程中触发  
Aware,InitializingBean,BeanPostProcessor回调实现类)流程:  
doGetBean()获取Bean实例
- 转换beanName,将自动生成的bean名称转换为可用的beanName
- 从Singleton缓存中获取Bean实例  
Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>(256)  
获取不到时,从【FactoryBean中获取新对象实例, 通过getObjectForBeanInstance获取后触发postProcessObjectFromFactoryBean】,触发  
BeanPostProcessor.postProcessAfterInitialization()  
3.若缓存不存在时, 先判断是否存在循环引用, 然后判断是否存在parentBeanFactory,若存在,则使用parentBeanFactory.getBean()获取对象实例

若不存在parentBeanFactory,则开始处理当Bean依赖列表,注册依赖Bean(synchronized同步注册),然后实例化依赖Bean

- 判断当前Bean类型(singleton,prototype,其他)分别实例化Bean(单例使用缓存对象,prototype每次创建新对象)
    - a. 单例Bean创建, 使用 AbstractAutowireCapableBeanFactory.createBean创建Bean实例, 期间会触发所有InstantiationAwareBeanPostProcessor extends BeanPostProcessor实现类的postProcessBeforeInstantiation,使得可以对Bean进行自定义修改, 然后若返回bean不为空会继续触发所有InstantiationAwareBeanPostProcessor extends BeanPostProcessor实现类的postProcessAfterInitialization以对Bean进一步操作。
- 若resolveBeforeInstantiation的BeanPostProcessor没有创建Bean实例, 则继续创建Bean;

真正Bean创建:

【AbstractAutowireCapableBeanFactory.doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[] args)真正创建Bean实例,Bean Class默认要求为public, 然后创建BeanWrapper,并创建Bean实例(通过CglibSubclassingInstantiationStrategy.instantiate创建), 创建完成后, 将Bean加入到registeredSingletons缓存中, 用来解决循环引用问题, 然后解析填充bean属性

AbstractAutowireCapableBeanFactory.populateBean(beanName, mbd, instanceWrapper);

填充bean属性时, 会进行autowire自动装配依赖bean,通过AUTOWIRE\_BY\_NAME或AUTOWIRE\_BY\_TYPE,即依赖注入的过程(DI)

AbstractAutowireCapableBeanFactory.autowireByName()

AbstractAutowireCapableBeanFactory.autowireByType()

注入时,设置属性名对应bean,并注册该属性和当前bean的依赖关系,注解@Autowired注入也在该步骤中(AutowiredAnnotationBeanPostProcessor extends InstantiationAwareBeanPostProcessorAdapter);

!!!Beans属性填充后开始初始化

Bean,AbstractAutowireCapableBeanFactory.initializeBean()初始化指定的Bean实例, 执行initMethod , beanPostProcessors 工厂回调,

!!!先执行invokeAwareMethods, 执行

BeanNameAware,BeanClassLoaderAware,BeanFactoryAware三种类型的回调,

!!!然后触发所有BeanPostProcessors.postProcessBeforeInitializations(),然后执行initMethod,即InitializingBean.afterPropertiesSet()

!!!然后触发所有BeanPostProcessors.postProcessAfterInitializations()

!!!最后注册destroyMethod方法,若当前bean 实现了DisposableBean,或实现了java.lang.AutoCloseable或提供了destroyMethod属性(当destroyMethodName为(inferred)时,存在close或shutdown方法),则将该bean注册为DisposableBean

】

最后进行类型转换,在返回bean时,将Bean转换为输入时要求的类型

(【在ApplicationContext中会注册BeanPostProcessor,一般为org.springframework.context.support.AbstractApplicationContext.refresh()中registerBeanPostProcessors()】)

- 创建Bean实例 早于 Aware调用 早于

BeanPostProcessor.postProcessBeforeInitializations() 早于

InitializingBean.afterPropertiesSet() 早于

BeanPostProcessor.postProcessAfterInitialization()

- 加载Bean时如果当前缓存中不存在,则到ParentBeanFactory中查

找加载

- ListableBeanFactory

由BeanFactory实现的扩展接口;

该工厂可以枚举其所有bean实例, 而不是按照名称一一尝试

- !!!XmlBeanFactory extends DefaultListableBeanFactory

(Spring3.1以后建议使用 DefaultListableBeanFactory(Bean实例化+DI)

+XmlBeanDefinitionReader(加载配置文件资源,扫描

Bean),DefaultListableBeanFactory是XmlBeanDefinitionReader的

BeanDefinitionRegistry)

容器的基础,加载xml配置文件构建BeanFactory,并加载初始化Bean

BeanFactory bf = new XmlBeanFactory( new ClassPathResource("spring-  
config.xml"));

!!!!IoC核心流程(Beans扫描加载为BeanDefinition+Bean实例化(创建Bean+初始化  
Bean+DI(依赖注入))):

- XmlBeanFacotry加载流程:

使用XmlBeanDefinitionReader从XML配置文件中加载Bean定义,并返回Bean个数

private final XmlBeanDefinitionReader reader = new

XmlBeanDefinitionReader(this);

this.reader.loadBeanDefinitions(resource);

- 1. D:\m2\repository\org\springframework\spring-  
beans\4.3.12.RELEASE\spring-beans-(4.3.12.RELEASE-  
sources.jar!\org\springframework\beans\factory\xml\XmlBeanDefinitionReader.jav  
a)

XmlBeanDefinitionReader .loadBeanDefinitions(new EncodedResource(resource));

将资源转换为已编码资源

- public int loadBeanDefinitions(EncodedResource encodedResource) throws  
BeanDefinitionStoreException {} 加载Bean  
将已加载Bean保存到一个HashSet中,同时判断是否多次加载过改配置资源  
3.获取ResourceInputStream,并设置编码
- 从指定的XML中真正加载BeanDefinitions  
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)  
throws BeanDefinitionStoreException {}  
doLoadBeanDefinitions中核心方法为doLoadDocument 和 registerBeanDefinitions  
从XML文档中读取Bean(使用JAXP和ResourceLoader解析Bean XML DTD 和Schema),将  
XML文件转换为Document,并注册Bean  
5.注册Bean,使用BeanDefinitionDocumentReader注册Bean  
DefaultBeanDefinitionDocumentReader.registerBeanDefinitions(doc,  
createReaderContext(resource));  
DefaultBeanDefinitionDocumentReader.doRegisterBeanDefinitions(Element root)  
{  
从根解析Document树,  
DefaultBeanDefinitionDocumentReader.parseBeanDefinitions(Element root,  
BeanDefinitionParserDelegate delegate)

解析Document节点分为2部分:

a.解析默认元素, 解析import(import标签),alias(alise标签),bean(bean标签),nestedBeans(beans标签)

parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate)

b.解析自定义元素

解析默认元素:

a. 解析import标签

protected void importBeanDefinitionResource(Element ele) {}

获取import标签的resource属性, 并解析placeholder

然后加载import对应的XML文件

AbstractBeanDefinitionReader.loadBeanDefinitions(String location, Set actualResources)

解析后触发 XmlReaderContext.fireImportProcessed(location, actResArray, extractSource(ele));事件

b. 解析alias标签

DefaultBeanDefinitionDocumentReader.protected void

processAliasRegistration(Element ele){}

注册别名, 并保存到org.springframework.core.AliasRegistry.aliasMap中,并检查是否有别名循环引用(即存在name和alise 对应的别名注册)

解析后触发 XmlReaderContext.fireAliasRegistered(name, alias, extractSource(ele));事件

c. 解析bean标签

BeanDefinitionParserDelegate.parseBeanDefinitionElement(ele)解析Bean,解析id,name属性, 并将name属性解析为alias列表, 按,或;分隔;默认将id作为beanName若id未提供,将名称列表中的第一个名称作为beanName;同时校验beanName,aliases是否唯一, 将所有beanName,aliases 保存到BeanDefinitionParserDelegate.usedNames中然后解析bean标签的各种属性, 若id,name都未提供, 则使用提供的Class名称,若class属性未提供, 则尝试使用父标签名字+"\$child"或FactoryBeanName+"\$created" 作为beanName,反之抛出异常,当为自动生成的beanName时,将若已存在当前名称,则设置beanName+"#" +counter

解析后触发 XmlReaderContext.fireComponentRegistered(new BeanComponentDefinition(bdHolder);事件

- d.解析beans标签

同解析Document根节点,按照根节点解析方法处理

最后通过

org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean()实例化Bean实例

- DefaultListableBeanFactory.getBean()

-

BeanFactory(XmlBeanFatory,XmlBeanDefinitionReader+ListableBeanFactory)Bean加载简单流程

1.XmlBeanDefinitionReader加载配置文件为Resource,并扫描所有Bean

2.ListableBeanFactory.getBean() 来加载及实例/初始化Bean,即IoC+DI,实例化Bean,及Bean属性赋值后触发部分Aware接口,然后触发

BeanPostProcessor.postProcessBeforeInitialications(),然后调用Bean的initMethod方法,然后触发BeanPostProcessor.postProcessAfterInitializations()



最后注册DisposableBean

3.实例化Bean时使用InstantiationStrategy策略接口,默认使用

CglibSubclassingInstantiationStrategy生成类实例(普通类及方法注入类,含SimpleInstantiationStrategy方法,实例化Bean时,若Bean没有override的方法,则使用反射创建Bean实例,反之使用CGLIB创建代理类)

4.Bean实例化有 2种方式,1种使用FactoryBean.getObject()创建,另一种使用InstantiationStrategy.instantiate()创建

- FactoryBean

由内部使用的对象实现的接口, 这些对象BeanFactory本身就是单个对象的工厂,是一个能生产或修饰对象生成的工厂Bean。如果Bean实现此接口, 则他将用作对象公开的工厂, 而不是直接用作将自生公开的Bean实例。

!!!注意:实现此接口的bean不能作为普通bean。FactoryBean已Bean样式定义, 但是为bean引用(getObject())公开的对象始终是它创建的对象, 可以通过&factoryBeanName获取其本身对象

- FactoryBean是一个工厂方法Bean,用于生成Bean实例,可以让我们自定义Bean创建过程;

FactoryBean提供getObject(),getObjectType(),isSingleton()方法  
使用方法:

- 1.创建自定义FactoryBean并implements FactoryBean,实现getObject(),getObjectType(),isSingleton()方法,并将该FactoryBean注册为Bean
- 2.使用时,自定义FactoryBean代理的对象无需注册为Bean,在创建Bean doCreateBean()时,会自动扫描到匹配到的FacotryBean.getObject()创建Bean实例

- 通过BeanFactory.getBean(Class requiredType)时,最终会调用到AbstractBeanFactory.doCreateBean(),会通过requiredType匹配所有的已知Bean,当匹配的Bean为FactoryBean类型时,会检查FactoryBean.getObjectType()是否匹配(DefaultListableBeanFactory.doGetBeanNamesForType,AbstractBeanFactory.isTypeMatch())requiredType,若匹配,则使用该FactoryBean.getObject()创建Bean对象

- @Component

```
public class AutowireBeanFactoryBean implements FactoryBean {  
    @Override  
    public Object getObject() throws Exception {  
        return new AutowireBean();  
    }  
    @Override  
    public Class<?> getObjectType() {  
        return AutowireBean.class;  
    }  
    @Override  
    public boolean isSingleton() {  
        return true;  
    }  
}
```

//直接注入Bean, Spring doCreateBean时会扫描autowireBean的FactoryBean

@Autowired

AutowireBean autowireBean;

- FactoryBean是一个能生产或修饰对象生成的工厂Bean。一个Bean如

果实现了FactoryBean接口，那么根据该Bean的名称获取到的实际上是getObject()返回的对象，而不是这个Bean自身实例，如果要获取这个Bean自身实例，那么需要在名称前面加上'&'符号。

一般情况下，Spring通过反射机制利用的class属性指定实现类实例化Bean，在某些情况下，实例化Bean过程比较复杂，如果按照传统的方式，则需要在中提供大量的配置信息。配置方式的灵活性是受限的，这时采用编码的方式可能会得到一个简单的方案。Spring为此提供了一个org.springframework.bean.factory.FactoryBean的工厂类接口，用户可以通过实现该接口定制实例化Bean的逻辑。FactoryBean接口对于Spring框架来说占用重要的地位，Spring自身就提供了70多个FactoryBean的实现。它们隐藏了实例化一些复杂Bean的细节，给上层应用带来了便利。从Spring3.0开始，FactoryBean开始支持泛型，即接口声明改为FactoryBean的形式。

beanFactory.getBean("consumeFactoryBean") 返回getObject()创建的对象

beanFactory.getBean("&consumeFactoryBean") 返回consumeFactoryBean本身对象

- org.springframework.beans.factory.config.AbstractFactoryBean

创建单例或原型对象的简单模版超类

- BeanFactory与FactoryBean的区别

- 1.1 BeanFactory 是Spring 访问容器的根入口,为IoC的核心处理类 主要

要有

XmlBeanFactory,DefaultListableBeanFactory,ConfigurableBeanFactory,AbstractAutowireCapableBeanFactory等实现类

1.2 FactoryBean 是内部使用对象的实现接口,本身是一个单个对象工厂。如果Bean实现这个接口,则这个Bean就作为一个公开的对象工厂,该Bean不能作为普通Bean,Bean引用(getObject())始终都是他创建的对象;我们可以自定义FactoryBean来控制Bean对象的创建过程,需要使用自定义FactoryBean创建的对象,不需要声明为Bean容器,在初始化实例时, Spring会扫描bean(BeanDefinitionName)列表,找到该Bean对应的

FactoryBean.getObject()获取对象实例

- 2.1. BeanFactory是IoC容器的底层接口,为Bean容器访问的根入口,是

ApplicationContext的顶级接口,是一个Bean工厂类,负责扫描生产和管理Bean的一个工厂类。

2.2. FactoryBean是Spring提供的工厂Bean接口,在IoC容器的基础上给Bean的实现添加了简单工厂模式和装饰模式,生产的对象由getObject()方法决定

- InitializingBean

被BeanFactory设置所有属性后需要作出一次反应的Bean接口

提供了afterPropertiesSet()方法,用于Bean属性设置完成后做的额外的操作,即初始化完成后做的操作

- DisposableBean

要在销毁时释放资源的bean所实现的接口

提供destroy方法, 支持在销毁Bean时, 调用destroy-method,即销毁时做的操作

- NamedBean

BeanNameAware 对应的获取Bean名称的接口,

只提供一个getBeanName方法

- xml包

org.springframework.beans.factory.xml

- spring-beans-.xsd

spring-tool-.xsd

spring-util-\*.xsd

定义文件

- XmlBeanFactory

Xml加载Bean的核心类

- XmlBeanDefinitionReader

Xml加载Bean对象真正的实现类

- DocumentLoader

XML读取策略接口

- DefaultDocumentLoader

Spring默认Document加载类

使用标准JAXP XML解析器(JDK提供)加载

- config包

- BeanPostProcessor 工厂hook允许对新的实例进行自定义修改,并提供2

个回调方法:

- postProcessBeforeInitialization

在任何bean初始化回调（如InitializingBean afterPropertiesSet 或自定义init-method）之前，将此BeanPostProcessor应用于给定的新bean实例。

- postProcessAfterInitialization

在任何bean初始化回调（如InitializingBean afterPropertiesSet 或自定义init-method）之后，将此BeanPostProcessor应用于给定的新bean实例。

- annotation 注解相关类实现包

如@Autowired,@Configurable及注解注入处理类等

- support

- RootBeanDefinition extends AbstractBeanDefinition

根bean定义表示合并的bean定义，该定义在运行时支持Spring BeanFactory中的特定bean。它可能是由多个相互继承的原始bean定义创建的，通常定义为

GenericBeanDefinitions。根bean定义本质上是运行时的“统一” bean定义视图。

- AbstractBeanDefinitionReader 实现BeanDefinitionReader的抽象接口

- InstantiationStrategy

负责创建与rootBeanDefinition相对于的实例策略接口，

包括使用CGLIB动态创建子类用于方法注入

- SimpleInstantiationStrategy

BeanFactory简单对象实例化策略,不支持方法注入,但提供了对子类重写方法注入的支持

- CglibSubclassingInstantiationStrategy extends

SimpleInstantiationStrategy

BeanFactory的默认对象实例化策略类(如果方法需要由容器重写以实现方法注入，则使用CGLIB动态生成子类)

- propertyeditors

属性编辑器用于将String值转换为对象类型，例如java.util.Properties。

- support

支持org.springframework.beans包的类，例如用于排序和保存bean列表的实用程序类。

- spring-beans核心类

DefaultListableBeanFactory

XmlBeanDefinitionReader

- DefaultListableBeanFactory

DefaultListableBeanFactory是整个bean加载的核心部分，是Spring注册及加载bean的

默认实现

- XmlBeanDefinitionReader

XmlBeanFactory与DefaultListableBeanFactory不同的地方其实是在XmlBeanFactory中使用了自定义的XML读取器XmlBeanDefinitionReader，实现了个性化的BeanDefinitionReader读取，DefaultListableBeanFactory继承了AbstractAutowireCapableBeanFactory并实现了ConfigurableListableBeanFactory以及BeanDefinitionRegistry接口

- ResourceLoader：定义资源加载器，主要应用于根据给定的资源文件地址返回对应的Resource

- BeanDefinitionReader：主要定义资源文件读取并转换为BeanDefinition的各个功能

- EnvironmentCapable：定义获取Environment方法

- DocumentLoader：定义从资源文件加载到转换为Document的功能

- AbstractBeanDefinitionReader：对EnvironmentCapable、BeanDefinitionReader类定义的功能进行实现

- BeanDefinitionDocumentReader：定义读取Document并注册BeanDefinition功能

- BeanDefinitionParserDelegate：定义解析Element的各种方法

- Spring核心加载类之间的类关系

- Resource,ResourceLoader 资源加载

- org.springframework.core.io.Resource 加载的配置资源接口：

子类有:ClassPathResource,AbstractResource,FileSystemResource,UrlResource等

- org.springframework.core.io.ResourceLoader 资源加载接口：

默认实现为:DefaultResourceLoader,自动检测路径进行Resource加载

- BeanDefinition扫描加载

- org.springframework.beans.factory.config.BeanDefinition Bean定义接口

□

默认实现为AbstractBeanDefinition抽象类，

常用类为RootBeanDefinition extends AbstractBeanDefinition

- org.springframework.beans.factory.support.BeanDefinitionReader

BeanDefinition加载的接口

默认实现为:AbstractBeanDefinitionReader

常用类有:XmlBeanDefinitionReader extends AbstractBeanDefinitionReader

- BeanFactory Bean创建管理流程

- org.springframework.beans.factory.BeanFactory Bean容器加载的入口, Spring IoC实现的核心接口

□, Spring IoC实现的核心接口

抽象实现为AbstractBeanFactory

主要实现类:DefaultListableBeanFactory extends

AbstractAutowireCapableBeanFactory extends AbstractBeanFactory implements

ConfigurableBeanFactory extends

BeanFactory(AbstractAutowireCapableBeanFactory.doCreateBean为创建Bean实例的主方法)

- 

org.springframework.beans.factory.support.DefaultSingletonBeanRegistry 共享Bean实例的基本注册表,实现了

org.springframework.beans.factory.config.SingletonBeanRegistry,允许缓存单例

Bean,通过名称获取bean实例

- FactoryBean 自定义实例化Bean
- `org.springframework.beans.factory.FactoryBean`

实现自定义初始化Bean的工厂方法接口

- ApplicationContext加载Spring
- `org.springframework.context.ApplicationContext`

为应用程序提供配置的中央接口,

主要实现类有:AbstractApplicationContext,ApplicationContext实现的核心抽象方法  
(refresh是加载Spring的主方法)

XmlWebApplicationContext extends

AbstractRefreshableWebApplicationContext extends

AbstractApplicationContext 通过web.xml()加载SpringXml的实现类

- InitializingBean
- `org.springframework.beans.factory.InitializingBean` 实现Bean加载后执行afterPropertiesSet回调的接口
- DisposableBean
- `org.springframework.beans.factory.DisposableBean`

实现Bean销毁时执行destroy-method回调的接口

- BeanPostProcessor
- `org.springframework.beans.factory.config.BeanPostProcessor` 实现Bean初始化前后的回调接口

- Spring Bean加载循环依赖问题

- 循环依赖即循环引用,是2个或2个以上Bean相互持有对方,最终形成闭环,如A依赖于B,B依赖于C,C依赖于A

Spring中循环依赖场景有:

- 1.构造器循环依赖
- 2.field属性循环依赖

Singleton方式支持循环依赖

Prototype不支持循环依赖

- 检查循环依赖:

Bean创建时,将Bean标记为正在创建中,Spring中使用singletonsCurrentlyInCreation  
ConcurrentHashMap保存标记

- 解决循环依赖: (只能解决属性循环依赖,不能解决构造方法循环依赖;对于缓存的操作都使用了singletonObject作为synchronized(singleton)对象)

Spring解决循环依赖使用了三级缓存,分别为

singletonObject: 单例对象Cache,一级缓存(ConcurrentHashMap)

earlySingletonObjects:提前曝光的单例对象Cache,二级缓存(HashMap)

singletonFactories:单例对象工厂的Cache,三级缓存(HashMap)

- Bean缓存获取流程

(`org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(beanName,allowEarlyReference)`)

- 获取Bean时,首先从一级缓存singletonObject中获取Bean
- 如果1中获取不到,并且对象正在创建中,则从二级缓存earlySingletonObject中获取

- 如果2中获取不到且允许singletonFactories通过getObject()获取,则从三级缓存singletonFactory.getObject()中获取
  - Cache流程:
- Bean加载时,首先完成了实例创建,然后将自己曝光到singletonFactories中(org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean())
- getSingleton时若singletonFactories.get(beanName)可以获取到Bean时,将Bean从singletonFactories转移到二级缓存earlySingletonObjects中(org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(beanName,allowEarlyReference),getBean时先getSingleton,获取不到时再doCreateBean,doCreateBean时做1和3步骤的操作)
- Bean完全加载完成之后,从二级缓存earlySingletonObjects移动到一级缓存singletonObjects中(org.springframework.beans.support.DefaultSingletonBeanRegistry.getSingleton(beanName,singletonFactory)后addSingleton(beanName,singletonObject))
  - 使用3级缓存:

若仅解决循环依赖问题,则2级缓存也可以实现,  
添加3级缓存是给用户提供了接口扩展(SmartInstantiationAwareBeanPostProcessor)

  - 3级缓存解决的循环依赖是基于单例类的Field字段级别的(setter)注入,构造方法注入循环依赖依然会有问题,且暂时无法解决
  - SpringBean延迟加载
    - <https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-dependency-resolution>
    - SpringBean延迟加载是指Spring扫描Bean以后不立即加载,而是在使用的时候采取加载Bean
  - Context模块

构建于Core和Beans模块基础之上,提供了一种类似于JNDI注册器的框架式的对象访问方法。Context模块继承了Beans的特性,为Spring核心提供了大量扩展,添加了对国际化(如资源绑定)、事件传播、资源加载和对Context的透明创建的支持。  
ApplicationContext接口是Context模块的关键  
org.springframework.context

  - org.springframework.context 构建在Core和Beans模块基础之上
  - annotation 对应用程序上下文的注解支出

包括公共注释, 组件扫描和用于创建Spring管理对象的基于java的元数据  
如:@Bean @ComponentScan等

  - config 用于高级应用程序上下文配置的支持包, 其中XML模式是主要配置格式。
  - event 应用程序事件的支持类, 例如标准上下文事件
  - expression Spring应用程序上下文中的表达式(SpEL)解析支持。
  - i18n 国际化支持
  - support 支持org.springframework.context包的类, 例如ApplicationContext实现和MessageSource实现的抽象基类。
  - weaving 在Spring的LoadTimeWeaver抽象基础上, 对Spring应用程序上下文的加载时编织支持。用于AspectJ
  - ApplicationContext

Spring应用关键接口

及启动初始化入口

ApplicationContext是BeanFactory的扩展

- org.springframework.context.support.AbstractApplicationContext

implements ConfigurableApplicationContext

org.springframework.context.ApplicationContext接口的抽象实现, 简单地说实现公共上下文功能。使用模板方法设计模式, 需要具体的子类来实现抽象方法

AbstractRefreshableConfigApplicationContext

添加对指定配置位置的处理,基于XML的ApplicationContext实现,

如:ClassPathXmlApplicationContext,FileSystemXmlApplicationContext,XmlWebApplicationContext等

- refresh()方法:

从Xml,properties或数据库模式中加载或刷新配置的持久表示;由于这是一个启动方法,如果启动失败应该销毁全部已创建的单例,以免浪费资源,调用方法后要么全部实例化,要么全部不实例化;

流程:

- 1.prepareRefresh,为刷新context做准备,设置启动时间和激活标记, 以及执行配置资源的初始化(初始化占位符资源)
- 2.obtainInFreshBeanFactory,刷新内部BeanFactory,清空已有BeanFactory,创建新的BeanFactory(DefaultListableBeanFactory),并扫描Bean容器,并loadBeanDefinitions()
- 3.prepareBeanFactory()为使用BeanFactory做准备
- 4.postProcessBeanFactory()post BeanFactory处理
- 5.invokeBeanFactoryPostProcessors()实例化并调用所有注册的BeanFactoryPostProcessor bean,如果给定显示顺序,则按顺序调用,必须在单例实例化之前调用
- !!!registerBeanPostProcessors注册所有BeanPostProcessorbean,如果给定显示顺序,则按顺序调用,必须在单例实例化之前调用(用于Bean初始化时的BeanFactoryPostProcessor回调),会按照PriorityOrdered->Ordered->nonOrdered顺序将BeanPostProcessor加入到列表中
- 7.initMessageSource()初始化messageSource,用于Message国际化的支持
- initApplicationEventMulticaster()为上下文初始化事件多宿主
- 9.onRefresh() 初始化其他指定的bean
- 10.注册ApplicationListener Bean
- 11.finishBeanFactoryInitialization()完成BeanFactory的初始化,并初始化所有剩余的单例SingletonBean(非延迟加载)
- !!!
- 【org.springframework.beans.factory.support.DefaultListableBeanFactory.preInstantiateSingletons()预初始化单例Beans,加载的是Service,Repository层Bean:首先获取到所有的BeanDefinitionNames,然后循环创建及初始化Bean,即调用org.springframework.beans.factory.AbstractBeanFactory.getBean(name)创建及初始化Bean
- 】
- 12.finishRefresh()完成refresh方法
- refresh后Bean加载时机: Bean加载是在refresh方法中加载的,具体在PostBeanFactory时的preInstantiateSingletons()循环加载初始化所有非延迟加载的SingletonBean

- Spring(含Bean)加载流程: Spring加载时,通过调用  
org.springframework.context.support.AbstractApplicationContext.refresh()方法开始  
初始化BeanFactory,扫描/加载BeanDefinitions,注册BeanFactoryPostProcessor,注册  
BeanPostProcessors,然后  
org.springframework.beans.factory.DefaultListableBeanFactory.preInstantiateSingletons()  
循环初始化所有非单例的Bean;在扫描加载BeanDefinitions时,使用  
org.springframework.beans.factory.support.AbstractBeanDefinitionReader.loadBeanDefinitions()  
加载BeanDefinitions;  
初始化Bean时使用  
org.springframework.beans.factory.support.AbstractBeanFactory.doCreateBean()  
创建及初始化bean;同时实例化Bean有2种情况,一种为默认的Bean初始化,即  
org.springframework.beans.factory.support.InstantiationStrategy()(含  
SimpleInstantiationStrategy和CglibSubclassingInstantiationStrategy),另一种为实现  
自定义org.springframework.beans.factory.FactoryBean,实现其中  
isSingleton(),getObjectType(),getObject()方法,当加载Bean时,会扫描所有  
BeanDefinitions,若扫描到的Bean类型为FactoryBean时,通过getObjectType来判断与  
目标Bean是否为相同的对象,若是,则使用该FactoryBean.getObject()创建Bean实例。

- !!!Spring基于web.xml,XML配置文件的启动是通过  
ContextLoaderListener中调用XmlWebApplicationContext.refresh()启动Spring加载  
(实质为调用AbstractApplicationContext.refresh())  
!!!SpringBoot基于SpringApplication.run的启动是通过  
SpringApplication.refreshContext(),最终调用AbstractApplicationContext.refresh()启动Spring加载

二者的核心都是使用AbstractApplicationContext.refresh()加载Spring

- SpringWeb中AbstractApplicationContext.refresh() 加载BeanFactory,  
扫描Beans加载BeanDefinitions,注册BeanPostProcessor.  
AbstractRefreshableConfigApplication来启动Spring,并初始化非延迟加载的所有  
Singleton单例Bean

SpringMVC中org.springframework.web.servlet.DispatcherServlet

- org.springframework.web.servlet.DispatcherServlet加载  
- ClassPathXmlApplicationContext  
- FileSystemXmlApplicationContext  
- GenericXmlApplicationContext  
- org.springframework.web.context.ContextLoaderListener extends  
ContextLoader implements ServletContextListener  
SpringMVC启动/关闭Spring Root WebApplicationContext 监听Listener;  
主要方法为contextInitialized()-

>org.springframework.web.context.ContextLoader.initWebApplicationContext(servletContext)

默认WebApplicationContext为

org.springframework.web.context.ContextLoader.properties配置的  
org.springframework.web.context.support.XmlWebApplicationContext

- org.springframework.web.context.ContextLoader  
执行根应用程序上下文(root application context)的初始化工作,被  
ContextLoaderListener调用  
initWebApplicationContext()方法初始化ApplicationContext



初始化流程:

- 1.校验web.xml中是否配置了多个ContextLoader,若配置多个则抛出异常
- 2.校验context(ApplicationContext)是否为null,为null则创建WebApplicationContext
- 3.若context instanceof ConfigurableWebApplicationContext()则执行

【configureAndRefreshWebApplicationContext()】方法,此步骤为核心加载方法;  
首先设置WebApplicationId,然后加载ParentContext,然后加载设置  
contextConfigLocation参数配置, 自定义Context,最后执行主要的Spring加载方法  
wac.refresh

!!!wac.refresh()-

>org.springframework.context.support.AbstractApplicationContext.refresh()

- 4.设置applicationContext加载标记

- org.springframework.cache Spring的通用缓存抽象。
- org.springframework.ejb
- org.springframework.format
- org.springframework.jmx 包含Spring的JMX支持, 其中包括将Spring托管的bean注册为JMX MBean以及对远程JMX MBean的访问。
- org.springframework.jndi 提供了对JNDI访问的类, 简化了对存在JNDI中的配置访问
- org.springframework.remoting Spring远程处理基础结构的异常层次结构, 独立于任何特定的远程方法调用系统。
- org.springframework.scheduling Spring调度模块
- org.springframework.scripting Spring脚本支持的核心接口。
- org.springframework.stereotype表示类型或方法在整个体系结构中的作用的注释 (在概念级别, 而不是在实现级别)
- org.springframework.ui 对通用UI层概念的支持
- org.springframework.validation 提供数据绑定和验证功能, 以用于业务和/或UI层。

- Expression Language模块

提供了一个强大的表达式语言用于在运行时查询和操纵对象, 该语言支持设置/获取属性的值, 属性的分配, 方法的调用, 访问数组上下文、容器和索引器、逻辑和算术运算符、命名变量以及从Spring的IoC容器中根据名称检索对象

- AOP

- AOP模块提供了一个符合AOP联盟标准的面向切面编程的实现, 它让你可以定义例如方法拦截器和切点, 从而将逻辑代码分开, 降低它们之间的耦合性, 利用source-level的元数据功能, 还可以将各种行为信息合并到你的代码中

Spring AOP模块为基于Spring的应用程序中的对象提供了事务管理服务, 通过使用Spring AOP, 不用依赖EJB组件, 就可以将声明性事务管理集成到应用程序中

- Data Access/Integration

- JDBC模块

JDBC模块提供了一个JDBC抽象层, 它可以消除冗长的JDBC编码和解析数据库厂商特有的错误代码, 这个模块包含了Spring对JDBC数据访问进行封装的所有类

- ORM模块

为流行的对象-关系映射API, 如JPA、JDO、Hibernate、iBatis等, 提供了一个交互层, 利用ORM封装包, 可以混合使用所有Spring提供的特性进行O/R映射, 如前边提到的简单声明性事务管理

- OXM模块

提供了一个Object/XML映射实现的抽象层，Object/XML映射实现抽象层包括JAXB，Castor，XMLBeans，JiBX和XStream

- JMS模块

(Java Message Service) 模块主要包含了一些制造和消费消息的特性

- Transactions模块

支持编程和声明式事物管理，这些事务类必须实现特定的接口，并且对所有POJO都适用

- Web

Web上下文模块建立在应用程序上下文模块之上，为基于Web的应用程序提供了上下文，所以Spring框架支持与Jakarta Struts的集成。Web模块还简化了处理多部分请求以及将请求参数绑定到域对象的工作。Web层包含了Web、Web-Servlet、Web-Struts和Web、Portlet模块

- Web模块

提供了基础的面向Web的集成特性，例如，多文件上传、使用Servlet listeners初始化IoC容器以及一个面向Web的应用上下文，它还包含了Spring远程支持中Web的相关部分

- Web-Servlet模块

web.servlet.jar：该模块包含Spring的model-view-controller(MVC)实现，Spring的MVC框架使得模型范围内的代码和web forms之间能够清楚地分离开来，并与Spring框架的其他特性基础在一起

- Web-Struts模块

该模块提供了对Struts的支持，使得类在Spring应用中能够与一个典型的Struts Web层集成在一起

- Web-Portlet模块

提供了用于Portlet环境和Web-Servlet模块的MVC的实现

- Test

- Test模块支持使用JUnit和TestNG对Spring组件进行测试

- 事件机制

- ApplicationEvent: 事件抽象类

- ApplicationListener 事件监听器接口

- 定义通用方法onApplicationEvent

- ApplicationEventMulticaster 事件广播器接口

- 用于事件监听器的注册和事件的广播

- ApplicationEventPublisher 事件发布者

- SpringIOC(源码)

- IOC容器初始化流程

- 加载解析XML流程

- 解析BeanDefinition流程

- 创建BeanDefinition流程

- PropertyValue属性解析流程

- 注册BeanDefinition流程

- Bean创建流程
  - org.springframework.beans.factory.support.AbstractBeanFactory#createBean
  - org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBean
  - org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#doCreateBean
  - Bean实例化流程
    - 构造器反射创建对象/CGLIB创建代理对象
    - FactoryBean创建对象
    - AOP产生动态代理对象流程
  - 属性填充流程(普通属性和依赖Bean)
    - StringValueType属性处理流程
      - 类型转换
    - RuntimeBeanReference属性处理流程
    - DI,依赖注入,注入依赖的Bean
  - Bean初始化流程
    - org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#initializeBean(final String beanName, final Object bean, RootBeanDefinition mbd)
    - 执行BeanNameAware回调
      - (BeanNameAware) bean).setBeanName(beanName)
    - BeanPostProcessor执行流程
      - 1.触发BeanPostProcessor.postProcessBeforeInitialization
      - 2.执行对象的initMethod方法(即执行InitializingBean.afterPropertiesSet)
      - 3.执行自定义initMethod
      - 4.触发BeanPostProcessor.postProcessAfterInitialization
- IOC模块BeanFactory
  - 解析BeanFactory继承体系
  - 解析BeanDefinition继承体系
  - 解析ApplicationContext继承体系
- ApplicationContext
  - 循环依赖问题
- SpringAOP
  - SpringAOP核心概念:SpringAOP是基于Java动态代理和CGLIB动态代理实现的方法级的AOP
    - Aspect: 是包含Pointcut和Advice的集合
    - Pointcut: 声明JoinPoint列表的切点
    - JoinPoint:连接点,具体的被切面的构造方法,属性,或方法

- Advice: 切入时,可以执行的操作;有Before(进入JoinPoint之前触发),AfterReturning(返回之后触发),AfterThrowing(异常之后触发),After(不管成功失败最后触发),Around(环绕通知,通过在process.proceed()前后添加处理逻辑) 5种操作

触发顺序Before->目标方法->After->AfterReturning或AfterAround

- @EnableAspectJAutoProxy(proxyTargetClass:false,exposeProxy:false)

声明自动处理被@Aspect注解的类

proxyTargetClass:是否强制使用CGLIB代理

exposeProxy:是否开启增强代理,用于目标对象内调用发生时,提供

AopContext.currentProxy()获取当前Bean

- @Aspect 定义一个类为AOP类

@Pointcut(value,argNames) 定义切入点,指定JoinPoint

value为切入点表达式:有2种形式,1为切入点表达式,如execution,target表达式等,第二种为声明@Pointcut注解的方法名(含括号)

- //情况一

```
@Before("execution(*
com.zejian.spring.springAop.dao.UserDao.addUser(..)")
public void before(){
System.out.println("前置通知....");
}
```

//2情况2

```
@Before("before()")
public void before2(){
System.out.println("前置通知....");
}
```

- @Before(value,argNames) 前置通知

JoinPoint, 是Spring提供的静态变量, 通过joinPoint 参数, 可以获取目标对象的信息,如类名称,方法参数,方法名称等,该参数是可选的。

- /\*\*

- 前置通知

- @param joinPoint 该参数可以获取目标对象的信息,如类名称,方法参数,方法名称等

```
/
@Before("execution( org.kangspace.UDao.add(..)")
public void before(JoinPoint joinPoint){
System.out.println("我是前置通知");
}
```

- @After(value,argNames) 后置通知

- /\*\*

- 后置通知, 不需要参数时可以不提供

```
/
@After(value="execution( com.kangspace.*(..)")
public void AfterReturning(JoinPoint joinPoint){
System.out.println("我是后置通知...");
}
```

- @AfterReturning(value,pointcut,returning,argsName)后置通知  
pointcut和value意义相同,pointcut会覆盖value值  
returning指定返回值参数名称,可在方法中获通过该值获取返回值,若无返回值,则为null  
- /\*\*

- 后置通知

- returnVal,切点方法执行后的返回值

```
/
@AfterReturning(value="execution( org.kangspace.*(..))",returning = "returnVal")
public void AfterReturning(JoinPoint joinPoint,Object returnVal){
    System.out.println("我是后置通知...returnVal"+returnVal);
}
```

- @AfterThrowing(value,pointcut,throwing,argnames) 后置异常通知  
throwing 指定异常参数名称,可在方法中获通过该值获取异常对象  
- /\*\*

- 抛出通知

- @param e 抛出异常的信息

```
/
@AfterThrowing(value="execution( org.kangspace.*(..))",throwing = "e")
public void afterThrowable(Throwable e){
    System.out.println("出现异常:msg="+e.getMessage());
}
```

- @Around(value,argNames)环绕通知

第一个参数必须是ProceedingJoinPoint, 通过该对象的proceed() 方法来执行目标函数, proceed()的返回值就是环绕通知的返回值。同样的, ProceedingJoinPoint对象也是可以获取目标对象的信息,如类名称,方法参数,方法名称等等。

```
- @Around("execution( org.kangspace.*(..))")
public Object around(ProceedingJoinPoint joinPoint) throws Throwable {
    System.out.println("我是环绕通知前....");
    //执行目标函数
    Object obj= (Object) joinPoint.proceed();
    System.out.println("我是环绕通知后....");
    return obj;
}
```

- 通知传递参数:

(org.springframework.aop.aspectj.AbstractAspectJAdvice#Object[] argBinding)  
SpringAOP中可以将匹配的方法相应的参数或对象自动传递给通知放方法。  
获取到匹配的方法参数后通过args指示符(参数名需和切入点参数名一致)和  
"argNames"属性指定参数名,argNames可省略,存在时必须与args指示器名称保持一致,  
多个参数按,分割。

- //带argNames参数

```
@Before(value="args(param)", argNames="param") //明确指定了
public void before(int param) {
    System.out.println("param:" + param);
}
```

- //不带argNmaes参数

```
@Before("execution(public * org.kangspace...addUser(..) && args(userId,..))")
```

```
public void before(int userId) {
    //调用addUser的方法时如果与addUser的参数匹配则会传递进来会传递进来
    System.out.println("userId:" + userId);
}
```

- Aspect 优先级: 如果多个通知需要在同一切点函数的过滤目标方法上执行, 那些在目标方法前执行("进入")的通知函数, 最高优先级的通知将会先执行, 在执行在目标方法后执行("退出")的通知函数, 最高优先级会最后执行。对于在同一个切面定义的通知函数将会根据在类中的声明顺序执行, 对于在不同类中定义的通知函数将根据Aspect定义的Order顺序执行@Order 或实现Ordered接口并实现getOrder()方法指定。

- Weaving: 织入, 是将切面与外部类链接起来以创建通知对象(advised object)的过程

- 切入点表达式(指示符)

作用在@Before,@After等注解中的value,pointcut参数值

(org.springframework.aop.aspectj.AspectJAdviceParameterNameDiscoverer)

- 运算符 and or not , && || !

- //匹配了任意实现了UDao接口的目标对象的方法并且该方法名称为add

```
@Pointcut("target(org.kangspace.UDao)&&execution( org.kangspace.UDao.add(..))")
private void pointcut(){}
- 通配符:
```

.. : 匹配方法定义中的任意数量的参数,或匹配类型定义中的任意数量的包

■ : 匹配给定类的子类

■ : 匹配任意数量的字符

- //任意返回值, 任意名称, 任意参数的公共方法

```
execution(public * (..))
```

//匹配实现了UDao接口的所有子类的方法

```
within(org.kangspace.UDao+)
```

//匹配org.kangspace包及其子包中所有类的所有方法

```
within(org.kangspace..)
```

- execution 方法签名表达式,可指定返回值类型,包,对象,参数类型等

//scope : 方法作用域, 如public,private,protect

//return-type: 方法返回值类型

//fully-qualified-class-name: 方法所在类的完全限定名称

//parameters 方法参数

```
execution( .*(parameters))
```

- //匹配UDaoImpl类中第一个参数为int类型的所有公共的方法

```
@Pointcut("execution(public * org.kangspace.UDaoImpl.(int , ..))")
```

- within 类型签名表达式,方便类型(如接口,类,包名)过滤

//type name 包名或类名

```
within()
```

- //匹配org.kangspace包及其子包中所有类中的所有方法

```
@Pointcut("within(org.kangspace..)")
```

- target : 用于匹配当前目标对象类型的执行方法

//target name: 目标类名

```
target()
```

- //匹配了任意实现了UDao接口的目标对象的方法进行过滤

```

@Pointcut("target(org.kangspace.UDao)")
private void pointcut(){}
    - bean:SpringAOP扩展,用于匹配特定名称的Bean对象的执行方法
bean()
    - //匹配名称中带有后缀Service的Bean
@Pointcut("bean(*Service)")
private void myPointcut1(){}
    - this:用于匹配当前AOP代理对象类型的执行方法,AOP代理类本身
//class name
this()
    - //匹配了任意实现了UDao接口的代理对象的方法进行过滤
@Pointcut("this(org.kangspace.UDao)")
private void myPointcut2(){}
    - @within:用于匹配所持有指定注解类型类内的方法(作用于类的注解)
@within()
    - //匹配使用了CustAnnotation注解的类(注意是类)
@Pointcut("@within(org.kangspace.CustAnnotation)")
private void pointcut(){}
    - @annotation:用于匹配所持有指定注解的方法(作用于方法的注解)
@annotation()
    - //匹配使用了MethodAnnotation注解的方法(注意是方法)
@Pointcut("@annotation(org.kangspace.MethodAnnotation)")
private void pointcut(){}
    - args(<paramName,...>):参数指示器,用于配合其他指示器获取传递参数
    - @Before("execution(public * org.kangspace..*.addUser(..)) &&
args(userId,...)")
public void before(int userId) {
//调用addUser的方法时如果与addUser的参数匹配则会传递进来会传递进来
System.out.println("userId:" + userId);
}

```

#### ■ AOP源码

##### ■ SpringAOP 是方法级的AOP,MehodProxy

核心处理类:org.springframework.aop.framework. CglibAopProxy#

DynamicAdvisedInterceptor

org.springframework.aop.framework.JdkDynamicAopProxy

CGLIB使用子类方式创建代理对象,并将父类的所有方法重写为final,

Spring中访问的实际是代理对象,在经过AOP前置增强方法,才会执行到真正target类的方法,再执行后置增强方法.

CGLIB利用cglib.proxy.MethodInteceptor配置

org.springframework.aop.framework.CglibAopProxy#DynamicAdvisedInt  
erceptor触发所有AopAdvice操作

##### ■ CGLIB动态代理类信息:

1.代理对象(使用JDK Proxy或CGLIB动态代理):

class

org.springframework.cloud.zookeeper.ZookeeperProperties\$\$Enha

- 代理类重写父类方法为final,并生成对应的内部方法:

//普通方法类

```
private static final Method CGLIB$setEnabled$0$Method;
```

//方法代理类

```
private static final MethodProxy CGLIB$setEnabled$0$Proxy;
```

//方法

```
final void CGLIB$setBaseSleepTimeMs$3(Integer paramInteger){
    super.setBaseSleepTimeMs(paramInteger);
}
```

//代理方法

```
public final void setBaseSleepTimeMs(Integer paramInteger){}
```

3.代理方法中执行对应拦截点织入的前置增强方法,  
然后执行被代理的方法,再执行织入的后置增强方法

- CGLIB动态代理方法执行流程:

- 访问代理对象方法时(method invoke),首先会触发方法拦截器

DynamicAdvisedInterceptor(org.springframework.aop.framework. CglibAopProxy#  
DynamicAdvisedInterceptor)#intercept(Object proxy(代理对象),Method  
method(target 实际方法),Object[] args(参数),MethodProxy methodProxy(target的方  
法代理类))方法,

2.先判断this.advised.exposeProxy是否为true,则设置

AopContext.setCurrentProxy(proxy);

3.然后获取目标对象class,通过method和目标对象targetClass扫描所有advised和  
Interceptor;

若advised为空同时被代理方法是public,则使用 methodProxy.invoke(target,  
argsToUse);执行;

反之new CglibMethodInvocation(proxy, target, method, args, targetClass, chain,  
methodProxy).proceed();处理

4.org.springframework.aop.frameworkReflectiveMethodInvocation#proceed(),中通  
过递归调用invokeJoinpoint()或interceptor.invoke()处理

invokeJoinpoint()中若方法为public则直接使用方法代理对象执行方法,反之使用反射,先  
将方法设置为public,再执行invoke;

简述: 若访问代理方式时,检测不存在方法AOP切入点及增强方法且该方法是public时,直  
接使用MethodProxy.fastClassInfo.f1.invoke(f1为原始对象,即被代理对象)执行方法,反  
之依次执行已定义的AOP增强方法(AspectJAdvice)

- 核心处理类:

org.springframework.aop.frameworkReflectiveMethodInvocation#proceed();

interceptorOrInterceptionAdvice使用调用链的方式执行: 先执行afterAdvice,after中执  
行around,around中执行before,然后执行被代理的方法,继续around,最后再执行after,若  
存在多个类型的advice,则依次嵌套执行,如after1-after2-around1-around2-before1-  
before2-invoke-around2 end-around1 end-after2 advice-after1 advice

调用链中第一个拦截器是ExposelInvocationInterceptor,

然后按顺序执行after-around-before-invoke-around end-afterAdvice

a.AspectJAroundAdvice中, invoke直接执行aroundAdvice方法,使用

ProceedingJoinPoint pjp.proceed()最终出发MethodProxy.invoke()调用原始对象  
invoke方法



b.AspectJAfterAdvice的invoke方法会继续执行传入invocation的proceed方法,即进行下一个Advice处理,然后在finally中执行afterAdvice的增强方法

c.Before使用MethodBeforeAdviceInterceptor#invoke中,先执行advice.before方法,然后继续执行Advice链的processed(),然后执行原始对象方法,若存在AroundAdvice时,before在AroundAdvice中执行

- 设置CGLIB代理类class生成位置:

```
System.setProperty("cglib.debugLocation", "D:\\class")
```

CGLIB代理类使用

```
org.springframework.cglib.core.DebuggingClassWriter(AbstractClassGenerator)生成,默认不生成.class文件
```

-- 设置输出jdk动态代理生成的class类

```
System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true");
```

Java Proxy动态代理使用sun.misc.ProxyGenerator生成字节码文件

- [aop:aspectj-autoproxy/](#)在SpringXML配置中开启AOP支持  
@EnableAspectJAutoProxy在SpringBoot中开启AOP
- proxy-target-class(true/false): 是否强制使用CGLIB创建对象代理, Spring AOP中使用JDK动态代理和CGLIB动态代理  
expose-proxy:增强代理,在目标对象内部的自我调用将无法实施切面中的增强(org.springframework.aop.framework.CglibAopProxy,),可开启aop:aspectj-autoproxy expose-proxy = "true"/>,并使用  
org.springframework.aop.framework.  
((A)AopContext.currentProxy()).method();或使用  
SpringApplicationContext.getBean(class).method()调用
- AOP注解的方法中,用this调用同类的注解方法时,注解不会生效(因为在A方法中调用B方法时,后进入的是A所在类的代理类,代理类中使用invoke来调用的实际类的方法,而实际方法A中再调用B时,不会走代理类,故不会触发AOP)
- SpringTx(事务)
  - Spring Transaction AOP注解的方法必须是public,抛出异常默认RuntimeException和Error回滚,其他异常默认不回滚
  - Spring Transactional 实现原理:  
主要依赖于AOP,基于  
AOP(org.springframework.transaction.interceptor.TransactionInterceptor(extends TransactionAspectSupport类似AroundAdvice)增强方法处理事务;  
在检测到Transactional时,根据传播属性和隔离级别,获取新的连接Connection,然后设置只读状态,设置事务隔离级别,然后关闭事务自动提交,然后在Update时,使用transaction中的Connection,事务结束后,先开启事务自动提交,然后恢复事务隔离级别,再重置readOnly状态;  
Spring Transactional事务依赖于事务管理器  
器,org.springframework.jdbc.datasource.DataSourceTransactionManager,事务管理器管理数据源,再配合tx-advice设置事务监听方法及事务属性配置,或使用  
@Transactional注解标记事务;  
数据源使用动态数据源管理:  
org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource并实现其中determineCurrentLookupKey方法,返回配置的数据源key
  - Spring @Transactional 执行流程:

- 注册DataSourceTransactionManager,并配置DataSource(单个数据源或多个数据源AbstractRoutingDataSource)
- 2.若配置了多数据源,则在进入@TransactionalAOP方法前加一层Before增强方法用来切换数据源key
- 3.@Transactional是以AOP方式  
(org.springframework.transaction.interceptor.TransactionInterceptor(extends TransactionAspectSupport类似AroundAdvice)#invoke()来处理事务
- 4.invoke()中再调用TransactionAspectSupport。invokeWithinTransaction处理,
- 4.1首先使用AbstractFallbackTransactionAttributeSource#getTransactionAttribute获取TransactionAttribute,  
获取时会将结果进行ConcurrentHashMap缓存方法对应的TransactionAttribute,该步骤中会校验方法是否是public和是否只允许public(默认为true),若为非public,则不处理事务;
- 4.2检查时第一步扫描方法上的@Transactional注解,方法上不存在时,第二步扫描该方法所在类上的注解
- 4.3检测并获取事务管理器  
PlatformTransactionManager(DataSourceTransactionManager)
- 4.4该步骤开始处理事务,TransactionAspectSupport。createTransactionIfNecessary,需要创建事务时,调用PlatformTransactionManager.getTransaction(getTransaction为AbstractPlatformTransactionManager的抽象方法,实际为DataSourceTransactionManager.getTransaction)
- 获取事务时,会检查是否允许嵌套事务来设置是否需要保存点
- 4.5获取事务后,检查是否已存在事务,若存在,则在已存在的事逻辑中继续处理  
(AbstractPlatformTransactionManager.handleExistingTransaction,该方法中处理各事务传播情况,如NEVER时抛出异常等),反之开始事务
- #DataSourceTransactionManager.doBegin(),开始从dataSource中获取Connection,若为动态数据源时,则会调用AbstractRoutingDataSource实现类的  
determineCurrentLookupKey获取对应数据源再获取Connection,  
#然后设置连接只读和事务的隔离级别  
con.setReadOnly()  
con.setTransactionIsolation(definition.getIsolationLevel())  
#设置自动提交为false,关闭自动提交  
con.setAutoCommit(false)
- 然后prepareTransactionalConnection(),若强制设置只读,则将事务设置为只读  
然后将当前事务绑定到ThreadLocal中  
TransactionSynchronizationManager.bindResource(getDataSource(),  
txObject.getConnectionHolder());,为当前数据源绑定hold的连接  
#事务设置结束,开始执行被@Transactional注解的方法  
select查询不使用事务开启时获取的连接,单独重新获取新连接,  
更新操作时,从transaction中获取到绑定的Connection进行处理  
#最后方法结束,提交事务,事务结束后,doCleanupAfterCompletion,会将连接的autocommit设置为true,然后重置连接的隔离级别,然后设置链接readOnly状态
- if (definition.getPropagationBehavior() ==  
TransactionDefinition.PROPROPAGATION\_NEVER) {  
    throw new IllegalStateException(  
        "Existing transaction found for transaction marked with propagation  
'never'");

```

}
- isolation_level:level_name
1:READ UNCOMMITTED
2:READ COMMITTED
4:REPEATABLE READ
8:SERIALIZABLE
- Oracle设置隔离级别(支持 2个2,8):
ALTER SESSION SET ISOLATION_LEVEL = READ COMMITTED
ALTER SESSION SET ISOLATION_LEVEL = SERIALIZABLE
- MySQL设置隔离级别(支持4个1,2,4,8):
SET SESSION TRANSACTION ISOLATION LEVEL [LEVEL_NAME]
- PostgreSQL设置隔离级别(支持4个1,2,4,8):
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL [LEVEL_NAME]
- 连接只读
- PostgreSQL设置连接为只读或读写:
SET SESSION CHARACTERISTICS AS TRANSACTION [READ ONLY | READ WRITE]
- 设置事务只读
- SET TRANSACTION READ ONLY
- 设置自动提交:
setAutoCommit
- Oracle连接为手动提交事务
- MySQL设置自动提交(1:是,0:否):
SET autocommit=1|0
- 查询事务隔离级别
- MySQL:
SELECT @@global.transaction_isolation;
SELECT @@session.transaction_isolation;
SHOW VARIABLES LIKE '%isolation%';

```

- Mybatis设置数据源, Spring Transaction也需要设置数据源, Mybatis在select查询时,单独从自己的数据源中获取连接,在开启事务后,执行update语句时,使用mybatis-spring从SpringTransaction的事务中获取连接;不开启事务时,获取到的Connection为自动提交的;
- Spring常见面试问题解析
- SpringMVC
  - SpringMVC执行流程解析
  - 六大组件介绍
    - MVCE大组件分析(DispatcherServlet. Handler、 View)
    - 其他三大组件分析(HandlerMapping. HandlerAdapter, ViewResolver)
  - 注解解析
    - @Controller
    - ...
  - SpringMVC源码解析
    - DispatcherServlet流程

- 初始化流程
  - 访问处理流程
  - 拦截器处理流程
- RequestMappingHandlerMapping工作流程
  - 初始化流程
  - 处理流程
- RequestMappingHandlerAdapter工作流程
  - 初始化流程
  - 处理流程
  - 参数绑定流程
    - 类型转换
  - 返回值处理流程
- SpEL
- Spring中用到的设计模式
  - 简单工厂(静态工厂方法模式): Spring中BeanFactory
  - 工厂方法模式: FactoryBean
  - 单例模式: BeanFactory是单例, 默认的Spring容器都是单例
  - 适配器模式: AOP, 拦截器
    - org.springframework.context.event.  
GenericApplicationListenerAdapter 基本ApplicationListener适配器,用于检测支持的事件类型  
EventPublishingRunListener.starting()时通过对指定类型的  
ApplicationListener发送广播(即invoke这些Listener的onApplicationEvent()  
方法)
  - 装饰器模式: 各种Wrapper,Decorator  
如创建Bean时的BeanWrapper
  - 代理模式: AOP就是代理模式,有2种代理:JDK动态代理(只支持有接口的类), CGLIB  
代理(基于asm,用于操作字节码)
  - 观察者模式: ApplicationListener,事件驱动
  - 策略模式: 实例化对象用到的Strategy模式,即  
org.springframework.factory.support.InstantiationStrategy接口
  - 模版方法:  
JdbcTemplate,RestTemplate,RedisTemplate,AbstractApplicationContext
- SpringIOC容器和SpringMVC容器
  - Spring IOC(父)和SpringMVC IOC(子)容器是父子关系,  
SpringIOC通过ContextLoaderListener,且先于SpringMVC加载;  
SpringMVC容器通过DispatcherServlet加载;
    - Spring使用org.springframework.web.context.ContextLoaderListener加载  
IOC  
SpringMVC使用org.springframework.web.servlet.DispatcherServlet.init()

(FrameworkServlet.init)加载MVC IOC

- Spring/SpringMVC 2个ioc容器: MVC单独负责View Controller 映射处理,使用DispatcherServlet初始化及访问入口处理, Spring负责Service,DAO及事务相关容器处理,使用ContextLoaderListener初始化  
Spring IOC为SpringMVC IOC的父容器,子容器可以访问父容器Bean,父容器Bean不能访问子容器Bean  
二者在同时使用时,若都扫描了整个项目,则会出现ControllerBean多次初始化,导致异常,需要单独为Spring IOC和SpringMVC IOC指定各自扫描包范围
- 子容器可以访问父容器中的bean, 但是父容器不可以访问子容器中的bean。
- 方案一, 传统型:  
父上下文容器中保存数据源、服务层、DAO层、事务的Bean。  
子上下文容器中保存Mvc相关的Action的Bean。  
事务控制在服务层。  
由于父上下文容器不能访问子上下文容器中内容, 事务的Bean在父上下文容器中, 无法访问子上下文容器中内容, 就无法对子上下文容器中Action进行AOP (事务)。

## SpringCloud

- SpringBoot
  - 加载流程  
(通过org.springframework.boot.SpringApplication.run()入口加载,若为WebApplication项目时,默认ApplicationContext为org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext,其他情况下为org.springframework.context.annotation.AnnotationConfigApplicationContext)
  - new org.springframework.boot.SpringApplication(source).run()  
首先创建SpringApplication对象,并初始化ApplicationContextInitializer,然后加载所有ApplicationListener,  
(ApplicationContextInitializer,ApplicationListener从jar包中/META-INF/spring.factories中读取,使用SpringApplication.getSpringFactoriesInstances(ApplicationContextInitializer.class)  
SpringApplication.getSpringFactoriesInstances(ApplicationListener.class),使用org.springframework.core.io.support.SpringFactoriesLoader)  
设置主方法  
然后执行SpringApplication.run():
- 设置StopWatch,  
配置headlessProperty  
!!!获取SpringApplicationRunListener(EventPublishingRunListener) 并将ApplicationListeners添加到SimpleApplicationEventMulticaster 上  
!!!开启SpringApplicationRunListeners.starting(),  
实际为EventPublishingRunListener.starting(),该步骤将触发SpringApplication创建时扫描到的与ApplicationStartedEvent事件相关的Listener的onApplicationEvent()方法;

【其中会触发1.1RestartApplicationListener.onApplicationEvent()方法进行初始化Restart,初始化后立即开启restaredMain重启应用,重新进行SpringApplication.run()操作,第二次启动事件广播时,由于Restarter为单例对象,且已经加载,所以不再进行初始化,即不再重启SpringApplication  
1.2然后触发LoggingApplicationListener的onApplicationEvent()事件,进行日志初始化】

- 创建applicationArguments
- prepareEnvironment准备环境  
创建环境对象,并配置环境信息  
配置PropertySource,并解析args添加到配置列表中;  
然后配置Profile,加载spring.profiles.active配置,并更新activeProfiles列表  
(org.springframework.core.env.AbstractEnvironment),  
!!!并触发所有支持ApplicationEnvironmentPreparedEvent 类型的  
SpringApplicationRunListener.onApplicationEvent  
【3.1首先触发BootstrapApplicationListener,加载spring.factories和  
bootstrap.properties/.yml配置,再次执行SpringApplication.run创建  
BootstrapContext,Bootstrap创建完后再继续处理restartedMain中  
SpringApplication.run->prepareEnvironment流程】
- printBanner 打印Banner信息,并返回Banner对象
- createApplicationContext()创建ApplicationContext,若为Web(通过检查是否存在  
javax.servlet.Servlet或  
org.springframework.web.context.ConfigurableWebApplicationContext)应用时创建  
AnnotationConfigEmbeddedWebApplicationContext,反之为  
AnnotationConfigApplicationContext
- !!! prepareContext准备context,  
postProcessApplicationContext()  
applyInitializers(context);  
触发listeners.contextPrepared(context);  
注册springApplicationArguments为singletonBean  
!!!load(context, sources.toArray(new Object[sources.size()])); 通过Source加载Beans  
【创建BeanDefinitionLoader->load()加载(BeansDefinitionLoader.load(objectSource))
- 先加载class  
org.springframework.cloud.bootstrap.config.PropertySourceBootstrapConfiguratio  
n 相关Bean,使用AnnotatedBeanDefinitionReader.registerBean(Class  
annotatedClass)注册Bean  
】  
最后触发listeners.contextLoaded(context);
- !!!refreshContext() 刷新上下文,刷新BeanFacotry,扫描BeanDefinitions,加载预初始化  
Bean,启动Spring;  
同时在OnRefresh()时调用  
org.springframework.boot.context.embedded.EmbeddedWebApplicationContext.o  
nRefresh()->创建  
EmbeddedServletContainer(即创建Tomcat容器),首先通过扫描  
org.springframework.boot.context.embedded.EmbeddedServletContainerFactory

相关Bean获取EmbeddedServletContainerFactory工厂来创建EmbeddedServletContainer,其中只能有1个ContainerFactory存在,否则抛出异常,然后创建

tomcat实例,然后beanFactory.preInstantiateSingleton(),初始化所有Bean(含Controller),知道Tomcat启动完成;(此处是在创建完BootstrapContext后,继续SpringApplication.run流程中触发,)

!!!SpringBoot启动后,main线程和restartMain线程结束,tomcat线程提供服务

- SpringBoot启动流程简述:

- 通过主方法执行org.springframework.boot.SpringApplication.run()方法开始加载SpringBoot,执行真正的run方法前会初始化SpringApplication,然后扫描所有ApplicationListener(其中会包含devtool的RestartApplicationListener,BootstrapApplicationListener,在spring-boot-devtools/META-INF/spring.factories中配置),Listener在/META-INF/spring.factories中配置扫描
- run()流程中首先设置Headless属性,然后扫描注册所有SpringApplicationRunListener,实际为EventPublishingRunListener,并将1中扫描到的ApplicationListener列表注册到EventPublishingRunListener的multicast,然后向扫描到的Listener广播ApplicationStartedEvent事件
- 广播ApplicationStartedEvent时,若ApplicationListener列表中存在RestartApplicationListener(优先级最高,Ordered.HIGHEST\_PRECEDENCE=Integer.MIN\_VALUE)时,在RestartApplicationListener的onApplicationEvent事件处理中,启动新的restartMain线程,使用SpringApplicationBuilder重新加载SpringApplication.run方法,然后主线程join,restartMain线程执行完后退出。  
(可通过/restart接口重启服务,在restartMain线程中重新启动SpringApplication时还会向RestartListener广播ApplicationStartedEvent,但由于Restart是单例对象,所以不会再次触发重启操作)
- 若存在BootstrapApplicationListener(优先级较高,Ordered.HIGHEST\_PRECEDENCE+5),则在BootstrapApplicationListener.onApplicationEvent方法中利用SpringApplicationBuilder中构造SpringApplication对象,再次重启SpringApplication并加载Bootstrap相关配置,并创建BootstrapContext  
(BootstrapContext加载时,重新触发BootstrapApplicationListener.ApplicationStartedEvent时,不再进行处理)
- BootstrapContext启动后,继续Application的run方法:  
打印Banner,创建ApplicationContext(AnnotationConfigApplicationContext或AnnotationConfigEmbeddedApplicationContext)
- prepareContext 准备上下文环境,会触发ApplicationEvent
- refreshContext刷新上下文,即调用AbstractApplicationContext.refresh() 扫描Bean,加载beanDefinitions(),并初始化所有单例Bean,在onRefresh时,调用EmbeddedWebApplicationContext创建tomcat容器

- 然后afterRefresh,listener.finished  
并触发响应的事件  
最后启动完成
  - SpringBoot事件广播机制  
(EventMulticaster)
    - org.springframework.boot.context.event.EventPublishingRunListener  
important SpringApplicationRunListener 用于监听SpringApplication.run  
方法去发布事件
    - org.springframework.context.event.SimpleApplicationEventMulticaster  
extends AbstractApplicationEventMulticaster 简单实现  
ApplicationEventMulticaster事件广播器,用于对ApplicationListener进行广  
播(观察者模式)
    - org.springframework.context.ApplicationListener extends EventListener :  
Application event listeners 的实现接口  
GenericApplicationListener extends ApplicationListener 基础  
ApplicationListener,提供检查支持的EventType类型及支持的SourceType类型
      - org.springframework.boot.devtools.restart.RestartApplicationListen  
er(Order:Int.Min) 初始化  
org.springframework.boot.devtools.restart.Restarter的  
ApplicationListener,  
用于处理  
ApplicationStartingEvent,ApplicationPreparedEvent,ApplicationRead  
yEvent,ApplicationFailedEvent事件  
devtools.Restart对象,用于重启应用  
(若存在devtools的RestartApplicationListener时,在执行Restart初始化  
时 SpringApplication 会进行一次重启,重启时使用restartedMain线程进  
行,此时主线程等待join)  
第一次初始化Restart时,在main主线程中进行,  
在重启SpringApplication.run()时,进行第二次事件广播,由于Restart对象  
为单例对象,所以不需要再次初始化,即不再执行重启操作
        - 当接收的事件为ApplicationStartingEvent时,Restart会进行初始化,  
并使用新线程restartedMain立即重启应用,重新进行  
SpringApplication.run()的操作
    - org.springframework.boot.logging.LoggingApplicationListener(Orde  
r:Int.Min+20) 配置日志系统的ApplicationListener.  
如果环境配置中包含logging.config则用于引导日志系统,反之使用默认配  
置。无论如何,当环境配置中包含logging.level.\*条目时,将会被定制处  
理。  
用于SpringBoot日志处理监听
    - org.springframework.boot.autoconfigure.BackgroundPreinitializer  
在耗时任务的后台线程中触发早期初始化



- org.springframework.cloud.bootstrap.BootstrapApplicationListener(
 Order: Int.Min+5)
 

在一个单独的bootstrap context中通过ApplicationContextInitializer 来准备SpringApplication,bootstrap context 是通过spring.factories定义的源作为BootstrapConfiguration,并且通过bootstrap.properties(或.xml,.yml,.yaml)配置文件来初始化

  - BootstrapContext初始化,初始化时,会使用SpringApplicationBuilder.run()再次启动SpringApplication.run()方法加载BootstrapContext
 

在启动BootstrapContext过程中,不再执行BootstrapApplicationListener;

BootstrapContext加载时preInstantiateSingletons()加载propertySourceBootstrapConfiguration propertyPlaceholderAutoConfiguration等Bean
- org.springframework.cloud.bootstrap.LoggingSystemShutdownListener
- org.springframework.boot.context.config.ConfigFileApplicationListener
- org.springframework.boot.context.config.AnsiOutputApplicationListener
- org.springframework.boot.logging.ClasspathLoggingApplicationListener
- org.springframework.boot.context.config.DelegatingApplicationListener
- org.springframework.cloud.context.restart.RestartListener
- org.springframework.boot.builder.ParentContextCloserApplicationListener
- org.springframework.boot.ClearCachesApplicationListener
- org.springframework.boot.context.FileEncodingApplicationListener
- org.springframework.boot.liquibase.LiquibaseServiceLocatorApplicationListener
- @Configuration 加载原理
  - org.springframework.context.annotation.Configuration注解
    - 1.该注解指示一个类声明了一个或多个@Bean方法,并且可以由Spring容器进行处理
    - 2.加载 @Configuration 类
      - 2.1通过AnnotationConfigApplicationContext(org.springframework.context.annotation):通过AnnotationBeanDefinitionReader.register(),registerBean(class)加载注解标注的类

@Configuration通常使用AnnotationConfigApplicationContext或支持Web的AnnotationConfigWebApplicationContext来加载,如:

```
AnnotationConfigApplicationContext ctx = new
```

```

AnnotationConfigApplicationContext();
ctx.register(AppConfig.class);
ctx.refresh();
ctx.getBean(MyBean.class);

```

2.2通过Spring xml: @Configuration还可以Spring XML配置文件中声明,及在标签中设置[context:annotation-config](#)即可

```

<context: annotation-config />
<bean class=" com.acme.AppConfig" />
</ beans>

```

2.3通过@ComponentScan组件扫描加载

注解被@Component元注解标注,所以标注为@Configuration的类会被扫描为Bean,通常使用SpringXml的[context:component-scan](#)

- @Configuration使用外部值

1.使用EnvironmentAPI

在@Configuration注入Environment envBean,通过

env.getProperty("bean.name")获取外部值;

同时可以使用@PropertySource("classpath:.property")配置属性源,

也可以使用@ConfigurationProperties来设置配置前缀

2.使用Value注解(@Value("\${propertyName}"))

@Configuration

@PropertySource("classpath:/com/acme/app.properties")

```

public class AppConfig {

```

```

    @Value("${bean.name}") String beanName;

```

```

    @Bean

```

```

    public MyBean myBean() {

```

```

        return new MyBean(beanName);

```

```

    }

```

```

}

```

- 编写@Configuration类

1.使用 @Import注解,使用@Import注解可以导入其他@Configuration类,类似

SpringXML中的标签,@Import的类可以通过构造方法注入

- 使用@Profile注解,添加@Profile注解可以指示仅当提供的Profile配置为active时才处理该类;另外,@Profile还可以配置在@Bean方法上

- 使用@ImportResource注解导入SpringXML配置文件,然后使用@Inject注入XML中的Bean

4.使用嵌套@Configuration类,可在@Configuration类中嵌套@Configuration并使用

@Inject注入该嵌套类

- //1. @Import注解示例

@Configuration

```

public class DatabaseConfig {

```

```

    @Bean

```

```

    public DataSource dataSource() {

```

```

        // instantiate, configure and return DataSource

```

```

    }

```

```

}

```

```

@Configuration
@Import(DatabaseConfig.class)
public class AppConfig {
    private final DatabaseConfig dataConfig;
    public AppConfig(DatabaseConfig dataConfig) {
        this.dataConfig = dataConfig;
    }
    @Bean
    public MyBean myBean() {
        // reference the dataSource() bean method
        return new MyBean(dataConfig.dataSource());
    }
}

- //2.@Profile注解示例
@Profile("development")
@Configuration
public class EmbeddedDatabaseConfig {
    @Bean
    public DataSource dataSource() {
        // instantiate, configure and return embedded DataSource
    }
}

//@Profile标注在@Bean方法上
@Configuration
public class ProfileDatabaseConfig {
    @Bean("dataSource")
    @Profile("development")
    public DataSource embeddedDatabase() { ... }
    @Bean("dataSource")
    @Profile("production")
    public DataSource productionDatabase() { ... }
}

- //3. @ImportResource注解示例
@Configuration
@ImportResource("classpath:/com/acme/database-config.xml")
public class AppConfig {
    @Inject DataSource dataSource; // from XML
    @Bean
    public MyBean myBean() {
        // inject the XML-defined dataSource bean
        return new MyBean(this.dataSource);
    }
}

- //嵌套@Configuration类示例
@Configuration
public class AppConfig {

```

```

@Inject DataSource dataSource;
@Bean
public MyBean myBean() {
    return new MyBean(dataSource);
}
@Configuration
static class DatabaseConfig {
    @Bean
    DataSource dataSource() {
        return new EmbeddedDatabaseBuilder().build();
    }
}
}

```

- @Configuration下的@Bean默认会被实例化,可以使用@Lazy注解来设置@Bean延迟加载,也可以直接在@Bean上设置@Lazy

- ConfigurationClassPostProcessor(org.springframework.context.annotation)该类为启动处理@Configuration类的BeanFactoryPostProcessor  
使用[context:annotation-config](#)或[context:component-scan](#)时自动注册

- @Configuration的加载位置:

在AbstractApplicationContext.refresh()中的invokeBeanFactoryPostProcessors(),其中PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors()中registryProcessors.postProcessBeanDefinitionRegistry(registry),实际调用org.springframework.context.annotation.ConfigurationClassPostProcessor.postProcessBeanDefinitionRegistry(registry)扫描处理@Configuration类

- @ConfigurationBean加载主要过程

- 1.Spring容器初始化时注册ConfigurationClassPostProcessor
- 2.Spring容器初始化执行refresh()中调用invokeBeanFactoryPostProcessor,并调用其中的ConfigurationClassPostProcessor
- 3.ConfigurationClassPostProcessor处理器使用ConfigurationClassParser完成配置类解析
- 4.ConfigurationClassParser配置内解析过程中完成嵌套的MemberClass、@PropertySource注解、@ComponentScan注解、@ImportResource、@Bean等处理
- 5.完成@Bean注册,@ImportResource指定bean的注册以及@Import(实现ImportBeanDefinitionRegistrar接口方式)的Bean注册
- 6.@Bean注解的方法在解析的时候作为@ConfigurationClass的一个属性,最后转换成BeanDefinition处理,实例化时作为一个工厂方法进行Bean的创建

#### ■ @Bean 加载原理

- org.springframework.context.annotation.Bean 指示一个方法产生一个由Spring容器管理的Bean,语义类似于SpringXML的  
Bean名称: 默认使用被@Bean修饰的方法名,也可以指定多个名称  
@Bean({"b1","b2"}),但不能是方法名  
通常@Bean方法在@Configuration类中声明,在运行时会被CGLIB子类代理,@Bean方法可以调用同类中的其他@Bean方法,为了保证"Bean间引用"和AOP语义,所以@Configuration类和他们的工厂方法不能定义为final或private.  
@Bean精简模式:

@Bean方法也可以在不使用@Configuration中的方法上使用,如在@Component下声明@Bean方法,这种情况下,@Bean方法会按照精简模式处理;容器将精简模式下的Bean视为普通工厂方法,精简模式下不支持"Bean间引用",调用另一个方法时,是普通的Java方法调用,Spring不会通过CGLIB代理拦截调用。在将BeanFactoryPostProcessor设置为@Bean方法返回对象时,需将方法设置为static,因为BeanFactoryPostProcessor需要在容器生命周期很早前就被实例化。

- //语义示例

```
@Bean
public MyBean myBean() {
    // instantiate and configure MyBean obj
    return obj;
}

//@Bean Methods in @Configuration Classes
@Configuration
public class AppConfig {
    @Bean
    public FooService fooService() {
        return new FooService(fooRepository());
    }
    @Bean
    public FooRepository fooRepository() {
        return new JdbcFooRepository(dataSource());
    }
    // ...
}

//@Bean Lite Mode
@Bean
public static PropertySourcesPlaceholderConfigurer pspc() {
    // instantiate, configure and return pspc ...
}
```

- @Profile指示一个或多个指定配置文件处于活动时,该组件有资格注册,通过ConfigurableEnvironment.setActiveProfiles()或声明spring.profiles.active作为JVM属性,或在测试集成中添加@ActiveProfiles配置处于活动的Profile;可以以下方式声明:
  - 1.在任何被@Component注释的类上,包括@Configuration
  - 2.作为元注解,构造自定义注解
  - 3.作为任何@Bean方法的方法级注解@Profile支持简单字符串和逻辑表达式(&,|,!),如@Profile("p1 & p2") 或@Profile({"p1","!p2"}),且表达式不能连写,如a&b&c是错误写法,需要加括号,如(a&b)|c;在不指定字符串时,表示任何profile下都有效@Scope 作用范围,与@Component类一起时,指示被注解的类的作用范围名称;与方法级@Bean一起时,@Scope指示该@Bean返回的对象实例的范围名称@Lazzy指示是否要延迟初始化Bean

@Component和@Bean上添加@Lazy注解时表示延迟加载该Bean,@Configuration上添加@Lazy注解时表示延迟加载@Configuration中所有的@Bean及import  
@DependsOn当前Bean所依赖的Bean,当不通过属性或构造方法注入时可以使用该注解声明依赖Bean  
@Primary指示当多个候选者有资格自动装配时,应该优先考虑Bean,若候选对象中只有1个@Primary时,该Bean为默认装配值  
@Order 定义容器注入优先级

- @ComponentScans/@ComponentScan
- SpringCloud架构
  - SpringCloud架构结构
  - 访问入口负载: Nginx
  - Gateway
    - Zuul(SpringCloud 1+)
      - 请求转发HttpClient(RouteHost!=null)或Ribbon(RouteHost==null&&serviceId!=null的请求)
      - org.springframework.cloud.netflix.zuul.filters.pre.PreDecorationFilter  
SimpleHostRoutingFilter  
RibbonRoutingFilter
    - Spring Gateway(SpringCloud2+)
      - SpringGateway由各种过滤器Filter实现(GlobalFilter),默认使用Netty实现Gateway网关,主要包括: NettyRoutingFilter(Http路由过滤器),WebsocketRoutingFilter(ws路由过滤器)
- 服务间相互调用
  - OpenFeign,底层使用HttpClient(默认)或OKHttp,底层为restTemplate,为Http RESTful协议  
OpenFeign由Ribbon(负载均衡/服务发现)+Hystrix(熔断)组成
    - Ribbon(负载均衡/服务发现)
      - 可使用@LoadBalance对restTemplate启用负载均衡
    - Hystrix(熔断和重试)
      - com.netflix.hystrix.AbstractCommand,创建HystrixCommand对象  
commandGroup=serverName  
commandKey=Feign接口的接口名#方法名(类型列表)  
如:BaseStudyServiceClient#getSystemConfig(BigDecimal)
      - 服务对应FeignLoadBalancer会缓存在org.springframework.util.ConcurrentReferenceHashMap中,默认为软引用,在内存回收时,数据被回收(软引用在内存不足时回收(原理是软引用对象在距离上次GC时间内没有被使用然后回收,与剩余堆空键大小有关),弱引用在任何GC中都回收)

- org.springframework.cloud.netflix.Feign.@FeignClient注解标记的对象,会被使用JDKProxy代理,并添加
   
feign.hystrix.HystrixInvocationHandler代理处理方法
   
this.h.invoke(this, m20, new Object[] { paramBigDecimal })
   
1.在invoke方法中创建HystrixCommand,并重写run方法,
   
2.run方法中再执行feign.SynchronousMethodHandler.invoke,
   
创建feign.RequestTemplate,和Retryer(clone新的
   
Retryer,feign.RetryableException触发重试)
   
public Object invoke(Object[] argv) {
   
RequestTemplate template =
   
this.buildTemplateFromArgs.create(argv);
   
Retryer retryer = this.retryer.clone();
   
while(true) {
   
try {
   
return this.executeAndDecode(template);
   
}
   
}
   
3.然后再使用
   
org.springframework.cloud.netflix.Feign.LoadBalancerFeignClient.execute执行请求
   
4.LoadBalancerFeignClient.execute中创建
   
FeignLoadBalancer.RibbonRequest对象,
   
然后获取当前ServerName的
   
FeignLoadBalancer(org.springframework.cloud.netflix.feign.ribbon.CachingSpringLoadBalancerFactory)
   
lbClientFactory.create(clientName)
   
#此处获取FeignLoadBalancer时有缓存,保存为
   
org.springframework.util.ConcurrentReferenceHashMap<serviceName,balancer>,默认为软引用,对键和值使用软引用或弱引用的
   
ConcurrentHashMap,支持空键和空值,可指定为若应用或软引用;
   
FeignLoadBalancer底层
   
为:com.netflix.loadbalancer.ZoneAwareLoadBalancer (extends
   
com.netflix.loadbalancer.DynamicServerListLoadBalancer)
   
若支持重试机制则创建
   
org.springframework.cloud.netflix.feign.ribbon.RetryableFeignLoadBalancer,反之创建FeignLoadBalancer,然后带LoadBalancer执行
   
executeWithLoadBalancer()
   
{com.netflix.client.AbstractLoadBalancerAwareClient#executeWithLoadBalancer}
   
5.然后从loadBalancer中选出一个server进行请求,默认使用轮询算法(还有RandomRule随机算法,BestAvailableRule选取并发最少服务,WeightedResponseTimeRule加权响应时间规则),
   
最终使用RetryTemplate执行请求
   
  - 3.
   
org.springframework.cloud.netflix.feign.ribbon.LoadBalancerFeignClient

```

@Override
public Response execute(Request request, Request.Options
options) throws IOException {
try {
URI asUri = URI.create(request.url());
String clientName = asUri.getHost();
URI uriWithoutHost = cleanUrl(request.url(), clientName);
FeignLoadBalancer.RibbonRequest ribbonRequest = new
FeignLoadBalancer.RibbonRequest(
this.delegate, request, uriWithoutHost);
IClientConfig requestConfig = getClientConfig(options,
clientName);
return
lbClient(clientName).executeWithLoadBalancer(ribbonRequest,
requestConfig).toResponse();
}
catch (ClientException e) {
IOException io = findIOException(e);
if (io != null) {
throw io;
}
throw new RuntimeException(e);
}
}
}

```

- #1. 获取到LoadBalancer后server存活检测  
(com.netflix.loadbalancer.BaseLoadBalancer中处理):  
BaseLoadBalancer使用PingTask,对Server列表进行存活检测,并将alive  
的server放到upServerList列表,PingTask 10s检测一次;  
com.netflix.loadbalancer.BaseLoadBalancer().lbTime.schedule(new  
PingTask,10\*1000);
- 2.Server列表更新  
(com.netflix.loadbalancer.DynamicServerListLoadBalancer中处理):  
在创建FeignLoadBalancer时(FeignLoadBalancer包装了一层  
com.netflix.loadbalance.DynamicServerListLoadBalancer)时,会更新  
一次Server列表,并开启定时  
(java.util.concurrent.ScheduledThreadPoolExecutor)默认每30s一次更  
新Server列表  
com.netflix.loadbalancer.PollingServerListUpdater().start();
- openfeign 调用根据serviceName获取服务列表是否有缓存?:  
答:有缓存,2层,一层是serviceName对应的LoadBalancer的缓存(使用  
org.springframework.util.ConcurrentReferenceHashMap Key-Value软  
引用),二层缓存是LoadBalancer的  
serverList(Collections.synchronizedList)  
每30s从zookeeper中更新一次,没10s ping一次server列表

- 授权中心



- SpringSecurity
- 认证中心
  - SpringSecurity
- 服务注册与发现
  - Zookeeper
  - Eureka
  - Consul
- 配置中心
  - Apollo(携程)
  - Zookeeper
- 消息队列
  - Kafka
  - ActiveMQ
- 缓存服务
  - Redis
  - Mongo
- 数据库
  - MySQL
  - Oracle
  - PostgreSQL
- 打包部署
  - Maven
  - Jenkins
  - Docker
  - K8S
- 监控
  - Zabbix
  - Grafana
  - PINPOINT(请求链路监控)
- 日志
  - Kafka+ES
  - GrayLog
- SpringCloud相关技术
  - Zookeeper
  - Eureka
  - Feign(Ribbon+Hystrix)
- 注解解析
- 常用包
- 常用方法

- 实现

org.springframework.web.servlet.config.annotation.WebMvcConfigurer.addInterceptors添加org.springframework.web.servlet.HandlerInterceptor来处理SpringMVC Controller请求

- @Component

```
public class SeataHandlerInterceptorConfiguration implements
WebMvcConfigurer {
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new
        MyHandlerInterceptor()).addPathPatterns("/**");
    }
}
```

- 实现feign.RequestInterceptor.apply()方法,可对Feign请求进行操作

- @Component

```
public class BasicAuthRequestInterceptor implements
RequestInterceptor {
    private final String headerValue;
    public BasicAuthRequestInterceptor(String username, String password) {
        this(username, password, Util.ISO_8859_1);
    }
    public BasicAuthRequestInterceptor(String username, String password,
    Charset charset) {
        Util.checkNotNull(username, "username", new Object[0]);
        Util.checkNotNull(password, "password", new Object[0]);
        this.headerValue = "Basic " + base64Encode((username + ":" +
        password).getBytes(charset));
    }
    private static String base64Encode(byte[] bytes) {
        return Base64.encode(bytes);
    }
    public void apply(RequestTemplate template) {
        template.header("Authorization", new String[]{this.headerValue});
    }
}
```

- 定时任务

- ElasticJob

- Scheduling

如何在SpringCloud中只有1个实例执行定时任务:

使用

org.springframework.cloud.client.discovery.DiscoveryClientgetInstances(serviceName); 获取服务注册列表中各服务IP, 若当前服务IP为最小/最大时执行任务

- Quartz

# Spring工具包

## 分页

## SpringCloud分布式架构与传统单体架构的优缺点

- 传统单体架构
  - 优点
    - (SSH/SSM)结构简单,适合小型项目,开发速度快,运维简单方便
  - 缺点
    - 传统单体应用中,将所有功能的表示层,业务逻辑层,数据访问层,包括静态资源都耦合在一个工程里,当业务越来越复杂,功能越来越多,参与的开发人员越来越多时,会出现:
      - 1.业务变复杂,代码量增大,代码可读性,可维护性,可扩展性下降
      - 2.单体应用扩展能力有限,访问量大时,应用响应能力下降,影响整个系统的访问
      - 3.单体应用容错率低,某个模块出问题,有可能导致整个应用崩溃
- SpringCloud架构
  - 优点
    - 将应用按业务服务划分多个功能模块项目服务,每个服务只负责自己的功能职责,降低了业务和项目代码耦合,以及可分别对不同的项目扩展维护,而不影响整个系统。主要优点有:
      - 1.按业务划分的微服务单元独立部署,运行在独立的环境中,各服务环境不耦合,有很好的扩展性和复用性
      - 2.服务之间通常采用RPC协议(http或Socket长连接),这种通信机制与平台和语言无关
      - 3.各微服务单元可使用自己独立的数据库,可降低数据库服务压力,提高服务吞吐效率
      - 4.服务集中化管理(服务注册于发现),监控服务监控状况等
      - 5.微服务架构是分布式架构,服务具有高可用特性
      - 6.使用容器化部署,提高了系统自动化部署能力
  - 缺点
    - - 1.项目构建过程复杂,微服务业务模块划分难度大,前期项目实现复杂,编码量增多(通常会有多层级的架构划分)
      - 2.服务调用链路边长,增加了服务调用成本及服务访问时间
      - 3.分布式系统中数据一致性难以保证,通常需要单独部署分布式事务系统,尽可能的保证事务一致性
- 单体架构升级微服务架构后的变化
  - - 1.服务可用性提高了,如:单体应用时,某个功能访问压力大,会导致整个系统响应慢,甚至整个系统不可用,升级后,个别服务不可用,不影响整个系统中其他功能的访问,同时,可单独对某个服务进行动态扩展资源,提高服务的可用性.
    - 2.服务可扩展性提高了,如:单体应用时,扩展服务需要单独新增tomcat,再复制war包,再启动项目,同时还得配置访问入口,升级后,由于微服务架构使用容器化管理部署,扩展时,只需要配置实例数即可,扩展速度快,生效时间快
    - 3.编码方面,单体架构时,如果做周边扩展业务时,会新开项目,重新集成实体关系映射,

这样会导致实体关系映射会维护在多个项目代码中,如果升级数据库表时,所有相关的项目都需要修改代码,处理成本高;升级微服务项目后,业务模块被单独划分,BASE(数据实体层)分别有各自的微服务项目,只需要处理一份代码即可,同时代码结构清晰明了。

## ORM

### Mybatis

<https://mybatis.org/mybatis-3/zh/getting-started.html>

- MyBatis 是一款优秀的持久层框架，它支持自定义 SQL、存储过程以及高级映射。MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

- maven:

```
org.mybatis
mybatis
x.x.x
```

- 从XML中构建SqlSessionFactory:  
每个基于 MyBatis 的应用都是以一个 SqlSessionFactory 的实例为核心的。SqlSessionFactory 的实例可以通过 SqlSessionFactoryBuilder 获得。而 SqlSessionFactoryBuilder 则可以从 XML 配置文件或一个预先配置的 Configuration 实例来构建出 SqlSessionFactory 实例。

- xml:

```
String resource = "org/mybatis/example/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
```

- mybatis-config.xml

environment 元素体中包含了事务管理和连接池的配置。mappers 元素则包含了一组映射器（mapper），这些映射器的 XML 映射文件包含了 SQL 代码和映射定义信息。

- 代码构建SqlSessionFactory:

```
DataSource dataSource = BlogDataSourceFactory.getBlogDataSource();
TransactionFactory transactionFactory = new JdbcTransactionFactory();
Environment environment = new Environment("development", transactionFactory,
dataSource);
Configuration configuration = new Configuration(environment);
configuration.addMapper(BlogMapper.class);
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(configuration);
```

注意该例中，configuration 添加了一个映射器类（mapper class）。映射器类是 Java 类，它们包含 SQL 映射注解从而避免依赖 XML 文件。不过，由于 Java 注解的一些限制以及某些 MyBatis 映射的复杂性，要使用大多数高级映射（比如：嵌套联合映射），仍然需要使用 XML 配置。有鉴于此，如果存在一个同名 XML 配置文件，MyBatis 会自动查找并加载它（在

这个例子中，基于类路径和 BlogMapper.class 的类名，会加载 BlogMapper.xml)。

- 从 SqlSessionFactory 中获取 SqlSession

- 既然有了 SqlSessionFactory，顾名思义，我们可以从中获得 SqlSession 的实例。SqlSession 提供了在数据库执行 SQL 命令所需的所有方法。你可以通过 SqlSession 实例来直接执行已映射的 SQL 语句。例如：

```
try (SqlSession session = sqlSessionFactory.openSession()) {  
    Blog blog = (Blog)  
    session.selectOne("org.mybatis.example.BlogMapper.selectBlog", 101);  
}
```

- 使用和指定语句的参数和返回值相匹配的接口（比如 BlogMapper.class）

```
try (SqlSession session = sqlSessionFactory.openSession()) {  
    BlogMapper mapper = session.getMapper(BlogMapper.class);  
    Blog blog = mapper.selectBlog(101);  
}
```

- 作用域和声明周期

- SqlSessionFactoryBuilder:

这个类可以被实例化、使用和丢弃，一旦创建了 SqlSessionFactory，就不再需要它了。因此 SqlSessionFactoryBuilder 实例的最佳作用域是方法作用域（也就是局部方法变量）。你可以重用 SqlSessionFactoryBuilder 来创建多个 SqlSessionFactory 实例，但最好还是不要一直保留着它，以保证所有的 XML 解析资源可以被释放给更重要的事情。

- SqlSessionFactory

SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由丢弃它或重新创建另一个实例。使用 SqlSessionFactory 的最佳实践是在应用运行期间不要重复创建多次，多次重建 SqlSessionFactory 被视为一种代码“坏习惯”。因此 SqlSessionFactory 的最佳作用域是应用作用域。有很多方法可以做到，最简单的就是使用单例模式或者静态单例模式。

- SqlSession

每个线程都应该有它自己的 SqlSession 实例。SqlSession 的实例不是线程安全的，因此是不能被共享的，所以它的最佳的作用域是请求或方法作用域。绝对不能将 SqlSession 实例的引用放在一个类的静态域，甚至一个类的实例变量也不行。

```
try (SqlSession session = sqlSessionFactory.openSession()) {  
    // 你的应用逻辑代码  
}
```

- 映射器实例

映射器是一些绑定映射语句的接口。映射器接口的实例是从 SqlSession 中获得的。虽然从技术层面上来讲，任何映射器实例的最大作用域与请求它们的 SqlSession 相同。但方法作用域才是映射器实例的最合适的作用域。也就是说，映射器实例应该在调用它们的方法中被获取，使用完毕之后即可丢弃。映射器实例并不需要被显式地关闭。尽管在整个请求作用域保留映射器实例不会有什么问题，但是你很快会发现，在这个作用域上管理太多像 SqlSession 的资源会让你忙不过来。因此，最好将映射器放在方法作用域内。就像下面的例子一样：

```
try (SqlSession session = sqlSessionFactory.openSession()) {  
    BlogMapper mapper = session.getMapper(BlogMapper.class);  
    // 你的应用逻辑代码
```

}

- 配置解析

configuration (配置)

- 属性 (properties)

这些属性可以在外部进行配置，并可以进行动态替换。你既可以在典型的 Java 属性文件中配置这些属性，也可以在 properties 元素的子元素中设置。例如：

设置好的属性可以在整个配置文件中用来替换需要动态配置的属性值。比如：

也可以在 SqlSessionFactoryBuilder.build() 方法中传入属性值。例如：

```
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, props);  
// ... 或者 ...
```

```
SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, environment, props);
```

- 如果一个属性在不只一个地方进行了配置，那么，MyBatis 将按照下面的顺序来加载：

- #首先读取在 properties 元素体内指定的属性。

- #然后根据 properties 元素中的 resource 属性读取类路径下属性文件，或根据 url 属性指定的路径读取属性文件，并覆盖之前读取过的同名属性。

- #最后读取作为方法参数传递的属性，并覆盖之前读取过的同名属性。

- 因此，通过方法参数传递的属性具有最高优先级，resource/url 属性中指定的配置文件次之，最低优先级的则是 properties 元素中指定的属性。

- 从 MyBatis 3.4.2 开始，你可以为占位符指定一个默认值。例如：

这个特性默认是关闭的。要启用这个特性，需要添加一个特定的属性来开启这个特性。例如：

- 如果你在属性名中使用了 ":" 字符（如：db:username），或者在 SQL 映射中使用了 OGNL 表达式的三元运算符（如：\${tableName != null ? tableName : 'global\_constants'}），就需要设置特定的属性来修改分隔属性名和默认值的字符。例如：

- 设置 (settings) :  
MyBatis 中极为重要的调整设置，它们会改变 MyBatis 的运行时行为。
  - 配置完整的 settings 元素的示例如下：

- 类型别名 (typeAliases)
  - 类型别名可为 Java 类型设置一个缩写名字。它仅用于 XML 配置，意在降低冗余的全限定类名书写。

也可以指定一个包名，MyBatis 会在包名下面搜索需要的 Java Bean，比如：

每一个在包 domain.blog 中的 Java Bean，在没有注解的情况下，会使用 Bean 的首字母小写的非限定类名来作为它的别名。比如 domain.blog.Author 的别名为 author；若有注解，则别名为其注解值。见下面的例子：

```
@Alias("author")
public class Author {
```

```
...  
}
```

- 类型处理器 (typeHandlers)

MyBatis 在设置预处理语句 (PreparedStatement) 中的参数或从结果集中取出一个值时, 都会用类型处理器将获取到的值以合适的方式转换成 Java 类型。

提示 从 3.4.5 开始, MyBatis 默认支持 JSR-310 (日期和时间 API)

- 可以重写已有的类型处理器或创建自己的类型处理器来处理不支持的或非标准的类型。具体做法为: 实现 org.apache.ibatis.type.TypeHandler 接口, 或继承一个很便利的类 org.apache.ibatis.type.BaseTypeHandler, 并且可以 (可选地) 将它映射到一个 JDBC 类型。如:

```
// ExampleTypeHandler.java  
@MappedJdbcTypes(JdbcType.VARCHAR)  
public class ExampleTypeHandler extends BaseTypeHandler {}  
并注册到配置文件typeHandlers中
```

- MyBatis 可以自动查找类型处理器:

注意在使用自动发现功能的时候, 只能通过注解方式来指定 JDBC 的类型。

- 处理枚举类型:

若想映射枚举类型 Enum, 则需要从 EnumTypeHandler 或者 EnumOrdinalTypeHandler 中选择一个来使用。默认情况下, MyBatis 会利用 EnumTypeHandler 来把 Enum 值转换成对应的名字。如:

单独指定typeHandler后,需要使用resultMap,而不能使用resultType

- 对象工厂 (objectFactory)

每次 MyBatis 创建结果对象的新实例时, 它都会使用一个对象工厂

(ObjectFactory) 实例来完成实例化工作。默认的对象工厂需要做的仅仅是实例化目标类, 要么通过默认无参构造方法, 要么通过存在的参数映射来调用带有参数的构造方法。如果想覆盖对象工厂的默认行为, 可以通过创建自己的对象工厂来实现。

- 要么通过存在的参数映射来调用带有参数的构造方法。如果想覆盖对象工厂的默认行为, 可以通过创建自己的对象工厂来实现。比如:

```
// ExampleObjectFactory.java  
public class ExampleObjectFactory extends DefaultObjectFactory {  
    public Object create(Class type) {  
        return super.create(type);  
    }  
    public Object create(Class type, List constructorArgTypes, List  
        constructorArgs) {  
        return super.create(type, constructorArgTypes, constructorArgs);  
    }  
}
```



```

    }
    public void setProperties(Properties properties) {
        super.setProperties(properties);
    }
    public boolean isCollection(Class type) {
        return Collection.class.isAssignableFrom(type);
    }
}

```

- ObjectFactory 接口很简单，它包含两个创建实例用的方法，一个是处理默认无参构造方法的，另外一个处理带参数的构造方法。另外，setProperties 方法可以被用来配置 ObjectFactory，在初始化你的 ObjectFactory 实例后，objectFactory 元素体中定义的属性会被传递给 setProperties 方法。

- 插件 (plugins)

MyBatis 允许你在映射语句执行过程中的某一点进行拦截调用。默认情况下，MyBatis 允许使用插件来拦截的方法调用包括：

#Executor (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)

#ParameterHandler (getParameterObject, setParameters)

#ResultSetHandler (handleResultSets, handleOutputParameters)

#StatementHandler (prepare, parameterize, batch, update, query)

- 通过 MyBatis 提供的强大机制，使用插件是非常简单的，只需实现 Interceptor 接口，并在注解 @Intercepts 中指定想要拦截的方法签名即可。

```
// ExamplePlugin.java
```

```
@Intercepts({@Signature(
```

```
type= Executor.class,
```

```
method = "update",
```

```
args = {MappedStatement.class, Object.class})))
```

```
public class ExamplePlugin implements Interceptor {
```

```
    private Properties properties = new Properties();
```

```
    public Object intercept(Invocation invocation) throws Throwable {
```

```
        // implement pre processing if need
```

```
        Object returnObject = invocation.proceed();
```

```
        // implement post processing if need
```

```
        return returnObject;
```

```
    }
```

```
    public void setProperties(Properties properties) {
```

```
        this.properties = properties;
```

```
    }
```

上面的插件将会拦截在 Executor 实例中所有的“update”方法调用，这里的 Executor 是负责执行底层映射语句的内部对象。

Mybatis 仅可以编写针对 ParameterHandler、ResultSetHandler、StatementHandler、

Executor 这 4 种接口的插件

- 环境配置 (environments)

MyBatis 可以配置成适应多种环境，这种机制有助于将 SQL 映射应用于多种数据库之中，现实情况下有多种理由需要这么做。例如，开发、测试和生产环境需要有不同的配置；或者想在具有相同 Schema 的多个生产数据库中使用相同的 SQL 映射。还有许多类似的使用场景。

不过要记住：尽管可以配置多个环境，但每个 SqlSessionFactory 实例只能选择一种环境。

- environments 元素定义了如何配置环境。

注意一些关键点：

默认使用的环境 ID（比如：default="development"）。

每个 environment 元素定义的环境 ID（比如：id="development"）。

事务管理器的配置（比如：type="JDBC"）。

数据源的配置（比如：type="POOLED"）

- 事务管理器 (transactionManager)

在 MyBatis 中有两种类型的事务管理器（也就是 type="JDBC|MANAGED"）

- #JDBC – 这个配置直接使用了 JDBC 的提交和回滚设施，它依赖从数据源获得的连接来管理事务作用域。

#MANAGED – 这个配置几乎没做什么。它从不提交或回滚一个连接，而是让容器来管理事务的整个生命周期（比如 JEE 应用服务器的上下文）。默认情况下它会关闭连接。然而一些容器并不希望连接被关闭，因此需要将 closeConnection 属性设置为 false 来阻止默认的关闭行为。例如：

- 数据源 (dataSource)

dataSource 元素使用标准的 JDBC 数据源接口来配置 JDBC 连接对象的资源。有三种内建的数据源类型 (也就是 type="[UNPOOLED|POOLED|JNDI]")

- 可以通过实现接口 org.apache.ibatis.datasource.DataSourceFactory 来使用第三方数据源实现:

```
public interface DataSourceFactory {  
    void setProperties(Properties props);  
    DataSource getDataSource();  
}
```

org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory 可被用作父类来构建新的数据源适配器, 比如下面这段插入 C3P0 数据源所必需的代码:

```
import  
org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory;  
import com.mchange.v2.c3p0.ComboPooledDataSource;  
public class C3P0DataSourceFactory extends  
    UnpooledDataSourceFactory {  
    public C3P0DataSourceFactory() {  
        this.dataSource = new ComboPooledDataSource();  
    }  
}
```

为了令其工作, 记得在配置文件中为每个希望 MyBatis 调用的 setter 方法增加对应的属性。下面是一个可以连接至 PostgreSQL 数据库的例子:

- 数据库厂商标识 (databaseIdProvider)

- 映射器 (mappers)

可以使用相对于类路径的资源引用, 或完全限定资源定位符 (包括 file:/// 形式的 URL), 或类名和包名等配置映射文件

- 

- XML映射(Mapper)

MyBatis 的真正强大在于它的语句映射

- SQL 映射文件只有很少的几个顶级元素（按照应被定义的顺序列出）：
  - #cache – 该命名空间的缓存配置。
  - #cache-ref – 引用其它命名空间的缓存配置。
  - #resultMap – 描述如何从数据库结果集中加载对象，是最复杂也是最强大的元素。
  - #parameterMap – 老式风格的参数映射。此元素已被废弃，并可能在将来被移除！请使用行内参数映射。文档中不会介绍此元素。
  - #sql – 可被其它语句引用的可重用语句块。
  - #insert – 映射插入语句。
  - #update – 映射更新语句。
  - #delete – 映射删除语句。
  - #select – 映射查询语句。

- select:

select 元素允许你配置很多属性来配置每条语句的行为细节。

```
<select
  id="selectPerson"
  parameterType="int"
  parameterMap="deprecated"
  resultType="hashmap"
  resultMap="personResultMap"
  flushCache="false"
  useCache="true"
  timeout="10"
  fetchSize="256"
  statementType="PREPARED"
  resultSetType="FORWARD_ONLY">
```

- 参数符号：#{id} MyBatis 会创建一个预处理语句（PreparedStatement）参数，在 JDBC 中，这样的参数在 SQL 中会由一个“?”来标识，并被传递到一个新的预处理语句中

- insert, update 和 delete

数据变更语句 insert, update 和 delete 的实现非常接近：

- sql

这个元素可以用来定义可重用的 SQL 代码片段，以便在其它语句中使用。参数可以静态地（在加载的时候）确定下来，并且可以在不同的 include 元素中定义不同的参数值。比如：

```
${alias}.id,${alias}.username,${alias}.password
```

这个 SQL 片段可以在其它语句中使用，例如：

```
<select id="selectUsers" resultType="map">
```

```
  select
```

```
,
```

```
  from some_table t1
```

```
  cross join some_table t2
```

- 字符串替换(\${})

默认情况下，使用 `#{} 参数语法` 时，MyBatis 会创建 `PreparedStatement` 参数占位符，并通过占位符安全地设置参数（就像使用 `?` 一样）。这样做更安全，更迅速，通常也是首选做法，不过有时你就是想直接在 SQL 语句中直接插入一个不转义的字符串。比如 `ORDER BY` 子句，这时候你可以：

```
ORDER BY ${columnName}
```

- 结果映射

`resultMap` 元素是 MyBatis 中最重要最强大的元素。它可以让你从 90% 的 JDBC `ResultSets` 数据提取代码中解放出来，并在一些情形下允许你进行一些 JDBC 不支持的操作。实际上，在为一些比如连接的复杂语句编写映射代码的时候，一份 `resultMap` 能够代替实现同等功能的数千行代码。`ResultMap` 的设计思想是，对简单的语句做到零配置，对于复杂一点的语句，只需要描述语句之间的关系就行了。

- 示例:

- 结果映射（resultMap）

- `#constructor` - 用于在实例化类时，注入结果到构造方法中

- `##idArg` - ID 参数；标记出作为 ID 的结果可以帮助提高整体性能

- `##arg` - 将被注入到构造方法的一个普通结果

- `#id` - 一个 ID 结果；标记出作为 ID 的结果可以帮助提高整体性能

- `#result` - 注入到字段或 JavaBean 属性的普通结果

- `#association` - 一个复杂类型的关联；许多结果将包装成这种类型

- 嵌套结果映射 - 关联可以是 `resultMap` 元素，或是对其它结果映射的引用

- `#collection` - 一个复杂类型的集合

- 嵌套结果映射 - 集合可以是 `resultMap` 元素，或是对其它结果映射的引用

- `#discriminator` - 使用结果值来决定使用哪个 `resultMap`

- `##case` - 基于某些值的结果映射

- 嵌套结果映射 - `case` 也是一个结果映射，因此具有相同的结构和元素；或者引用其它的结果映射

-

- 关联

MyBatis 有两种不同的方式加载关联

- 嵌套 Select 查询：通过执行另外一个 SQL 映射语句来加载期望的复杂类型。
- 嵌套结果映射：使用嵌套的结果映射来处理连接结果的重复子集。

- 集合

集合元素和关联元素几乎是一样的，

- 鉴别器

有时候，一个数据库查询可能会返回多个不同的结果集（但总体上还是有一定的联系的）。鉴别器（discriminator）元素就是被设计来应对这种情况的，另外也能处理其它情况，例如类的继承层次结构。鉴别器的概念很好理解——它很像 Java 语言中的 switch 语句。

- 示例:

## ■ 缓存

MyBatis 内置了一个强大的事务性查询缓存机制，它可以非常方便地配置和定制。为了使它更加强大而且易于配置，我们对 MyBatis 3 中的缓存实现进行了许多改进。

默认情况下，只启用了本地的会话缓存，它仅仅对一个会话中的数据进行缓存。要启用全局的二级缓存，只需要在你的 SQL 映射文件中添加一行：

- 基本上就是这样。这个简单语句的效果如下：
  - #映射语句文件中的所有 select 语句的结果将会被缓存。
  - #映射语句文件中的所有 insert、update 和 delete 语句会刷新缓存。
  - #缓存会使用最近最少使用算法（LRU, Least Recently Used）算法来清除不需要的缓存。
  - #缓存不会定时进行刷新（也就是说，没有刷新闻隔）。
  - #缓存会保存列表或对象（无论查询方法返回哪种）的 1024 个引用。
  - #缓存会被视为读/写缓存，这意味着获取到的对象并不是共享的，可以安全地被调用者修改，而不干扰其他调用者或线程所做的潜在修改。
  - #提示 缓存只作用于 cache 标签所在的映射文件中的语句。如果你混合使用 Java API 和 XML 映射文件，在共用接口中的语句将不会被默认缓存。你需要使用 @CacheNamespaceRef 注解指定缓存作用域。
- 这些属性可以通过 cache 元素的属性来修改。比如：

这个更高级的配置创建了一个 FIFO 缓存，每隔 60 秒刷新，最多可以存储结果对象或列表的 512 个引用，而且返回的对象被认为是只读的，因此对它们进行修改可能会在不同线程中的调用者产生冲突。

- 可用的清除策略有：
  - #LRU – 最近最少使用：移除最长时间不被使用的对象。
  - #FIFO – 先进先出：按对象进入缓存的顺序来移除它们。
  - #SOFT – 软引用：基于垃圾回收器状态和软引用规则移除对象。
  - #WEAK – 弱引用：更积极地基于垃圾收集器状态和弱引用规则移除对象。默认的清除策略是 LRU。
- flushInterval（刷新闻隔）属性可以被设置为任意的正整数，设置的值应该是一个以毫秒为单位的合理时间量。默认情况是不设置，也就是没有刷新闻隔，缓存仅仅会在调用语句时刷新。
  - size（引用数目）属性可以被设置为任意正整数，要注意欲缓存对象的大小和运行环境中可用的内存资源。默认值是 1024。
  - readOnly（只读）属性可以被设置为 true 或 false。只读的缓存会给所有调用者返回缓存对象的相同实例。因此这些对象不能被修改。这就提供了可观的性能提升。而可读写的缓存会（通过序列化）返回缓存对象的拷贝。速度上会慢一些，但是更安全，因此默认值是 false。

提示 二级缓存是事务性的。这意味着，当 SqlSession 完成并提交时，或是完成并回滚，但没有执行 flushCache=true 的 insert/delete/update 语句时，缓存会获得更新。

- 使用自定义缓存

除了上述自定义缓存的方式，你也可以通过实现你自己的缓存，或为其他第三方缓存方案创建适配器，来完全覆盖缓存行为。

- 这个示例展示了如何使用一个自定义的缓存实现。type 属性指定的类必须实现 org.apache.ibatis.cache.Cache 接口，且提供一个接受 String 参数作为 id 的构造器。这个接口是 MyBatis 框架中许多复杂的接口之一，但是行为却非常简单。

```
public interface Cache {  
    String getId();  
    int getSize();  
    void putObject(Object key, Object value);  
    Object getObject(Object key);  
    boolean hasKey(Object key);  
    Object removeObject(Object key);  
    void clear();  
}
```

为了对你的缓存进行配置，只需要简单地在你的缓存实现中添加公有的 JavaBean 属性，然后通过 cache 元素传递属性值，例如，下面的例子将在你的缓存实现上调用一个名为 setCacheFile(String file) 的方法：

可以使用占位符（如 \${cache.file}）

上一节中对缓存的配置（如清除策略、可读或可读写等），不能应用于自定义缓存。

- 延迟加载

Mybatis 仅支持 association 关联对象和 collection 关联集合对象的延迟加载，association

指的就是一对一，collection 指的就是一对多查询。在 Mybatis 配置文件中，可以配置是否

启用延迟加载 lazyLoadingEnabled=true|false,默认false;

- 原理:

使用 CGLIB 创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用 a.getB().getName()，拦截器 invoke()方法发现 a.getB()是 null 值，那么就会单

独发送事先保存好的查询关联 B 对象的 sql，把 B 查询上来，然后调用 a.setB(b)，于是 a 的

对象 b 属性就有值了，接着完成 a.getB().getName()方法的调用。这就是延迟加载的基本原

理。

- 动态 SQL

动态 SQL 是 MyBatis 的强大特性之一。



- 借助功能强大的基于 OGNL 的表达式,有以下表达式

if

choose (when, otherwise)

trim (where, set)

foreach

script

bind

- if:

使用动态 SQL 最常见情景是根据条件包含 where 子句的一部分。

```
<select id="findActiveBlogWithTitleLike"
```

```
  resultType="Blog">
```

```
  SELECT * FROM BLOG
```

```
  WHERE state = 'ACTIVE'
```

```
  AND title like #{title}
```

- choose、when、otherwise:

有时候, 我们不想使用所有的条件, 而只是想从多个条件中选择一个使用。针对这种情况, MyBatis 提供了 choose 元素, 像 Java 中的 switch 语句

- <select id="findActiveBlogLike"

```
  resultType="Blog">
```

```
  SELECT * FROM BLOG WHERE state = 'ACTIVE'
```

```
  AND title like #{title}
```

```
  AND author_name like #{author.name}
```

```
  AND featured = 1
```

- trim、where、set

- where 元素只会在子元素返回任何内容的情况下才插入 "WHERE" 子句。而且, 若子句的开头为 "AND" 或 "OR", where 元素也会将它们去除;

```
<select id="findActiveBlogLike"
```

```
  resultType="Blog">
```

```
  SELECT * FROM BLOG
```

```
  state = #{state}
```

```
  AND title like #{title}
```

AND author\_name like #{author.name}

如果 where 元素与你期望的不太一样，你也可以通过自定义 trim 元素来定制 where 元素的功能。比如，和 where 元素等价的自定义 trim 元素为：

...

prefixOverrides 属性会忽略通过管道符分隔的文本序列（注意此例中的空格是必要的）。上述例子会移除所有 prefixOverrides 属性中指定的内容，并且插入 prefix 属性中指定的内容。

- 用于动态更新语句的类似解决方案叫做 set。set 元素可以用于动态包含需要更新的列，忽略其它不更新的列。比如：

update Author

```
username=#{username},
password=#{password},
email=#{email},
bio=#{bio}
```

where id=#{id}

这个例子中，set 元素会动态地在行首插入 SET 关键字，并会删掉额外的逗号（这些逗号是在使用条件语句给列赋值时引入的）。

来看看与 set 元素等价的自定义 trim 元素吧：

...

- foreach

动态 SQL 的另一个常见使用场景是对集合进行遍历（尤其是在构建 IN 条件语句的时候）

- ```
<select id="selectPostIn" resultType="domain.blog.Post">
  SELECT *
  FROM POST P
  WHERE ID in

  #{item}
```

foreach 元素的功能非常强大，它允许你指定一个集合，声明可以在元素体内使用的集合项（item）和索引（index）变量。它 also 允许你指定开头与结尾的字符串以及集合项迭代之间的分隔符。

你可以将任何可迭代对象（如 List、Set 等）、Map 对象或者数组对象作为集合参数传递给 foreach。当使用可迭代对象或者数组时，index 是当前迭代的序号，item 的值是本次迭代获取到的元素。当使用 Map 对象（或者 Map.Entry 对象的集合）时，index 是键，item 是值。

- script

要在带注解的映射器接口类中使用动态 SQL，可以使用 script 元素。比如：

```
@Update({""})
```

```
void updateAuthorValues(Author author);
```

- bind

bind 元素允许你在 OGNL 表达式以外创建一个变量，并将其绑定到当前的上下文。比如：

```
<select id="selectBlogsLike" resultType="Blog">
```

```
SELECT * FROM BLOG
```

```
WHERE title LIKE #{pattern}
```

- 多数据库支持

如果配置了 databaseIdProvider，你就可以在动态代码中使用名为

“databaseId”的变量来为不同的数据库构建特定的语句。比如下面的例子：

```
<selectKey keyProperty="id" resultType="int" order="BEFORE">
```

```
<if test="databaseId == 'oracle'">
```

```
select seq_users.nextval from dual
```

```
select nextval for seq_users from sysibm.sysdummy1"
```

```
insert into users values (#{id}, #{name})
```

- JavaAPI

- SqlSession

使用 MyBatis 的主要 Java 接口就是 SqlSession。你可以通过这个接口来执行命令，获取映射器示例和管理事务。在介绍 SqlSession 接口之前，我们先来了解如何获取一个 SqlSession 实例。SqlSessions 是由 SqlSessionFactory 实例创建的。SqlSessionFactory 对象包含创建 SqlSession 实例的各种方法。而 SqlSessionFactory 本身是由 SqlSessionFactoryBuilder 创建的，它可以从 XML、注解或 Java 配置代码来创建 SqlSessionFactory。

包含了所有执行语句、提交或回滚事务以及获取映射器实例的方法。

- 语句执行方法：

这些方法被用来执行定义在 SQL 映射 XML 文件中的 SELECT、INSERT、UPDATE 和 DELETE 语句。你可以通过名字快速了解它们的作用，每一方法都接受语句的 ID 以及参数对象，参数可以是原始类型（支持自动装箱或包装类）、JavaBean、POJO 或 Map。

```
T selectOne(String statement, Object parameter)
```

```
List selectList(String statement, Object parameter)
```

```
Cursor selectCursor(String statement, Object parameter)
```

```
<K,V> Map<K,V> selectMap(String statement, Object parameter, String  
mapKey)
```

```
int insert(String statement, Object parameter)
```

```
int update(String statement, Object parameter)
```

```
int delete(String statement, Object parameter)
```

由于并不是所有语句都需要参数，所以这些方法都具有一个不需要参数的重载形式。

- selectOne 和 selectList 的不同仅仅是 selectOne 必须返回一个对象或 null 值。如果返回值多于一个，就会抛出异常。如果你不知道返回对象会有多少，请使用 selectList。如果需要查看某个对象是否存在，最好的办法是查询一个 count 值（0 或 1）。selectMap 稍微特殊一点，它会将返回对象的其中一个属性作为 key 值，将对象作为 value 值，从而将多个结果集转为 Map 类型值。由于并不是所有语句都需要参数，所以这些方法都具有一个不需要参数的重载形式。

- 游标（Cursor）与列表（List）返回的结果相同，不同的是，游标借助迭代器实现了数据的惰性加载。

```
try (Cursor entities = session.selectCursor(statement, param)) {  
    for (MyEntity entity:entities) {  
        // 处理单个实体  
    }  
}
```

- insert、update 以及 delete 方法返回的值表示受该语句影响的行数。

```
T selectOne(String statement)  
List selectList(String statement)  
Cursor selectCursor(String statement)  
<K,V> Map<K,V> selectMap(String statement, String mapKey)  
int insert(String statement)  
int update(String statement)  
int delete(String statement)
```

- 立即批量更新方法

当你将 ExecutorType 设置为 ExecutorType.BATCH 时，可以使用这个方法清除（执行）缓存在 JDBC 驱动类中的批量更新语句。

```
List flushStatements()
```

- 事务控制方法

有四个方法用来控制事务作用域。当然，如果你已经设置了自动提交或你使用了外部事务管理器，这些方法就没什么作用了。然而，如果你正在使用由 Connection 实例控制的 JDBC 事务管理器，那么这四个方法就会派上用场：

```
void commit()  
void commit(boolean force)  
void rollback()  
void rollback(boolean force)
```

默认情况下 MyBatis 不会自动提交事务，除非它检测到调用了插入、更新或删除方法改变了数据库。如果你没有使用这些方法提交修改，那么你可以在 commit 和 rollback 方法参数中传入 true 值，来保证事务被正常提交（注意，在自动提交模式或者使用了外部事务管理器的情况下，设置 force 值对 session 无效）。大部分情况下你无需调用 rollback()，因为 MyBatis 会在你

没有调用 commit 时替你完成回滚操作。不过，当你要在一个可能多次提交或回滚的 session 中详细控制事务，回滚操作就派上用场了。

- 本地缓存

Mybatis 使用到了两种缓存：本地缓存（local cache）和二级缓存（second level cache）。

- 每当一个新 session 被创建，MyBatis 就会创建一个与之相关联的本地缓存。任何在 session 执行过的查询结果都会被保存在本地缓存中，所以，当再次执行参数相同的相同查询时，就不需要实际查询数据库了。本地缓存将会在做出修改、事务提交或回滚，以及关闭 session 时清空。

默认情况下，本地缓存数据的生命周期等同于整个 session 的周期。由于缓存会被用来解决循环引用问题和加快重复嵌套查询的速度，所以无法将其完全禁用。但是你可以通过设置 localCacheScope=STATEMENT 来只在语句执行时使用缓存。

注意，如果 localCacheScope 被设置为 SESSION，对于某个对象，MyBatis 将返回在本地缓存中唯一对象的引用。对返回的对象（例如 list）做出的任何修改将会影响本地缓存的内容，进而将会影响到在本次 session 中从缓存返回的值。因此，不要对 MyBatis 所返回的对象作出更改，以防后患。

你可以随时调用以下方法来清空本地缓存：

```
void clearCache()
```

- 确保 SqlSession 被关闭

```
void close()
```

- 使用映射器

```
T getMapper(Class type)
```

- 一个映射器类就是一个仅需声明与 SqlSession 方法相匹配方法的接口。下面的示例展示了一些方法签名以及它们是如何映射到 SqlSession 上的。

```
public interface AuthorMapper {  
    // (Author) selectOne("selectAuthor",5);  
    Author selectAuthor(int id);  
    // (List) selectList("selectAuthors")  
    List selectAuthors();  
    // (Map<Integer,Author>) selectMap("selectAuthors", "id")  
    @MapKey("id")  
    Map<Integer, Author> selectAuthors();  
    // insert("insertAuthor", author)  
    int insertAuthor(Author author);  
    // updateAuthor("updateAuthor", author)  
    int updateAuthor(Author author);  
    // delete("deleteAuthor",5)  
    int deleteAuthor(int id);  
}
```

- 映射器注解

- @Insert

- @Select

- @Update

- 等

- 映射注解示例

- 这个例子展示了如何使用 @SelectKey 注解来在插入前读取数据库序列的值：

- @Insert("insert into table3 (id, name) values(#{nameId}, #{name})")

- @SelectKey(statement="call next value for TestSequence",

- keyProperty="nameId", before=true, resultType=int.class)

- int insertTable3(Name name);

- SqlSessionFactoryBuilder

- SqlSessionFactoryBuilder 有五个 build() 方法，每一种都允许你从不同的资源中创建一个 SqlSessionFactory 实例。

- SqlSessionFactory build(InputStream inputStream)

- SqlSessionFactory build(InputStream inputStream, String environment)

- SqlSessionFactory build(InputStream inputStream, Properties properties)

- SqlSessionFactory build(InputStream inputStream, String env, Properties props)

- SqlSessionFactory build(Configuration config)

- 第一种方法是最常用的，它接受一个指向 XML 文件（也就是之前讨论的 mybatis-config.xml 文件）的 InputStream 实例。可选的参数是 environment 和 properties。environment 决定加载哪种环境，包括数据源和事务管理器。比如：

...

...

...

...

如果你调用了带 environment 参数的 build 方法，那么 MyBatis 将使用该环境对应的配置。当然，如果你指定了一个无效的环境，会收到错误。如果你调用了不带 environment 参数的 build 方法，那么就会使用默认的环境配置（在上面的示例中，通过 default="development" 指定了默认环境）。

- 总结一下，前四个方法很大程度上是相同的，但提供了不同的覆盖选项，允许你可选地指定 environment 和/或 properties。以下给出一个从 mybatis-config.xml 文件创建 SqlSessionFactory 的示例：

```
String resource = "org/mybatis/builder/mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(inputStream);
```

- 最后一个 build 方法接受一个 Configuration 实例。Configuration 类包含了对一个 SqlSessionFactory 实例你可能关心的所有内容。在检查配置时，Configuration 类很有用，它允许你查找和操纵 SQL 映射（但当应用开始接收请求时不推荐使用）。你之前学习过的所有配置开关都存在于 Configuration 类，只不过它们是以 Java API 形式暴露的。以下是一个简单的示例，演示如何手动配置 Configuration 实例，然后将它传递给 build() 方法来创建 SqlSessionFactory。

```
DataSource dataSource = BaseDataTest.createBlogDataSource();
TransactionFactory transactionFactory = new JdbcTransactionFactory();
Environment environment = new Environment("development",
transactionFactory, dataSource);
Configuration configuration = new Configuration(environment);
configuration.setLazyLoadingEnabled(true);
configuration.setEnhancementEnabled(true);
configuration.getTypeAliasRegistry().registerAlias(Blog.class);
configuration.getTypeAliasRegistry().registerAlias(Post.class);
configuration.getTypeAliasRegistry().registerAlias(Author.class);
configuration.addMapper(BoundBlogMapper.class);
configuration.addMapper(BoundAuthorMapper.class);
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(configuration);
```

现在你就获得一个可以用来创建 SqlSession 实例的 SqlSessionFactory 了。

- SqlSessionFactory

SqlSessionFactory 有六个方法创建 SqlSession 实例。通常来说，当你选择其中一个方法时，你需要考虑以下几点：

#事务处理：你希望在 session 作用域中使用事务作用域，还是使用自动提交

（auto-commit）？（对很多数据库和/或 JDBC 驱动来说，等同于关闭事务支持）

#数据库连接：你希望 MyBatis 帮你从已配置的数据源获取连接，还是使用自己提供的连接？

#语句执行：你希望 MyBatis 复用 PreparedStatement 和/或批量更新语句（包括插入语句和删除语句）吗？

- 基于以上需求，有下列已重载的多个 openSession() 方法供使用。

```
SqlSession openSession()
SqlSession openSession(boolean autoCommit)
SqlSession openSession(Connection connection)
SqlSession openSession(TransactionIsolationLevel level)
SqlSession openSession(ExecutorType execType,
TransactionIsolationLevel level)
SqlSession openSession(ExecutorType execType)
SqlSession openSession(ExecutorType execType, boolean autoCommit)
SqlSession openSession(ExecutorType execType, Connection connection)
Configuration getConfiguration();
```

- 默认的 `openSession()` 方法没有参数，它会创建具备如下特性的 `SqlSession`：
  - #事务作用域将会开启（也就是不自动提交）。
  - #将由当前环境配置的 `DataSource` 实例中获取 `Connection` 对象。
  - #事务隔离级别将会使用驱动或数据源的默认设置。
  - #预处理语句不会被复用，也不会批量处理更新。
 相信你已经能从方法签名中知道这些方法的区别。向 `autoCommit` 可选参数传递 `true` 值即可开启自动提交功能。若要使用自己的 `Connection` 实例，传递一个 `Connection` 实例给 `connection` 参数即可。注意，我们没有提供同时设置 `Connection` 和 `autoCommit` 的方法，这是因为 MyBatis 会依据传入的 `Connection` 来决定是否启用 `autoCommit`。对于事务隔离级别，MyBatis 使用了一个 Java 枚举包装器来表示，称为 `TransactionIsolationLevel`，事务隔离级别支持 JDBC 的五个隔离级别（`NONE`、`READ_UNCOMMITTED`、`READ_COMMITTED`、`REPEATABLE_READ` 和 `SERIALIZABLE`），并且与预期的行为一致。
- `ExecutorType`:
  - `ExecutorType.SIMPLE`：该类型的执行器没有特别的行为。它为每个语句的执行创建一个新的预处理语句。
  - `ExecutorType.REUSE`：该类型的执行器会复用预处理语句。
  - `ExecutorType.BATCH`：该类型的执行器会批量执行所有更新语句，如果 `SELECT` 在多个更新中间执行，将在必要时将多条更新语句分隔开来，以方便理解。

- SQL 语句构建器

<https://mybatis.org/mybatis-3/zh/statement-builders.html>

- `SqlSession` 执行流程

- - 通过 `SqlSessionFactoryBuilder.build()` 加载配置文件或调用 JAVA API 创建 `SqlSessionFactory(org.apache.ibatis.session.defaults.DefaultSqlSessionFactory)`
  - 2. 通过 `SqlSessionFactory.openSession()` 获取 `SqlSession` 对象 (`DefaultSqlSession`);
    - 可通过数据源获取 `SqlSession` 或指定的连接获取;
    - 获取连接时调用 `openSessionFromDataSource()` 方法;
- `openSessionFromDataSource()` 中
  - 通过 `Environment` 获取 `TransactionFactory`, `newTransaction` 创建新的事务,
  - 然后通过事务和 `executorType` 创建不同 `Executor` (`ExecutorType.BATCH`: `BatchExecutor`, `ExecutorType.REUSE`: `ReuseExecutor`, 其余的返回 `SimpleExecutor`)
  - 若开启了二级缓存 (`cacheEnabled=true`, 默认开启), 则创建 `new CachingExecutor((Executor)executor)`; 一级缓存在 `BaseExecutor.query()` 中使用 `PerpetualCache` 实现
  - 创建 `executor` 的 `InterceptorChain` (拦截器链) 的代理对象, 并返回; 代理对象使用 `Plugin.wrap(target, this)` 创建 (基于接口的 Java 动态代理), 代理对象 `invoke` 执行时, 会从 `@Intercep` 注解中获取需要执行拦截器方法的对象;
- 4. 根据 `Executor` 创建 `new DefaultSqlSession(this.configuration, executor, autoCommit);`
- 根据 `DefaultSqlSession` 获取 `getMapper()`, 生成 `Mapper` 接口的代理类; 从 `configuration`



中取得 Mapper，最终调用了 MapperProxyFactory 的 newInstance，其中关键的拦截逻辑在 MapperProxy 中，其 invoke 方法最后由 mapperMethod.execute 执行，执行方法时，最终调用了 sqlSession 对应的 insert/update/delete/select 等方法

- MyBatis 的 Executor 常用的有以下几种：

#SimpleExecutor: 默认的 Executor，每个 SQL 执行时都会创建新的 Statement

#ReuseExecutor: 相同的 SQL 会复用 Statement

#BatchExecutor: 用于批处理的 Executor

#CachingExecutor: 可缓存数据的 Executor，用代理模式包装了其它类型的 Executor；这些 Executor 都继承了 (除 CachingExecutor) BaseExecutor，BaseExecutor 提供了缓存管理和事务管理的基本功能

- select(DefaultSqlSession 中 select()) 执行：

1. 获取 MappedStatement ms = this.configuration.getMappedStatement(statement);  
首先检查缓存中是否有数据：

a. 如果有，则先根据需要处理是否清除缓存，然后若使用缓存，且缓存不为 null 时，直接返回缓存；若缓存为 null，则执行一次代理查询，查询完成后，存入缓存；

b. 如果没有缓存，则再调用代理对象的 query，默认是 SimpleExecutor。Executor 是一个接口，下面有个实现类是 BaseExecutor，其中实现了其它 Executor 通用的一些逻辑，包括 doQuery 以及 doUpdate 等，其中封装了 JDBC 的相关操作；query 中先根据参数处理 SQL，然后执行查询数据库查询 doQuery

c. SimpleExecutor.doQuery() 中先从 connection 获取并配置 preparedStatement，然后执行 jdbc 查询，然后将返回结果由 ResultHandler 处理，查询时会经过一级缓存 PerpetualCache 的处理

- update 执行：

update 的执行与 select 类似，都是从 CachingExecutor 开始：

@Override

```
public int update(MappedStatement ms, Object parameterObject) throws  
SQLException {  
    // 检查是否需要刷新缓存  
    flushCacheIfRequired(ms);  
    // 调用代理类的 update  
    return delegate.update(ms, parameterObject);  
}
```

update 会使得缓存的失效，第一步是检查是否需要刷新缓存，接下来再交给代理类去执行真正的数据库更新操作。

## ■ 缓存

Mybatis 中进行 SQL 查询是通过 org.apache.ibatis.executor.Executor 接口进行的，总体来讲，它一共有两类实现，一类是 BaseExecutor，一类是 CachingExecutor。前者是非启用二级缓存时使用的，而后者是采用的装饰器模式，在启用了二级缓存时使用，当二级缓存没有命中时，底层还是通过 BaseExecutor 来实现的。

一，二级缓存底层都是使用的 HashMap 实现

### ■ 一级缓存：

SqlSession 级别缓存，默认开启，且不能关闭；

在同一个 sqlSession 执行同一个 SQL 时，第二次会从一级缓存中获取，缓存不存在才会走数据库查询；

一级缓存默认 1024 条 SQL；

一级缓存是默认启用的，在 BaseExecutor 的 query() 方法中实现，底层默认使用的

是PerpetualCache实现，PerpetualCache采用HashMap存储数据。一级缓存会在进行增、删、改操作时进行清除。

- 一级缓存的范围有SESSION和STATEMENT两种，默认是SESSION，如果我们不需要使用一级缓存，那么我们可以把一级缓存的范围指定为STATEMENT，这样每次执行完一个Mapper语句后都会将一级缓存清除。如果只是需要对某一条select语句禁用一级缓存，则可以在对应的select元素上加上flushCache="true"。如果需要更改一级缓存的范围，请在Mybatis的配置文件中，在下通过localCacheScope指定。

- 二级缓存:  
二级缓存可以跨SqlSession,默认开启(cacheEnabled),且必须在Mapper.xml中添加标签才能让二级缓存生效;  
二级缓存在CachingExecutor中实现;  
可在settings配置中关闭二级缓存

默认情况下Mapper中所有select语句的useCache都是true，如果我们在启用了二级缓存后，有某个查询语句是我们不想缓存的，则可以通过指定其useCache为false来达到对应的效果

- 对于同一个Mapper来讲，它只能使用一个Cache，当同时使用了和时使用定义的优先级更高。
- 缓存清除:  
二级缓存默认是会在执行update、insert和delete语句时进行清空的，具体可以参考CachingExecutor的update()实现。如果我们不希望在执行某一条更新语句时清空对应的二级缓存，那么我们可以在对应的语句上指定flushCache属性等于false。如果只是某一条select语句不希望使用二级缓存和一级缓存，则也可以在对应的select元素上加上flushCache="true"。
- 四大组件  
org.apache.ibatis.session.Configuration中创建组件实例
  - Executor
    - 通过SqlSessionFactory创建SqlSession时,会根据ExecutorType创建对应类型的Executor,创建Executor后会基于InterceptorChain创建拦截器代理对象  
public Executor newExecutor(Transaction transaction, ExecutorType executorType) {  
...  
Executor executor = (Executor)this.interceptorChain.pluginAll(executor);  
return executor;
  - StatementHandler
    - BaseStatementHandler:  
执行SimpleExecutor.doQuery()时会创建statementHandler,创建newStatementHandler时,会在RoutingStatementHandler的基础上基于InterceptorChain创建拦截器的代理对象  
public StatementHandler newStatementHandler(Executor executor,

```

MappedStatement mappedStatement, Object parameterObject,
RowBounds rowBounds, ResultHandler resultHandler, BoundSql
boundSql) {
    StatementHandler statementHandler = new
RoutingStatementHandler(executor, mappedStatement,
parameterObject, rowBounds, resultHandler, boundSql);
    StatementHandler statementHandler =
(statementHandler)this.interceptorChain.pluginAll(statementHandler);
    return statementHandler;
}

```

- ParameterHandler

- 创建新的newParameterHandler时会 InterceptorChain创建拦截器的代理对象

```

public ParameterHandler newParameterHandler(MappedStatement
mappedStatement, Object parameterObject, BoundSql boundSql) {
    ParameterHandler parameterHandler =
mappedStatement.getLang().createParameterHandler(mappedStatement
, parameterObject, boundSql);
    parameterHandler =
(ParameterHandler)this.interceptorChain.pluginAll(parameterHandler);
    return parameterHandler;
}

```

- ResultSetHandler

- 创建新的newResultSetHandler时会 InterceptorChain创建拦截器的代理对象

```

public ResultSetHandler newResultSetHandler(Executor executor,
MappedStatement mappedStatement, RowBounds rowBounds,
ParameterHandler parameterHandler, ResultHandler resultHandler,
BoundSql boundSql) {
    ResultSetHandler resultSetHandler = new
DefaultResultSetHandler(executor, mappedStatement,
parameterHandler, resultHandler, boundSql, rowBounds);
    ResultSetHandler resultSetHandler =
(ResultSetHandler)this.interceptorChain.pluginAll(resultSetHandler);
    return resultSetHandler;
}

```

- 自定义插件

- 实现Interceptor,制定拦截的方法签名,并注册到plugins中

- 注解解析

- @Sql
- @Insert
- @ResultMaps

- findField方式查询

- 优点

- MyBatis 把 sql 语句从 Java 源程序中独立出来, 放在单独的 XML 文件中编写, 给

程序的

维护带来了很大便利。

- MyBatis 封装了底层 JDBC API 的调用细节，并能自动将结果集转换成 Java Bean 对象，大大简化了 Java 数据库编程的重复工作
- 因为 MyBatis 需要程序员自己去编写 sql 语句，程序员可以结合数据库自身的特点灵活控制 sql 语句，因此能够实现比 Hibernate 等全自动 orm 框架更高的查询效率，能够完成复杂查询。
- 知识点
  - Mybatis 的 Xml 映射文件和 Mybatis 内部数据结构之间的映射关系：Mybatis 将所有 Xml 配置信息都封装到 All-In-One 重量级对象 Configuration 内部。在 Xml 映射文件中，标签会被解析为 ParameterMap 对象，其每个子元素会被解析为 ParameterMapping 对象。标签会被解析为 ResultMap 对象，其每个子元素会被解析为 ResultMapping 对象。每一个<select>、标签均会被解析为 MappedStatement 对象，标签内的 sql 会被解析为 BoundSql 对象。
  - MyBatis 的接口绑定的好处：接口映射就是在 MyBatis 中任意定义接口,然后把接口里面的方法和 SQL 语句绑定,我们直接调用接口方法就可以了
  - 接口绑定有2种实现方式:
- 注解,在接口的方法上面加上 @Select@Update 等注解里面包含 Sql 语句来绑定
- 2.XML,编写 SQL 来绑定,在这种情况下,要指定 xml 映射文件里面的 namespace 必须为接口的完整限定名。
  - MyBatis 实现一对一的2种查询方式:
- 表关联查询,1条SQL内,表关联进行查询,结果集中通过association 节点配置一对一的类;2.;嵌套查询, resultMap中配置association,并关联其他select标签查询的SQL
  - Mybatis 执行一对一、一对多的关联查询:
- 使用selectList进行查询
- 关联对象查询有2种方式,一种为1条SQL内做表关联查询,在结果集中配置关联对象;另一种为使用嵌套查询,在resultMap中指定嵌套标签并指对应select标签语句
  - Mybatis 是如何将 sql 执行结果封装为目标对象并返回的? Mybatis执行结果封装对象有2种方式:
- resultMap配置对象，字段映射关系
- 使用SQL字段别名(as),指定为对象属性,Mybatis通过反射注入对象中
  - Xml 映射文件中，除了常见的 select|insert|update|delete 标签之外，还有、、、、<selectKey>，加上动态 sql 的 9 个标签，trim|where|set|foreach|if|choose|when|otherwise|bind 等，其中为 sql 片段标签，通

过标签引入 sql 片段, <selectKey>为不支持自增的主键生成策略标签。

- 通常一个 Xml 映射文件, 都会写一个 Dao 接口与之对应, Dao 的工作原理, 是否可以重载?

不能重载, 因为通过 Dao 寻找 Xml 对应的 sql 的时候全限定名+方法名的保存和寻找策略。

接口工作原理为 jdk 动态代理原理, 运行时会为 dao 生成 proxy, 代理对象会拦截接口

方法, 去执行对应的 sql 返回数据。

- Mybatis 映射文件中, 如果 A 标签通过 include 引用了 B 标签的内容, 请问, B 标签能否定义在 A 标签的后面?

虽然 Mybatis 解析 Xml 映射文件是按照顺序解析的, 但是, 被引用的 B 标签依然可以

定义在任何地方, Mybatis 都可以正确识别。原理是, Mybatis 解析 A 标签, 发现 A 标签引

用了 B 标签, 但是 B 标签尚未解析到, 尚不存在, 此时, Mybatis 会将 A 标签标记为未解

析状态, 然后继续解析余下的标签, 包含 B 标签, 待所有标签解析完毕, Mybatis 会重新

解析那些被标记为未解析的标签, 此时再解析 A 标签时, B 标签已经存在, A 标签也就可

以正常解析完成了。

- Mybatis 的 Xml 映射文件中, 不同的 Xml 映射文件, id 是否可以重复?  
不同的 Xml 映射文件, 如果配置了 namespace, 那么 id 可以重复; 如果没有配置 namespace, 那么 id 不能重复; 毕竟 namespace 不是必须的, 只是最佳实践而已。原因就

是 namespace+id 是作为 Map<String, MappedStatement>的 key 使用的, 如果没有

namespace, 就剩下 id, 那么, id 重复会导致数据互相覆盖。

- Mybatis 三种基本的 Executor 执行器:

SimpleExecutor、ReuseExecutor、BatchExecutor。

1.SimpleExecutor: 每执行一次 update 或 select, 就开启一个 Statement 对象, 用完立刻关闭 Statement 对象。

2.ReuseExecutor: 执行 update 或 select, 以 sql 作为 key 查找 Statement 对象, 存在就使用, 不存在就创建, 用完后, 不关闭 Statement 对象, 而是放置于 Map

3.BatchExecutor: 完成批处理。

还有一个二级缓存CachingExecutor,使用装饰模式代理3中基本exector,增加缓存处理

- Mybatis 中如何指定使用哪一种 Executor 执行器?

1.在 Mybatis 配置文件中, 可以指定默认的 ExecutorType 执行器类型;

2.手动给

DefaultSqlSessionFactory 的创建 SqlSession 的方法传递 ExecutorType 类型参数。

- 在 mapper 中传递多个参数:
- 直接在方法中传递参数, xml 文件用#{0} #{1}来获取
- 2.使用 @param 注解, 直接在 xml 文件中通过#{name}来获取
  - resultMap resultMap 的区别:
    - 1.类的字段名字和数据库相同时, 可以直接设置 resultMap 参数为 Pojo 类
- 若不同, 需要设置 resultMap 将结果名字和 Pojo 名字进行转换
  - 使用 MyBatis 的 mapper 接口调用时的要求:
    - 1) Mapper 接口方法名和 mapper.xml 中定义的每个 sql 的 id 相同
    - 2) Mapper 接口方法的输入参数类型和 mapper.xml 中定义的每个 sql 的 parameterType 的类型相同
    - 3) Mapper 接口方法的输出参数类型和 mapper.xml 中定义的每个 sql 的 resultMap 的类型相同
    - 4) Mapper.xml 文件中的 namespace 即是 mapper 接口的类路径。

## Hibernate

- 架构解析
- 用法解析
- 注解解析

## 容器/Server中间件/分布式应用程序协调服务

---

### 分布式应用程序协调服务

- Zookeeper
  - Zookeeper是一个开源的分布式协调服务,为Hadoop的子项目  
是为应用提供一致性的软件, 功能包括: 配置维护、命名服务、分布式同步、组服务等  
主要包含文件系统+监听通知服务
  - Zookeeper包含一个简单的原语集, 提供Java和C的接口, 提供分布式独享锁, 选举, 队列接口。
  - Zookeeper基本运行流程
    - 1、选举Leader(恢复模式)
      - 同步数据
      - 选举Leader过程中算法有很多, 但要达到选举标准是一致的(zk为fast paxos)
    - 4.Leader要具有最高的执行ID,类似root权限
      - 集群中大多数的机器得到响应并接受选出的Leader
  - Zookeeper维持一个类似文件系统的结构, 每个子目录被称为一个znode,在znode下可以创建, 删除子znode, znode可以存储数据(删除目录节点时,目录下不包含任何子节点才能删除成功)
  - zk对于传输数据有一个 1MB 的大小限制
  - 集群配置

- Zookeeper集群通过 conf/zoo.cfg 来配置
  - 生成配置文件  
配置不同的zoo.cfg, 同时配置好clientPort,server列表
  - 标识Server ID  
在zoo.cfg中配置的数据Dir文件夹下, 创建myid文件, 并填入对应的Server ID值, 即, 1-255
  - 启动时, 分别zkServer start conf/zoo-1.cfg 等启动
    - zoo.cfg
      - clientPort 配置启动端口, 默认2181
      - dataDir 数据保存目录, 默认含日志文件
      - dataLogDir 数据日志保存目录
      - server.A=B:C:D[:E]
- 如: server.1=127.0.0.1:2881:3881  
 A为1-255数字, 表示server对应的ID  
 B为server ip  
 C为server 运行端口,默认2181, 即对外提供服务的端口  
 D为Leader选举的端口, 用来执行选举时的通信端口,默认2182  
 E为peerType配置, 用于配置peerType=observer观察者模式(observer模式,只提供读服务,不参与选举), 默认为participant参与选举模式
- 集群下数据同步原理
    - 集群服务数必须为奇数个( $n \geq 3$ ), 因为需要选举, 过半机制
    - 集群中所有服务数据相同
    - 集群角色
- org.apache.zookeeper.server.quorum  
 .QuorumPeer.LearnerType
- Leader(领导者)
    - LearnerType为PARTICIPANT
  - Follower(跟随者)
    - 参与投票, 提供读服务
    - LearnerType为PARTICIPANT
  - Observer(观察者)
    - LearnerType为OBSERVER
    - 不参与投票, 提供读服务
    - 与Follower的区别
      - Follower 参与投票, Observer不参与投票
      - Follower 会得到2个消息, 1个是通过广播得到proposal的内容, 一个是commit消息
    - Observer 只会得到1个commit的INFORM消息
- 适用场景
    - 网络状态不稳定的情况下, 使用Observer模式, 提高读服务节点量
    - Zookeeper的集群读写负载较高的情况下, 使用Observer模式, 提高读服务性能, 而不影响写服务
  - 客户端多, 跨机房跨区域
  - 使用Observer模式的好处:
- 可以扩展读请求服务, 接收更多的请求流量, 而不会牺牲写操作的吞吐量。因为写操作

的吞吐量取决于quorum的size，增加更多的server投票，quorum会增大，降低写操作吞吐量

- 使用observer模式的zoo.cfg配置:

```
peerType=observer
```

```
server.1:localhost:2181:3181:observer
```

- 集群中至少需要有1个PARTICIPANT

- Zookeeper选举和同步

- 一个 ZooKeeper 集群同一时刻只会有一个 Leader，其他都是 Follower 或 Observer。ZooKeeper 配置很简单，每个节点的配置文件(zoo.cfg)都是一样的，只有 myid 文件不一样。myid 的值必须是 zoo.cfg中server.{数值} 的{数值}部分。

- Zookeeper集群Leader选举

```
org.apache.zookeeper.server.quorum
```

```
.QuorumPeer#startLeaderElection
```

```
org.apache.zookeeper.server.quorum.QuorumPeerConfig
```

- LeaderElection

- AuthFastLeaderElection

- FastLeaderElection （最新默认）

```
org.apache.zookeeper.server.quorum.FastLeaderElection
```

- 流程简述

- 目前有5台服务器，每台服务器均没有数据，它们的编号分别是1,2,3,4,5,按编号依次启动，它们的选择过程如下：

- 服务器1启动，给自己投票，然后发投票信息，由于其它机器还没有启动所以它收不到反馈信息，服务器1的状态一直属于Looking(选举状态)。

- 服务器2启动，给自己投票，同时与之前启动的服务器1交换结果，由于服务器2的编号大所以服务器2胜出，但此时投票数没有大于半数，所以两个服务器的状态依然是Looking。

- 服务器3启动，给自己投票，同时与之前启动的服务器1,2交换信息，由于服务器3的编号最大所以服务器3胜出，此时投票数正好大于半数，所以服务器3成为领导者，服务器1,2成为Follower。

- 服务器4启动，给自己投票，同时与之前启动的服务器1,2,3交换信息，尽管服务器4的编号大，但之前服务器3已经胜出，所以服务器4只能成为Follower。

- 服务器5启动，后面的逻辑同服务器4成为Follower。

- 选举操作使用UDP广播消息

```
DatagramSocket udpSocket
```

- 概念

- ServerId: 服务器Id

如有3台服务器，编号为1,2,3,值为1-255

- Zxid: 数据ID，64位，前32位为Epoch,后32位为全局序列  
服务器中存放的最大数据ID,值越大数据越新，在选举中数据越新权重越大  
Zookeeper是要用zxid保证顺序一致性

- Epoch:逻辑时钟，(纪元,时代,新世纪)

或者叫投票的次数，同一轮投票过程中的逻辑时钟值是相同的。

每投完一次票这个数值就会增加，然后与收到的其他服务器返回的投票信息中的值比较，做出不同的判断

- Server状态：选举状态

LOOKING: 竞选状态



FOLLOWING: 随从状态, 同步leader状态, 参与投票

OBSERVING: 观察状态, 同步leader状态, 不参与投票

LEADING: 领导者状态

- 选举消息内容

- 投票完成后, 需要将所有投票信息发送给集群中的所有机器, 包含:

ServerId, Zxid, Epoch 逻辑时钟, 选举状态

- 选举流程

- 1. 服务启动时, 读取当前Server数据及配置信息(dataDir下):

读取Zxid, currentEpoch, acceptedEpoch

然后创建选举线程, 开始选举

- 2. 发送投票信息

首先, 每个Server第一轮都会投票给自己, 申请自己为Leader

投票信息包括: 所选举Leader的Serverid, Zxid, Epoch. Epoch 会随着选举轮数增加而增加

- 3. 接收投票信息

- 若服务器B接收服务器A的信息(A为选举状态LOOKING)

- 1. 判断Epoch逻辑时钟

- a) 若收到的逻辑时钟Epoch大于当前Server的逻辑时钟。

首先更新本Server逻辑时钟Epoch, 同时清空本轮逻辑时钟收集到的其他server的选举数据。

然后判断是否需要更新当前自己选举的Leader Serverid.

判断规则rule judging: 保存的Zxid最大值和Leader Serverid 来进行判断。先判断Zxid, Zxid大者胜出, 然后判断Leader Serverid, Leader Serverid大者胜出。

然后将自身的选举结果(Leader Serverid, Zxid ,Epoch)广播给其他Server

- b) 若收到的逻辑时钟Epoch小于当前Server的逻辑时钟

说明对方Server在一个相对较早的Epoch中, 这时, 只需要将自己的状态数据(Leader Serverid, Zxid, Epoch)广播给其他Server

- c) 若收到的逻辑时钟Epoch等于当前Server的逻辑时钟

根据rule judging来选举Leader, 再将自己选举结果广播给其他Server

- 2. 其次, 判断服务器是不是已经收集到了所有的选举状态:

若是, 根据选举结果设置自己的角色 (FOLLOWING, LEADING), 然后退出选举状态。

若没有收到所有服务器的选举状态, 则判断选举过程中最新选举的Leader是不是得到超过半数以上的服务器支持, 若是, 则尝试200ms之内接受一次数据, 若没有新数据到达, 则说明所有服务器已经默认当前结果, 然后设置自己的角色, 退出选举状态; 反之, 继续选举。

- 服务器A处在其他状态(FOLLOWING, LEADING)

- a) 逻辑时钟Epoch等于当前Server的逻辑时钟, 将该数据保存到

recvset。此时的集群已经处于LEADING状态, 说明此时的集群已经选出结果。

若此时当前Server(服务器B)宣称自己为Leader, 则判断是否有半数以上的服务器选举它, 如果是, 则当前Server为LEADING状态, 反之为FOLLOWING状态, 然后退出选举。

- b) 否则, 这是一条于当前逻辑时钟不符合的消息, 说明在另一个选举中已经有了选举结果,

于是将该选举结果加入到outofelection集合中, 再根据outofelection来判断是否可以结束选举, 如果可以, 保存逻辑时钟, 设置选举状态, 退出选举

- Zookeeper的同步过程

- Leader

- leader需要告知其他服务器当前的最新数据，即最大zxid是什么，此时leader会构建一个NEWLEADER的数据包，包括当前最大的zxid，发送给follower或者observer，  
此时leader会启动一个learnerHandler的线程来处理所有follower的同步请求，同时阻塞主线程，  
只有半数以上的follower同步完毕之后，leader才成为真正的leader，退出选举同步过程。

#### - Follower

- 首先与leader建立连接，如果连接超时失败，则重新进入选举状态选举leader，如果连接成功，则会将自己的最新zxid封装为FOLLOWERINFO发送给leader

- 首先会尝试与leader建立连接,这里有一个机制,如果一定时间内没有连接上,就报错退出,重新回到选举状态.

其次在函数learner::registerWithLeader中发送FOLLOWERINFO封包,该封包中带上自己的最大数据id,也就是会告知leader本机保存的最大数据id.

最后,根据前面对LeaderHandler的分析,leader会根据不同的情况发送

DIFF,UPTODATE,TRUNC,SNAP,依次进行处理就是了,此时follower跟leader的数据也就同步上了.

由于leader端发送的最后一个封包是UPTODATE,因此在接收到这个封包之后follower结束同步数据过程,发送ACK封包回复leader.

#### - 同步算法

- 直接差异化同步 (DIFF同步)

- 仅回滚同步，即删除多余的事务日志 (TRUNC)

- 先回滚再差异化同步 (TRUNC+DIFF)

- 全量同步 (SNAP同步)

- 差异化同步 (DIFF) 条件:  $\text{MinCommittedLog} < \text{peerLastZxid} < \text{MaxCommittedLog}$

MaxCommittedLog

- 使用Socket进行数据同步

#### - 过半机制

- 在领导者选举的过程中，如果某台zkServer获得了超过半数的选票，则此zkServer就可以成为Leader了

#### - 节点读写服务分工

- ZooKeeper 集群的所有机器通过一个 Leader 选举过程来选定一台被称为『Leader』的机器，Leader服务器为客户端提供读和写服务。

- Follower 和 Observer 都能提供读服务，不能提供写服务。两者唯一的区别在于，Observer机器不参与 Leader 选举过程，也不参与写操作的『过半写成功』策略，因此 Observer 可以在不影响写性能的情况下提升集群的读性能。

- 一个Zookeeper集群 ( $N \geq 3$ ,  $N$  为奇数)，那么只有一个Leader (通过FastLeaderElection选主策略选取)，所有的写操作 (客户端请求Leader或Follower的写操作) 都由Leader统一处理，Follower虽然对外提供读写，但写操作会提交到Leader，由Leader和Follower共同保证同一个Follower请求的顺序性，Leader会为每个请求生成一个zxid (高32位是epoch，用来标识leader选举周期，每次一个leader被选出来，都会有一个新的epoch，标识当前属于哪个leader的统治时期，低32位用于递增计数)

- zookeeper 如何保证半数提交后剩下的节点上最新的数据

- zookeeper 的leader和follower的prepare和commit时，只要半数的节点通过就算同意，leader就会commit，那么剩下的半数节点的数据如何同步到最新的呢？

剩下的节点，会进行版本比对，发现版本不一致的话，会更新节点的数据。

- Session

- Session 是指客户端会话;

在 ZooKeeper 中，一个客户端连接是指客户端和 ZooKeeper 服务器之间的TCP长连接。

- ZooKeeper 对外的服务端口默认是2181，客户端启动时，首先会与服务器建立一个TCP 连接，从第一次连接建立开始，客户端会话的生命周期也开始了，通过这个连接，客户端能够通过心跳检测和服务器保持有效的会话，也能够向 ZooKeeper 服务器发送请求并接受响应，同时还能通过该连接接收来自服务器的 Watch 事件通知。

- Session 的 SessionTimeout 值用来设置一个客户端会话的超时时间。当由于服务器压力太大、网络故障或是客户端主动断开连接等各种原因导致客户端连接断开时，只要在 SessionTimeout 规定的时间内能够重新连接上集群中任意一台服务器，那么之前创建的会话 仍然有效。

- 集群配置示例: 3台机器集群

■ Zookeeper会话生命周期: CONNECTING, CONNECTED, CLOSE

■ 在会话到期时，群集将删除该会话拥有的任何/所有短暂节点，并立即通知任何/所有连接的客户端该更改（任何监听这些znode的客户端）

■ Zookeeper节点命名

■ 任何unicode字符都可以在受以下约束限制的路径中使用:

■ 空字符 (\u0000) 不能是路径名的一部分。（这会导致C绑定出现问题。）

■ 无法使用以下字符，因为它们无法正常显示或以令人困惑的方式呈现:

\u0001 - \u0019和\u007F - \u009F。

■ 不允许使用以下字符: \ud800 - \uF8FFF, \uFFFF0 - \uFFFF。

■ "." character可以用作另一个名称的一部分，但是"." 并且"..."不能单独用于表示沿路径的节点，因为ZooKeeper不使用相对路径。以下内容无效: "/ a / b /./ c"或"/a/b/./c"。

■ . 或 ..不能单独作为节点名使用

■ 令牌"zookeeper"被保留。

■ znode类型

■ PERSISTENT-持久化目录节点

客户端与Zookeeper断开连接后，节点依然存在

■ TTL - TTL节点

创建持久化节点时，可以设置接待你的TTL(毫秒)。如果节点在TTL内未修改，且没有子节点，则会被服务删除。

默认为禁用。

■ PERSISTENT\_SEQUENTIAL-持久化顺序标号目录节点

客户端与Zookeeper断开连接后，节点依然存在，只是Zookeeper给该节点进行%10d的顺序编号;

重新连接后，创建的顺序节点会继续按序号增加

■ EPHEMERAL-临时目录节点

客户端与Zookeeper断开连接后，节点被删除

- EPHMERAL\_SEQUENTIAL-临时顺序编号目录节点  
客户端与Zookeeper断开连接后，节点被删除，只是Zookeeper给该节点名称进行顺序编号
- znode操作  
<https://zookeeper.apache.org/doc/current/zookeeperProgrammers.html#Container+Nodes>
  - zkClient -server ip:port[ip2:port2...]/[rootpath] 【连接ZkServer】  
可设置多个server连接地址，若第一个地址连接失败，则会尝试之后的地址 /rootpath 设置连接后使用的根节点，连接后的所有操作都基于该节点
  - create [-s] [-e] path data acl 【创建节点】  
-s 创建节点为顺序节点，创建顺序节点时，会在节点名后自动添加顺序号，编号格式为【%10d】，不足的数字位已0填充，如00000000001，最大值为2147483647  
-e 创建节点为临时节点，节点默认为持久化节点  
path 创建的节点路径,按/分割层级，有多层级时，父节点必须存在才能创建；临时节点不能有子节点；  
acl 节点的访问控制列表，控制权限，权限有5种：  
CREATE,READ,WRITE,DELETE,ADMIN，简写为crwda  
身份认证有4中方式：  
world :默认方式，不限制访问  
auth: 已通过认证的用户访问  
digest: 用户名密码认证  
ip:使用ip认证访问
    - 创建节点： create /kangspace.org "kangspace domain:kangspace.org" world:anyone:crwda  
且节点权限为world,不限制访问

Created /kangspace.org

- 创建持久有序节点列表:

create /kangspace.org/psnode "SAVED PERSISTENT\_SEQUENTIAL NODES"

Created /kangspace.org/psnode

create -s /kangspace.org/psnode/ps "001"

Created /kangspace.org/psnode/ps0000000001

create -s /kangspace.org/psnode/ps "002"

Created /kangspace.org/psnode/ps0000000002

create -s /kangspace.org/psnode/ps "003"

Created /kangspace.org/psnode/ps0000000003

create -s /kangspace.org/psnode/ps "004"

Created /kangspace.org/psnode/ps0000000004

- 创建临时有序节点:

create /kangspace.org/esnode "SAVED EPHEMERAL\_SEQUENTIAL NODES"

Created /kangspace.org/esnode

create -s -e /kangspace.org/esnode/es "001"

Created /kangspace.org/esnode/es0000000000

create -s -e /kangspace.org/esnode/es "002"

Created /kangspace.org/esnode/es0000000001

```
create -s -e /kangspace.org/esnode/es "003"
Created /kangspace.org/esnode/es0000000002
create -s -e /kangspace.org/esnode/es "004"
Created /kangspace.org/esnode/es0000000003
  - ls path [WATCH] 【获取节点下子节点名称列表】
    (只含一级子节点)；可设置监视
      - ls /kangspace.org
[esnode, psnode]
  - ls2 path [WATCH] 【获取节点下子节点名称列表】，并包含该节点属性信息；可设置监视
    - ls2 /kangspace.org
[esnode, psnode]
cZxid = 0x400457aa6
ctime = Wed Jun 24 13:39:42 CST 2020
mZxid = 0x400457aa6
mtime = Wed Jun 24 13:39:42 CST 2020
pZxid = 0x400457ab8
cversion = 6
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 30
numChildren = 2
  - get path [WATCH] 【获取节点数据及属性信息】；
    可设置监视
    - get /kangspace.org
kangspace domain:kangspace.org
cZxid = 0x400457aa6
ctime = Wed Jun 24 13:39:42 CST 2020
mZxid = 0x400457aa6
mtime = Wed Jun 24 13:39:42 CST 2020
pZxid = 0x400457ab8
cversion = 6
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 30
numChildren = 2
  - set path data [version] 【更新节点数据】
    version为要更新的dataVersion，每次更新后dataVersion+1，
    若输入的version不是节点当前version时，更新失败。
    - set /kangspace.org/esnode/en "update001"
cZxid = 0x400457ac0
ctime = Fri Jun 26 10:37:48 CST 2020
mZxid = 0x400457ac5
mtime = Fri Jun 26 10:50:20 CST 2020
```

pZxid = 0x400457ac0  
cversion = 0  
dataVersion = 1  
aclVersion = 0  
ephemeralOwner = 0x172b69b32ba0016  
dataLength = 9  
numChildren = 0  
set /kangspace.org/esnode/en "update001" 3  
version No is not valid : /kangspace.org/esnode/en  
(此时version值应为1)

- delete path [version] 【删除节点】

version 为要更新的数据Version

若节点下存在子节点，则需先删除子节点才能删除父节点

- delete /kangspace.org/esnode/es1 0

#### ■ 节点监听通知机制(WATCH)

- 客户端监听它关注的目录节点，当目录节点发生变化(数据改变、被删除、子目录节点增加删除)时，Zookeeper会通知客户端

#### ■ 监听机制特点

- 一次性触发，客户端监听event，再收到一次事件后，不再监听
- 发送给客户端，监听事件异步发送，watcher的通知事件从server到client端是异步的；所以只有客户端监听后才能感知节点的变化；并且所有客户端收到的事件顺序是一致的。
- 被设置监视的数据，Zookeeper有数据监视和子数据监视；  
getData()和exists()设置数据监视，getChildren()设置子节点监视
- 注册watcher，getData，exists，getChildren
- 触发watcher，create delete setData

#### ■ 可对所有读取操作设置监听

命令行操作:

ls path [WATCH]

ls p2 path [WATCH]

get path [WATCH]

Java API:

getData()

getChildren()

exists()

#### ■ 监听的语义

- Created event 创建事件: 通过exists监听
- Deleted event 删除事件: 通过exists,getData,getChildren监听
- Changed event 变更事件: 通过exists,getData 监听
- Child event 子节点事件: 通过getChildren监听

#### ■ watcher中的相关事件(及对应监听)

##### ■ 节点操作

- 事件名称

- 触发监听
    - create("/path")
      - EventType.NodeCreated
      - exists(), getData()
    - delete("/path")
      - EventType.NodeDeleted
      - exists(),getData()
    - setData("/path")
      - EventType.NodeDataChanged
      - exists(),getData()
    - create("/path/child")
      - EventType.NodeChildrenChanged
      - exists(), getData(), getChildren()[针对于父节点监听]
    - delete("/path/child")
      - EventType.NodeChildrenChanged
      - exists(), getData()
  - Java Zookeeper Api
  - 应用
    - <https://segmentfault.com/a/1190000012185866>
    - Master选举
      - 适用场景
        - 需要实现主从master-slave模式的应用  
(需考虑网络抖动情况下, 主节点最好不随意波动)
      - 算法
        - 最小编号
          - 假设存在选举节点/kangspace.org/leader/election, leader节点/kangspace.org/leader/instance, follower节点/kangsapce.org/servers
  - 所有Server启动时, 都向选举节点下创建临时顺序节点(-e -s)
  - 判断当前节点是否是最小编号, 若是, 则设置当前节点为Master, 其他server为salve; 若不是最小编号, 则监听当前节点的前一个节点的删除事件, 等待下次选举(这里监听前一个节点而不监听父节点, 是为了避免羊群效应, 当监听客户端很多时, 服务器阻塞主线程来通知其他客户端);  
或监听leader节点删除事件, 当leader发生变化时触发重新选举
  - 将Master节点信息保存到leader节点中  
将follower节点信息保存到follower节点下
    - 实现框架: Apache cautor开源框架  
org.apache.curator.framework.recipes.leader.LeaderLatch
    - Apache Curator框架提供的第一种Leader选举策略是Leader Latch。
- 这种选举策略, 其核心思想是初始化多个LeaderLatch, 然后在等待几秒钟后, Curator会自动从中选举出Leader。

Leader Latch选举的本质是连接ZooKeeper，然后在“/curator/latchPath”路径为每个LeaderLatch创建临时有序节点：

在创建临时节点时，org.apache.curator.framework.recipes.leader.LeaderLatch 的 checkLeadership(List children) 方法会将选举路径（/curator/latchPath）下面的所有节点按照序列号排序，如果当前节点的序列号最小，则将该节点设置为leader。反之则监听比当前节点序列号小一位的节点的状态（PS：因为每次都会选择序列号最小的节点为leader，所以在比当前节点序列号小一位的节点未被删除前，当前节点是不可能变成leader的）。如果监听的节点被删除，则会触发重新选举方法——reset()

- Apache Curator框架提供的另一种Leader选举策略是Leader Election。

这种选举策略跟Leader Latch选举策略不同之处在于每个实例都能公平获取领导权，而且当获取领导权的实例在释放领导权之后，该实例还有机会再次获取领导权。另外，选举出来的leader不会一直占有领导权，当 takeLeadership(CuratorFramework client) 方法执行结束之后会自动释放领导权

//创建 CuratorFrameworkImpl实例

```
client = CuratorFrameworkFactory.newClient(SERVER, SESSION_TIMEOUT,
CONNECTION_TIMEOUT, retryPolicy);
```

//创建LeaderSelectorListenerAdapter实例

```
CustomLeaderSelectorListenerAdapter leaderSelectorListener =
    new CustomLeaderSelectorListenerAdapter(client, PATH, "Client #" + i);
leaderSelectorListener.start();
```

```
leaderSelectorListenerList.add(leaderSelectorListener);
```

- 随机竞争分布式锁

- 假设存在Leader节点/kangspace.org/leader/instance,

- 服务启动时,向Leader节点尝试创建Master临时节点，若创建成功，则表示当前没有Master,该服务为Master;若创建失败，则表示当前存在Master，该服务设置为Follower。
- 监听Master临时节点的删除事件;当事件触发时，所有server同时竞争分布式锁，得到锁的服务，去创建Master临时节点，升级为Master，其他服务为follower
  - 多数投票
  - 同Zookeeper FastLeaderElection
- 数据发布与订阅(配置中心)

- 描述：多个订阅者对象同时监听同一主题对象，主题对象状态变化时通知所有订阅者对象更新自身状态。

发布方和订阅方独立封装、独立改变，当一个对象的改变需要同时改变其他对象，并且它不知道有多少个对象需要改变时，可以使用发布订阅模式。（类似观察者模式）

- 配置管理：指集群中的机器拥有某些配置，并且这些配置信息可以动态地改变，那么可以使用发布订阅模式把配置做统一管理。

- 算法：

假设存在配置节点/kangsapce.org/config

用于配置管理

- 配置中心管理服务Manage Server 通过config节点下发配置信息
- 各服务节点监听config节点的数据变化事件，来动态更新自身配置
  - 分布式协调服务/通知(注册中心)
  - 服务发现：指集群中的服务上下线做统一管理，每个服务都可以作为数据的发布方，向集群注册自己的基本信息，而让某些监控服务器作为订阅方。当工作服务器的基本信息改变时，如服务上下线，服务角色变更等，监控服务器可以得



到通知并响应这些变化

- 算法:

假设存在server列表节点/kangspace.org/servers

- 当服务启动时，在servers节点下创建临时节点，用于服务注册
- 各服务节点可以监听servers节点的变化，来同步各自内存中维护的服务列表
  - 分布式锁
    - 分布式锁是在分布式环境下，保护跨进程，跨主机，跨网络的共享资源，实现互斥访问，保护一致性

- 算法:

假设存在锁节点/kangspace.org/lock

- 所有服务都向锁节点下创建临时有序节点，并获取当前锁节点下的所有节点，并排序，排序后各个服务检查自身是否是最小编号节点，最小编号的服务获取到锁。
- 没有获取到锁的服务删除自身节点
- 获取到锁的服务，在执行完成后删除自身节点

- Apache curator实现了分布式锁

```
final CuratorFramework client =
```

```
CuratorFrameworkFactory.newClient(ZK_ADDRESS, new RetryNTimes(10, 5000));  
client.start();
```

```
final InterProcessMutex mutex = new InterProcessMutex(client, ZK_LOCK_PATH);
```

```
//获取锁
```

```
mutex.acquire(1, TimeUnit.SECONDS)
```

```
//释放锁
```

```
mutex.release();
```

- 分布式消息队列

- Zookeeper可以通过顺序节点实现分布式队列

- 算法:

假设存在队列节点 /kangspace.org/queue

- 生产者在队列节点上创建持久化顺序节点来存放数据
- 消费者通过读取顺序节点来取数据  
相当于对于列的put,offer操作
- 消费者监听队列节点子节点变化，触发消费  
相当于队列的take()操作  
Apache curator实现了SimpleDistributedQueue  
(org.apache.curator.framework.recipes.queue.SimpleDistributedQueue)

- 不建议使用zk作为队列来使用

- zk对于传输数据有一个 1MB 的大小限制。

这就意味着实际中zk节点ZNodes必须设计的很小

但实际中队列通常都存放着数以千计的消息

- 如果有很多大的ZNodes，那会严重拖慢的zk启动过程。

包括zk节点之间的同步过程

如果正要用zk当队列，最好去调整initLimit与syncLimit

- 如果一个ZNode过大，也会导致清理变得困难

Netflix不得不设计一个特殊的机制来处理这个大体积的nodes

- 如果zk中某个node下有数千子节点，也会严重拖累zk性能

- zk中的数据都会放置在内存中。
  - 分布式命名服务
    - 类似JNDI的功能,
- 将系统中各种服务名称, 地址及目录信息保存到Zookeeper,需要的时候读取
- 分布式ID生成,
- 利用Zookeeper顺序节点的特性生成分布式ID
- 封装ZookeeperApi的第三方框架
    - Apache curator: Zookeeper Java client
- <http://curator.apache.org/>

## ■ Eureka

### ■ 原理

- Spring Cloud 体系中核心、默认的注册中心组件  
SpringBoot服务式注册中心
- Eureka 分为2部分
  - 服务提供者(Server)
    - 服务注册
      - Eureka Client启动时, 会向Eureka Server注册信息, Server存储这些服务信息,  
并有2层缓存机制维护整个注册表
    - 提供注册表
      - 服务消费者在调用服务时, 如果 Eureka Client 没有缓存注册表的话, 会从 Eureka Server 获取最新的注册表  
/eureka/apps
    - 同步状态
      - Eureka Client 通过注册、心跳机制和 Eureka Server 同步当前客户端的状态。
  - 服务消费者(Client)
    - Eureka Client 是一个 Java 客户端, 用于简化与 Eureka Server 的交互。Eureka Client 会拉取、更新和缓存 Eureka Server 中的信息因此当所有的 Eureka Server 节点都宕掉, 服务消费者依然可以使用缓存中的信息找到服务提供者, 但是当服务有更改的时候会出现信息不一致
    - Register: 服务注册
      - 当Eureka Client向Eureka Server注册时, 会提供自生的元数据, 如application,hostname,ipAddr,等
    - Renew: 服务续约
      - Eureka Client 会每隔 30 秒发送一次心跳来续约。通过续约来告知 Eureka Server 该 Eureka Client 运行正常, 没有出现问题。默认情况下, 如果 Eureka Server 在 90 秒内没有收到 Eureka Client 的续约, Server 端会将实例从其注册表中删除, 此时间可配置, 一般情况不建议更改。  
(由客户端配置,服务端会保存不同服务应用的不同配置)

服务续约任务的调用间隔时间，默认为30秒

`eureka.instance.lease-renewal-interval-in-seconds=30`

服务失效的时间，默认为90秒。

`eureka.instance.lease-expiration-duration-in-seconds=90`

- Eviction: 服务剔除

- 当 Eureka Client 和 Eureka Server 不再有心跳时，Eureka Server 会将该服务实例从服务注册列表中删除，即服务剔除

- Cancel: 服务下线

- Cancel: 服务下线

- GetRegistry: 获取注册列表信息

- Eureka Client 从服务器获取注册表信息，并将其缓存在本地。客户端会使用该信息查找其他服务，从而进行远程调用。该注册列表信息定期（每30秒钟）更新一次。每次返回注册列表信息可能与 Eureka Client 的缓存信息不同，Eureka Client 自动处理。

如果由于某种原因导致注册列表信息不能及时匹配，Eureka Client 则会重新获取整个注册表信息。Eureka Server 缓存注册列表信息，整个注册表以及每个应用程序的信息进行了压缩，压缩内容和没有压缩的内容完全相同。Eureka Client 和 Eureka Server 可以使用 JSON/XML 格式进行通讯。在默认情况下 Eureka Client 使用压缩 JSON 格式来获取注册列表的信息。

- 获取服务是服务消费者的基础，所以必有两个重要参数需要注意：

#启用服务消费者从注册中心拉取服务列表的功能

`eureka.client.fetch-registry=true`

#设置服务消费者从注册中心拉取服务列表的间隔

`eureka.client.registry-fetch-interval-seconds=30`

- Remote Call: 远程调用

- 当 Eureka Client 从注册中心获取到服务提供者信息后，就可以通过 Http 请求调用对应的服务；服务提供者有多个时，Eureka Client 客户端会通过 Ribbon 自动进行负载均衡。

- 自我保护机制

- 默认情况下，如果 Eureka Server 在一定的 90s 内没有接收到某个微服务实例的心跳，会注销该实例。但是在微服务架构下服务之间通常都是跨进程调用，网络通信往往会面临着各种问题，比如微服务状态正常，网络分区故障，导致此实例被注销。
- Eureka Server 在运行期间会去统计心跳失败比例在 15 分钟之内是否低于 85%，如果低于 85%，Eureka Server 即会进入自我保护机制。
- Eureka Server 进入自我保护机制，会出现以下几种情况：
  - (1 Eureka 不再从注册列表中移除因为长时间没收到心跳而应该过期的服务
  - (2 Eureka 仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)
  - (3 当网络稳定时，当前实例新的注册信息会被同步到其它节点中

Eureka 自我保护机制是为了防止误杀服务而提供的一个机制。当个别客户端出现心跳失联时，则认为是客户端的问题，剔除掉客户端；当 Eureka 捕获到大量的心跳失败时，则认为可能是网络问题，进入自我保护机制；当客户端心跳恢复时，Eureka 会自动退出自我保护机制。

- 通过在 Eureka Server 配置如下参数，开启或者关闭保护机制，生产环境建议打开：

```
eureka.server.enable-self-preservation=true
```

```
#设置保护机制的阈值，默认是0.85
```

```
eureka.server.renewal-percent-threshold=0.5
```

- 集群原理

- 工作原理:我们假设有三台 Eureka Server 组成的集群，第一台 Eureka Server 在北京机房，另外两台 Eureka Server 在深圳和西安机房。这样三台 Eureka Server 就组建成了一个跨区域的高可用集群，只要三个地方的任意一个机房不出现问题，都不会影响整个架构的稳定性。

- Eureka Server 集群相互之间通过 Replicate 来同步数据，相互之间不区分主节点和从节点，所有的节点都是平等的。

在这种架构中，节点通过彼此互相注册来提高可用性，每个节点需要添加一个或多个有效的 serviceUrl 指向其他节点。

如果某台 Eureka Server 宕机，Eureka Client 的请求会自动切换到新的 Eureka Server 节点。当宕机的服务器重新恢复后，Eureka 会再次将其纳入到服务器集群管理之中。当节点开始接受客户端请求时，所有的操作都会进行节点间复制，将请求复制到其它 Eureka Server 当前所知的所有节点中。

另外 Eureka Server 的同步遵循着一个非常简单的原则：只要有一条边将节点连接，就可以进行信息传播与同步。所以，如果存在多个节点，只需要将节点之间两两连接起来形成通路，那么其它注册中心都可以共享信息。每个 Eureka Server 同时也是 Eureka Client，多个 Eureka Server 之间通过 P2P 的方式完成服务注册表的同步。

Eureka Server 集群之间的状态是采用异步方式同步的，所以不保证节点间的状态一定是一致的，不过基本能保证最终状态是一致的。

- Eureka 分区

Eureka 提供了 Region 和 Zone 两个概念来进行分区，这两个概念均来自于亚马逊的 AWS

- Region:

可以理解为地理上的不同区域，比如亚洲地区，中国区或者深圳等等。没有具体大小的限制。根据项目具体的情况，可以自行合理划分 region。

- Zone:

可以简单理解为 region 内的具体机房，比如说 region 划分为深圳，然后深圳有两个机房，就可以在此 region 之下划分出 zone1、zone2 两个 zone。

- Eureka 保证 AP:(Availability Partition )

优先保证可用性，提供最终一致性

- Eureka Server 各个节点都是平等的，几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而 Eureka

Client 在向某个 Eureka 注册时, 如果发现连接失败, 则会自动切换至其它节点。只要有一台 Eureka Server 还在, 就能保证注册服务可用(保证可用性), 只不过查到的信息可能不是最新的(不保证强一致性)。

- 配置

- 集群配置时需要开启向服务注册和拉取注册表

- #向注册中心注册自己

- eureka.client.register-with-eureka=true

- #拉取注册表

- eureka.client.fetch-registry=true

- Server:

- (集群中instance.metadata-map.zone不同,其他配置相同)

- //设置实例的zone为zone1

- eureka.instance.metadata-map.zone=zone1

- //设置client的region为region-test

- eureka.client.region=region-east

- //设置zone1的服务列表

- eureka.client.service-url.zone1= <http://localhost:8761/eureka/>  
<http://localhost:8762/eureka/>

- //设置zone2的服务列表

- eureka.client.service-url.zone2= <http://localhost:8763/eureka/>  
<http://localhost:8764/eureka/>

- //设置可用zone

- eureka.client.availability-zones.region-east= zone1,zone2

- Client:

- instance.metadata-map.zone不同

- 其他同Server配置

- Eureka工作流程

- Eureka Server启动成功后, 等待服务注册。在启动时, 如果配置了集群, 集群之间会定时通过Replicate同步注册表, 每个Eureka Server都存在完整独立的注册表信息

- Server/Client:

- #注册中心路径, 如果有多个eureka server, 在这里需要配置其他eureka server的地址, 用","进行区分, 如"<http://address:8888/eureka/>,<http://address:8887/eureka/>"

- eureka.client.service-url.default-

- zone=http://\${eureka.instance.hostname}:\${server.port}/eureka

- Eureka Client启动, 根据配置的Eureka Server去注册服务

- Eureka Client每个30s向Eureka Server进行renew 心跳, 证明服务正常

- Client:

- #心跳间隔5s, 默认30s。每一个服务配置后, 心跳间隔和心跳超时时间会被保存在server端, 不同服务的心跳频率可能不同, server端会根据保存的配置来分别探活

- eureka.instance.lease-renewal-interval-in-seconds=5

- ■ 当Eureka Server 90s内没收到Eureka Client的心跳时，会剔除(eviction)该服务
- Client:
  - #心跳超时时间10s，默认90s。从client端最后一次发出心跳后，达到这个时间没有再次发出心跳，表示服务不可用，将它的实例从注册中心移除
  - eureka.instance.lease-expiration-duration-in-seconds=10
- ■ 单位时间内Eureka Server 统计到大量的Eureka Client 没有发送心跳，则认为可能是网络异常，进入自我保护机制，不再剔除心跳超时的客户端
- Server:
  - #开启注册中心的保护机制，默认是开启
  - eureka.server.enable-self-preservation=true
  - #设置保护机制的阈值，默认是0.85
  - eureka.server.renewal-percent-threshold=0.5
- ■ 当Eureka Client 心跳恢复正常后，Eureka Server自动退出
- ■ Eureka Client定时全量或增量从注册中心同步注册表，并且将注册表信息缓存到本地(第一次为全量，定时同步时为增量)
- ■ 服务调用时，Eureka Client先从本地缓存中获取服务信息。若获取不到，则从注册中心拉取刷新注册表，再同步到本地缓存
- Client:
  - #向注册中心注册自己
  - eureka.client.register-with-eureka=true
  - #同步注册表
  - eureka.client.fetch-registry=true
- ■ Eureka Client获取到目标服务器信息，发起请求
- ■ Eureka Client正常退出时向Eureka Server发送取消(cancel) 请求，Eureka Server将实例从实例列表中删除
- 实例
  - 公共依赖
    - 依赖:

```
org.springframework.boot
spring-boot-dependencies
${spring.boot.version}
pom
import
```

```
org.springframework.cloud
spring-cloud-dependencies
${spring.cloud.version}
pom
import
```

```
org.projectlombok
lombok
${lombok.version}
import
```

- eureka-server

- 依赖:

```
org.springframework.cloud
spring-cloud-context
```

```
org.springframework.cloud
spring-cloud-netflix-eureka-server
```

- 配置:

```
spring.application.name=eureka-server
server.port=8001
#服务地址
eureka.instance.hostname=localhost
eureka.instance.instance-id=${spring.application.name}
eureka.instance.virtual-host-name=${spring.application.name}
eureka.instance.prefer-ip-address=false
eureka.environment=demo
#不向注册中心注册自己
eureka.client.register-with-eureka=false
#取消检索服务
eureka.client.fetch-registry=true
#开启注册中心的保护机制，默认是开启
eureka.server.enable-self-preservation=true
#设置保护机制的阈值，默认是0.85
eureka.server.renewal-percent-threshold=0.5
#注册中心路径，如果有多个eureka server，在这里需要配置其他
eureka server的地址，用","进行区分，如"http://address:8888/eureka
,http://address:8887/eureka"
eureka.client.service-url.default-
zone=http://${eureka.instance.hostname}:${server.port}/eureka
```

- 服务启动代码:

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServerApplication{
```

- eureka-client

- 配置:

```
spring.application.name=eureka-client
server.port=8002
```

#服务地址

eureka.instance.hostname=eureka-client

eureka.instance.instance-id=\${spring.application.name}

eureka.instance.virtual-host-name=\${spring.application.name}

#使用ip作为hostname

eureka.instance.prefer-ip-address=true

eureka.environment=demo

#注册中心路径，表示我们向这个注册中心注册服务，如果向多个注册中心注册，用“,”进行分隔

eureka.client.serviceUrl.defaultZone=<http://localhost:8001/eureka>

#心跳间隔5s，默认30s。每一个服务配置后，心跳间隔和心跳超时时间会被保存在server端，不同服务的心跳频率可能不同，server端会根据保存的配置来分别探活

eureka.instance.lease-renewal-interval-in-seconds=5

#心跳超时时间10s，默认90s。从client端最后一次发出心跳后，达到这个时间没有再次发出心跳，表示服务不可用，将它的实例从注册中心移除

eureka.instance.lease-expiration-duration-in-seconds=10

debug=true

■ 服务启动代码:

//@EnableEurekaClient中已包含@EnableDistroyClients

@SpringBootApplication

@EnableEurekaClient

@Slf4j

public class EurekaClientApplication {}

■ 备注

■ Eureka 客户端注册列表

<http://192.168.10.150:8001/eureka/apps>

■ 若使用ServerName做RestTemplate请求，需在RestTemplate 声明的Bean上添加@LoadBalanced注解

■ 与Zookeeper的区别

■ 相同点

- 都可以用作服务注册中心
- 都是分布式服务，都能保证高可用

■ 不同点

■ Zookeeper是文件目录结构的分布式协调服务

Eureka 是SpringCloud默认自带SpringBoot注册中心服务

■ Zookeeper 实现的是CP模式，保证服务的强一致性，集群采用主从模式，主服务宕机从服务选举时,不能提供对外服务，主服务提供读写操作，从服务提供读服务；服务间使用长连接(响应比Eureka跟快)，keepalive同步状态；

Eureka 实现的是AP模式，保证服务的高可用性，各Server节点都是平等的，某个Server节点宕机时不影响其他Server对外提供服务，只要有1个Server节点存活，服务就可用；服务间采用轮询机制同步状态；

■ Zookeeper开源，且支持监听(WATCH)，可用于Master选举，分布式



锁，消息队列等；

Eureka 1.x开源，2.0以后闭源；只用做服务注册，是SpringCloud默认的注册中心，不支持监听(WATCH)

■ 如何选择:

- 若存在专门的Zookeeper维护团队,或有在使用Zookeeper其他功能(选举,分布式锁)时,可选择Zookeeper
- 若运维能力较弱的团队,则可使用轻量的,基于AP的Eureka

## WebServer

- Nginx
  - 反向代理
  - 负载均衡
  - 静态服务器
- Tomcat
- Resin

## Docker

- 基础应用
- K8S

## 负载均衡

- LVS
- Nginx
- F5
- Ribbon(框架)
  - Feign = Ribbon + Hystrix
- 选型对比

## 消息中间件

---

### Kafka

- Kafka是最初由Linkedin公司在2010年开发，是一个分布式、支持分区的（partition）、多副本的（replica），基于zookeeper协调的分布式流平台(分布式消息系统)  
用Scala语言编写  
<http://kafka.apache.org/intro>
  - <https://www.jianshu.com/p/734cf729d77b>
- 流平台的3个关键功能
  - 发布和订阅记录流，类似于消息队列或企业消息传递系统
  - 以容错的持久化方式存储记录流

- 处理产生的记录流
- 特性
  - 高吞吐量、低延迟：
 

kafka每秒可以处理几十万条消息，它的延迟最低只有几毫秒，每个topic可以分多个partition，

consumer group 对分区进行consume操作
  - 可扩展性： kafka集群支持热扩展
  - 持久性、可靠性: 消息被持久化到本地磁盘，并且支持数据备份，防止数据丢失
  - 容错性: 允许集群中的节点失败(若副本数量为n,则允许n-1个节点失败)
  - 高并发: 支持数千个客户端同时读写
- 使用场景
  - 消息系统:
 

解耦生产者和消费者，缓存消息等
  - 存储系统:
 

kafka提供稳定的性能和容错的持久化
  - 日志收集:
 

可用Kafka收集各种服务log，通过Kafka以统一服务的方式和开放给各个消费者
  - 流处理:
 

Spark streaming 和 storm
  - 用户活动跟踪:
 

Kafka经常被用来记录web用户或者app用户的各种活动，如网页浏览、搜索、点击等活动，这些活动被服务器发布到topic中，通过订阅这些topic来做实时监控分析。
  - 运营指标:
 

Kafka经常被用来记录运营监控数据。
- 设计思想
  - Kafka Broker Leader的选举:
 

Kafka Broker集群使用Zookeeper管理，

注册成功的Broker为 Kafka Broker Controller

其他Broker为 Kafka Broker follower，

Broker之间是平等的,Broker上的Partition是有主从关系的;
  - Consumer Group: 消费者组
 

对于topic ,Consumer可以组成一个组，

\*一个分区partition 中消息只能被同一个组中的一个消费者消费，其他消费者阻塞；

\*一个分区中消息可以被多个不同组同时消费；

不同组消费的数据是相同的。

\*多个分区partition 中的消息可以被同一组中相同数量的消费者消费；

这些消息会分布在不同的分区中，同一组中的消费者消费不同(分区)的数据；

一个topic有多(n)个分区时，

当消费者组中消费者数量小于n时，则消费者会平均监听这n个分区；

当消费者组中消费者数量等于n时，则每个消费者监听一个分区；

当消费者组中消费者数量大于n时，则有n个消费者分别监听n个分区，剩下的消费者阻塞；

最佳实践: topic partition 数量等于消费者组中消费者的数量

producer流量增大时，可扩展分区和消费者

- Consumer Rebalance的触发条件
  - Consumer增加或删除会触发 Consumer Group的Rebalance
  - Broker的增加或者减少都会触发 Consumer Rebalance
  - Topic 分区变化会触发Consumer Rebalance
- Consumer 消费者: Consumer处理partition的message的时候是O(1)顺序读取，所以需要维护消费的offset，高级API offset存在于Zookeeper中，低级API offset由自己维护。一般使用高级API
  - Consumer 的delivery gurarantee，默认是读完message先commit再处理message，autocommit默认为true；  
这种情况下如果提交完再处理消息，消息处理失败的情况下，若不记录就会出现消息丢失；
- Delivery Mode 消息分发模式
- Topic & Partition
  - topic 相当于一个队列，生产者发送的消费者必须指定topic， kafka会均匀的把数据分不到不同的partition，每个partition相当于一个子queue。  
在物理存储上，每个partition对应一个物理目录，文件夹命名为[topicname]-[partition序号]，一个topic可以有无数个partition。  
添加新的partition后，旧partition中的数据不会改变，新的分区内容为空，在随后的进入topic的消息会加入到新的分区中
- Partition Replica (分区副本)  
每个partition可以在其他kafka broker节点上存副本，以便某个kafka broker节点宕机不会影响kafka集群。  
存Replica是按照kafka broker的顺序存。
  - 消息传送:  
producer先把消息发送到partition leader，再由leader发送给其他follower
  - 在向Producer发送ACK前需要保证有多少个Replica已经收到该消息：  
根据ack配的个数而定
  - 怎样处理某个Replica不工作的情况：
- 如果这个不工作的partition replica不在ack列表中，就是producer在发送消息到partition leader上，partition leader向partition follower发送message没有响应而已，这个不会影响整个系统，也不会有什么問題。
- 如果这个不工作的partition replica在ack列表中的话，producer发送的message的时候会等待这个不工作的partition replica写message成功，但是会等到time out，然后返回失败因为某个ack列表中的partition replica没有响应，此时kafka会自动的把这个不工作的partition replica从ack列表中移除，以后的producer发送message的时候就不会有这个ack列表下的这个部工作的partition replica了。
  - 怎样处理Failed Replica恢复回来的情况：
- 如果这个partition replica之前不在ack列表中，那么启动后重新受Zookeeper管理即可，之后producer发送message的时候，partition leader会继续发送message到这个partition follower上

- 如果这个partition replica之前在ack列表中，此时重启后，需要把这个partition replica再手动加到ack列表中。（ack列表是手动添加的，出现某个不工作的partition replica的时候自动从ack列表中移除的）

kafka 2.5.0中 broker server上线后，会重新加入rsr列表

- Partition leader与follower

- partition也有leader和follower之分。

leader是主partition，producer写kafka的时候先写partition leader，再由partition leader push给其他的partition follower。

partition leader与follower的信息受Zookeeper控制，一旦partition leader所在的broker节点宕机，zookeeper从其他的broker的partition follower上选择follower变为partition leader。

- Topic分配partition和partition replica的算法

- (1) 将Broker (size=n) 和待分配的Partition排序。
  - (2) 将第i个Partition分配到第  $(i \% n)$  个Broker上。
  - (3) 将第i个Partition的第j个Replica分配到第  $((i + j) \% n)$  个Broker上

- 架构图:

[https://collabnix.com/wp-content/uploads/2019/05/IMG\\_20190525\\_212913-1024x801-1-1024x801.jpg](https://collabnix.com/wp-content/uploads/2019/05/IMG_20190525_212913-1024x801-1-1024x801.jpg)

- 消息投递可靠性

- 发送成功就算成功投递; 不保证消息成功投递到broker
  - Master-Slave 模型，只有当Master和Slave都收到消息才算投递成功。保证了最高的投递可靠性，降低了性能
  - Master确认收到及表示成功投递；

- Partition ack

消息生产成功状态确认

- 当ack=1，表示producer写partition leader成功后，broker就返回成功，无论其他的partition follower是否写成功
    - broker宕机导致partition的follower和leader切换，会导致丢数据
  - 当ack=2，表示producer写partition leader和其他一个follower成功的时候，broker就返回成功，无论其他的partition follower是否写成功
  - 当ack=-1[partition的数量]的时候，表示只有producer全部写成功的时候，才算成功，kafka broker才返回成功信息。
  - ACK前需要保证有多少备份
ISR(in-sync Replica)
    - 0: 不等待broker的消息确认 延迟最小，但可能丢数据
    - 1: leader 已经接受了数据的确认消息，Replica异步拉取消息，比较折中,默认配置:

```
transaction.state.log.min.isr=1
```
    - -1: ISR列表中的所有Replica都需要返回确认信息
    - min.insync.replicas=1 至少有一个Replica返回成功，否则Producer异常

- Message状态

- kafka中，消息的状态保存在consumer中，broker只记录offset值。

- Message持久化
  - Kafka会把消息持久化到本地本间系统，并且保持O(1)极高的效率。  
Kafka写入和读取都是按顺序操作的。  
可达到单机每秒10w数据
- Message有效期
  - Kafka会长久保留其中的消息，以便Consumer可以多次消费。  
可配置。默认168h/7天
- Kafka吞吐量
  - kafka的高吞吐量体现在读写上，分布式并发的读写都非常快；  
写的性能体现在以O(1)的时间复杂度进行顺序写入；  
读的性能体现在以O(1)的时间复杂度进行顺序读取；  
增加topic的分区partition，和consumer group中的consumer，可提高并发性能
- Kafka delivery guarantee(message传送保证)
  - At most once , 最多1次，消息可能会丢失，不会重复传输
  - At least once , 最少1次，消息不会丢，可能会传送多次
  - Exactly once , 仅传输1次
- 批量发送
  - Kafka支持以消息集合为单位进行批量发送，以提高push效率。
- PUSH-AND-PULL
  - kafka中的Producer和Consumer采用PUSH-AND-PULL模式。  
Producer向broker push消息，  
Consumer从broker pull消息；  
二者异步处理
- Kafka集群中broker之间的关系
  - broker不是主从关系，各个broker在集群中的地位是相同的，可以随意增加或删除broker节点
- 负载均衡
  - kafka 0.8.x 使用metadata api来管理broker的负载
  - kafka0.7.x 使用Zookeeper实现负载
- 同步异步
  - Producer 采用异步PUSH方式，极大提高了Kafka系统的吞吐率  
可通过参数控制同步/异步方式
- 分区机制
  - Kafka的broker支持消息分区partition，Producer可以决定把消息发到哪个partition中；  
在一个partition中，message的顺序就是Producer发送消息的顺序，一个topic可以有多个partition，partition数量可配置  
partition可以设置Replic副本，副本保存在不同的broker上，第一个partition是leader，其他partition是follower；  
消息先写到partition leader上，再由leader push到其他follower上。
- 离线数据装载
  - kafka由于对可拓展的数据持久化的支持，它也适合想Hadoop或者数据仓库

中进行数据装载

- 实时数据和离线数据
  - kafka即支持离线数据，也支持实时数据，因为kafka的message持久化到文件，并可以设置有效期(默认7天)；  
因此可以把kafka作为高效的存储来使用，可以作为离线数据；  
当作为分布式实时消息系统时，大多数情况下还是用于实时处理的。
- 插件支持
  - 不少活跃社区已经开发出不少插件来扩展Kafka的功能
- 解耦和异步通信
  - 作为MQ，Producer和Consumer异步处理了消息
- 冗余
  - replica 有多个副本，保证一个broker节点宕机后不影响整个服务
- 扩展性
  - broker节点可以水平扩展，partition也可以水平增加，partition replica也可以水平增加
- 峰值
  - 在访问量剧增的情况下，kafka水平扩展，应用仍然需要继续发挥作用
- kafka文件存储机制
  - kafka名词解释
    - Broker: Kafka节点，一个Kafka节点就是一个broker，多个broker可以组成一个Kafka集群。
    - Topic: Kafka 节点数据保存的队列
    - Partition: topic物理上的分组，一个topic可以分为多个partition，每个partition是一个有序的队列,物理上topic按分区保存，名称为[topicname]-[partitionnum]；.index,.log,为稀疏索引,数据是顺序保存
    - Partition Replica: partition 分区副本，平均分布在各个broker,可以指定 $\geq 1$ ,  $\leq$ broker数量的副本数，  
第一个Replica会作为Leader Partition提供服务，其他Replica作为follower；  
选择follower时需要兼顾一个问题,就是新leader server上所已经承载的partition leader的个数,如果一个server上有过多的partition leader,意味着此server将承受着更多的IO压力.在选举新leader,需要考虑到"负载均衡",partition leader较少的broker将会更有可能成为新的leader。
      - 副本分配算法
        - 将所有N Broker和待分配的i个Partition排序.
        - 将第i个Partition分配到第 $(i \bmod n)$ 个Broker上.
        - 将第i个Partition的第j个副本分配到第 $((i + j) \bmod n)$ 个Broker上.
    - Segment: partition物理上由多个segment组成，每个Segment存着message信息，消息存储的文件；包含.index和.log2部分

- Producer: 消息的生产者, 向Kafka Leader Partition 发送消息;可以有多个;  
 Producer将消息发送到哪个Broker:  
 Producer发送消息到Broker是由生产者客户端决定的, 生产者客户端会获取所有Broker和Leader Partition信息,  
 根据partition负载均衡算法(或自己实现kafka.producer.Partitioner接口)利用key和partition 决定消息写入到哪个partition, 然后找到对应的Broker来发送消息  
 生产者负责选择将哪个记录分配给主题中的哪个分区。可以简单地平衡负载而以循环方式完成此操作, 也可以根据某些语义分区功能(例如基于记录中的某些键)完成此操作
- Consumer: 订阅topic消费message, consumer作为一个线程来消费  
 由消费者维护topic partition的消费情况,即offset位置, 存储在Zookeeper和Kafka服务端
- Consumer Group: 消费者组, 1个partition只能被同一个消费者组中的一个消费者消费, 其他消费者阻塞;  
 1个partition 可被同步的消费者组中的各1个消费者同时消费;  
 多个partition可被消费者组中相同数量的消费者消费;  
 当高并发下, 可同时增加partition和消费者提高消息的消费效率;
- 持久化
  - 消息会被持久化到系统文件(segment),只有被刷新到磁盘的消息才会被消费
- kafka中文件存储结构
  - topic中partition存储分布:  
 topic 在磁盘上是按partition保存的,  
 partition 是一个文件夹目录;  
 以[topicName]-[partitionIndex]命名  
 partitionIndex起始为0  
 Partition是一个Queue的结构, 每个Partition中的消息都是有序的, 生产的消息被不断追加到Partition上, 其中的每一个消息都被赋予了一个唯一的offset值。
  - partition中文件存储方式:  
 每个partition(目录)相当于一个巨型文件被平均分配到多个大小相等segment(段)数据文件中。但每个段segment file消息数量不一定相等, 这种特性方便old segment file快速被删除。  
 每个partition只需要支持顺序读写就行了, segment文件生命周期由服务端配置参数决定。
  - partition中segment文件存储结构:  
 producer发message到某个topic, message会被均匀的分布到多个partition上(随机或根据用户指定的回调函数进行分布), kafka broker收到message往对应partition的最后一个segment上添加该消息, 当某个segment上的消息条数达到配置值或消息发布时间超过阈值时, segment上的消息会被flush到磁盘, 只有flush到磁盘上的消息consumer才能消费, segment达到一定的大小后将不会再往该segment写数据, broker会创建新的segment。  
 每个part在内存中对应一个index, 记录每个segment中的第一条消息偏移。

segment file组成：由2大部分组成，分别为index file和data file，此2个文件一一对应，成对出现，后缀".index"和".log"分别表示为segment索引文件、数据文件。

segment文件命名规则：partition全局的第一个segment从0开始，后续每个segment文件名为上一个全局partition的最大offset(偏移message数)。数值最大为64位long大小，19位数字字符长度(无符号整数)，没有数字用0填充;共20位,首位为0

每个segment中存储很多条消息，消息id由其逻辑位置决定，即从消息id可直接定位到消息的存储位置，避免id到位置的额外映射。

- 如:  
00000000000000000000.index  
00000000000000000000.log  
00000000000000368769.index  
00000000000000368769.log

#### ■ kafka最佳实践

- Topic: 创建topic时，指定partition数量，最好 $\geq$ broker数，提高kafka效率，和broker使用率;  
同时指定replica数量，replica最好 $>2$ , $\leq$ broker数量;  
replica=1时，表示只有1份数据；replica不能大于 $>$ broker数量; replica $\geq 2$ 时,才能保证有某个broker宕机时，不影响业务(当所有存replica的broker都宕机后，该partition不能正常工作)

- Kafka 消息存储可设置过期时间，默认7天

- Kafka提供的命令行大部分需要提供 bootstrap-server 或 zookeeper

- 查询消费者组--new-consumer需要bootstrap-server  
查询其他消费者组用zookeeper
- 消费者消费数据时链接zookeeper  
./bin/kafka-console-consumer.sh --zookeeper 192.168.1.183:2181 --from-beginning --topic dyzjPaperTaskSubmitTopic
- 生产者发布数据是用broker-list  
./bin/kafka-console-producer.sh --broker-list 192.168.1.183:9092 --topic dyzjPaperTaskSubmitTopic

#### ■ Kafka目录及配置,命令说明

默认端口: 9092

##### ■ 目录及配置

- bin/bin windows Kafka命令所在目录
  - --bootstrap-server or --zookeeper 是命令中必传项
  - kafka-server-start.sh [-daemon] server.properties [--override property=value]\*  
启动Server
    - 如:  
./bin/kafka-server-start.sh config/server.properties &
  - kafka-server-stop.sh 停止集群服务
    - 如:



./bin/kafka-server-stop.sh

■ kafka-topics.sh kafka topic相关操作

- --bootstrap-server [String ip:port,...] 指定kafka broker地址, 与zookeeper其一必填
- --zookeeper [String ip:port,...] 指定zookeeper地址与--bootstrap-server其一必填
- --create 创建
- --alter 修改
- --delete 删除
- --topic [String topicName] 指定topic名称
- --partitions [int cnt] 指定分区数量, 默认为1  
--create --alter 可指定
- --replication-factor [int cnt] 指定复制因子, 即副本数量,  
--create 时可指定
- --describe 查看topic详情, 含partition,rsr , replica等
- --list 查看topic列表
- --if-not-exists 当topic不存在时执行  
--create可指定
- --replica-assignment 手工指定分区副本  
这种方式根据分区号的数值大小按照从小到大的顺序进行排列, 分区与分区之间用逗号"," 隔开, 分区内多个副本用冒号":"隔开。并且在使用 replica-assignment 参数创建主题时不需要原本必备的 partitions 和 replication-factor 这两个参数。  
如:  
--replica-assignment 1:2,3:4,5:6  
topic有三个partition, 其中partition\_0的replica分布在broker1和2上, partition\_1的replica分布在broker3和4上, partition\_2的replica分布在broker5和6上。  
--create , --alter可指定
- 如:

```
./bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --topic test-topic --partitions 3 --replication-factor 3
```

Created topic test-topic.

java.lang.IllegalArgumentException: Only one of --bootstrap-server or --zookeeper must be specified

--create 创建topic

--alter 修改topic partitions,replication-fator,配置等信息

--delete 删除partitions

--replication-factor 复制因子

--partitions 分区数

--topic topic名称

--list 查看kafka topic列表

--describe 查看topic信息

```
./bin/kafka-topics.sh --list --bootstrap-server 127.0.0.1:9092
```

查看topic信息:

```
./bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic test-topic
```

修改Topic分区:

```
./bin/kafka-topics.sh --alter --zookeeper localhost:22181 --topic test-topic --  
partitions 2
```

- kafka-console-producer.sh kafka控制台生产者
- 如:

```
./bin/kafka-console-producer.sh --bootstrap-server 127.0.0.1:9092 --topic test-topic
```

- kafka-console-consumer.sh kafka控制台消费者
- --group 指定消费者分组
- --from-beginning 从最早位置开始消费数据
- --offset [String offsetId] 指定消费位置id
- --partition [int pid] 指定监听的分区号
- 如:

```
./bin/kafka-con
```

```
sole-consumer.sh --bootstrap-server localhost:9092 --topic test-topic --from-  
beginning --group 0
```

- kafka-consumer-groups.sh kafka消费者组操作
- --all-groups 应用到所有消费者组
- --group [String group] 指定要查看的组

```
- ./bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --  
describe --offsets --group 1
```

- --describe 显示消费者组及消费情况

需指定--all-groups 或--group 使用

- >./bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --

```
describe --all-groups
```

| GROUP       | TOPIC      | PARTITION | CURRENT-OFFSET | LOG-END-OFFSET | LAG |
|-------------|------------|-----------|----------------|----------------|-----|
| CONSUMER-ID | HOST       | CLIENT-ID |                |                |     |
| 1           | test-topic | 0         | 85             | 109            | 24  |

TOPIC topic名字

PARTITION 分区id

CURRENT-OFFSET 当前已消费的条数

LOG-END-OFFSET 总条数

LAG 未消费的条数

CONSUMER-ID 消费id

HOST 主机ip

CLIENT-ID 客户端id

- --offsets 只显示消费者组分区消费情况

仅在--bootstrap-server ,--describe下使用

是--describe的默认子行为

- >./bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --

```
describe --all-groups --offsets
```

- --list 显示所有消费者组名称

- >./bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list

```
0
```

```
1
```

- --members 显示消费者组中成员的描述信息

仅在--bootstrap-server ,--describe下使用

不显示offset信息

- >./bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --

describe --group 1 --members

- --delete 删除组

- --delete-offsets 删除消费者组消费位置

同时只支持一个消费者和多个topic

- >./bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --

delete-offsets --group 0 --topic test-topic execute

- --execute 执行操作: --reset-offsets

- --reset-offsets 重置消费者组消费信息

可使用--dry-run或--execute 执行

- --all-topics Consider all topics assigned to a

group in the `reset-offsets` process.

- --topic [String topicName] 指定topic名称, 用于delete 或 offset process

支持多个topic

需指定--all-topics 或 --topic [topicName:partitionId,..] 使用

. In `reset-offsets` case, partitions can be specified using this format:

`topic1:0,1,2`, where 0,1,2 are the

partition to be included in the process.

- 重新指定消费者组消费位置

--to-earliest Reset offsets to earliest offset.

--to-latest Reset offsets to latest offset.

--to-offset <Long: offset> Reset offsets to a specific offset.

需在--reset-offsets 下指定该参数

- >./bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --

reset-offsets --group 0 --topic test-topic --to-earliest execute

- config 配置文件目录

<http://kafka.apache.org/documentation/#configuration>

- server.properties kafka broker(Server)配置

- broker.id BrokerId

- log.dirs 数据文件保存目录,按逗号分隔的目录列表

- zookeeper.connect zookeeper链接地址

格式:

hostname1:port1,hostname2:port2,hostname3:port3/chroot/path

- listeners 设置监听类型和端口

FORMAT:

listeners = listener\_name://[host.name]:port

host.name 为变量或具体指定的hostname

默认: PLAINTEXT://:9092,

host.name默认取java.net.InetAddress.getCanonicalHostName(),因为使用hostname作为监听地址,所以在生产者/消费者服务器需要配置broker的hostname:ip映射

- num.partitions=1 设置每个topic默认分区数

- compress-type 压缩类型

gzip snappy lz4 zstd

- offsets.topic.replication.factor topic 分区默认副本数

- offsets.topic.segment.bytes topic分区保存磁盘的segment文件大小

- 日志刷新磁盘缓存数

log.flush.interval.messages=10000

日志刷盘频率

log.flush.interval.ms=1000

- log.retention.hours=168 日志最大保留时长,单位:小时
- log.segment.bytes=1073741824 日志分片大小,若超过该大小,则创建新的日志

片段文件,默认片段大小:1G

- producer.properties kafka-console-pruducer生产者配置
- consumer.properties kafka-console-consumer消费者配置
- libs jar包目录
- logs 日志目录
- site-docs 文档目录

#### ■ 常用命令

##### ■ 创建Topic:

```
kafka-topic.sh --create --bootstrap-server 127.0.0.1:9092 --partitions 1 --  
replication-factor 1 --topic newTopic
```

##### ■ 查看topic信息:

```
kafka-topic.sh --describe --bootstrap-server 127.0.0.1:9092 --topic  
newTopic
```

##### ■ 修改topic分区:

```
kafka-topic.sh --alter --bootstrap-server 127.0.0.1:9092 --partitions 2 --  
topic newTopic
```

##### ■ 删除分区:

```
kafka-topic.sh --delete --bootstrap-server 127.0.0.1:9092 --topic  
newTopic
```

##### ■ 查看topic消费情况:

```
./bin/kafka-consumer-groups.sh --bootstrap-server localhost:9092 --  
describe --all-groups --offsets
```

#### ■ Zookeeper结构

##### ■ /brokers kafka实例目录

/brokers/ids/ kafka实例列表

/brokers/topics/ kafka下topic列表

/brokers/topics/[topic] kafka下某个topic

/brokers/topics/[topic]/partitions/ kafka下topic分区列表

/brokers/topics/[topic]/partitions/0..n/state

/brokers/topics/\_consumer\_offsets

/consumers kafka所有消费者

#### ■ 快速开始

<http://kafka.apache.org/quickstart>

- 下载最新安装包并解压

```
tar -xzf kafka_2.12-2.5.0.tgz
```

```
cd kafka_2.12-2.5.0
```

- 启动Kafka Server (需Java环境)

依赖于zookeeper , 需先启动Zookeeper

bin/kafka-server-start.sh config/server.properties&

可使用已有Zookeeper服务,或启动Kafka自带的单节点Zookeeper服务

(启动Kafka自带Zookeeper单节点服务)

bin/zookeeper-server-start.sh config/zookeeper.properties&

- ■ 创建topic  
创建topic命令需指定broker地址或zookeeper地址,  
java.lang.IllegalArgumentException: Only one of --bootstrap-server or --  
zookeeper must be specified  
topic名称,partition数量,partition replica数量  
./bin/kafka-topics.sh --create --bootstrap-server 127.0.0.1:9092 --topic  
test-topic  
Created topic test-topic.  
java.lang.IllegalArgumentException: Only one of --bootstrap-server or --  
zookeeper must be specified  
--create 创建topic  
--alter 修改topic partitions,replication-factor,配置等信息  
--delete 删除partitions  
--replication-factor 复制因子  
--partitions 分区数  
--topic topic名称  
--list 查看kafka topic列表  
--describe 查看topic信息  
./bin/kafka-topics.sh --list --bootstrap-server 127.0.0.1:9092  
查看topic信息:  
./bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --topic  
test-topic  
修改Topic分区:  
./bin/kafka-topics.sh --alter --zookeeper localhost:22181 --topic test-topic  
--partitions 2  
手工指副本分配:  
--replica-assignment broker\_id\_for\_part1\_replica1 :  
broker\_id\_for\_part1\_replica2...
- ■ 创建生产者, 发送消息  
./bin/kafka-console-producer.sh --bootstrap-server 127.0.0.1:9092 --topic  
test-topic
- ■ 启动消费者, 消费消息  
./bin/kafka-consumer.sh --bootstrap-server localhost:9092 --topic test-topic --  
from-beginning --group 0  
--from-beginning 从头开始消费  
--group 指定Consumer Group
- ■ 创建kafka server集群
  - 复制多个server.properties配置文件, 并修改以下配置:  
broker.id=0  
listeners=PLAINTEXT://:9093  
logDir=/tmp/kafka-logs-1

- 分别启动3个Server
  - bin/kafka-server-start.sh config/server-1.properties &
  - ...
  - bin/kafka-server-start.sh config/server-2.properties &
  - ...
- 创建一个具有3分区,3复制因子的topic: topic-three
  - ./bin/kafka-topics.sh --create --bootstrap-server localhost:9092 --
  - partitions 3 --replication-factor 3 --topic topic-three
  - ./bin/kafka-topics.sh --describe --bootstrap-server localhost:9092 --
  - topic topic-three
  - Topic: topic-three    PartitionCount: 3    ReplicationFactor: 3
  - Configs: segment.bytes=1073741824
  - Topic: topic-three    Partition: 0    Leader: 1    Replicas: 1,2,0 Isr:
  - 1,2,0
  - Topic: topic-three    Partition: 1    Leader: 0    Replicas: 0,1,2 Isr:
  - 0,1,2
  - Topic: topic-three    Partition: 2    Leader: 2    Replicas: 2,0,1 Isr:
  - 2,0,1

- UI客户端
  - kafkatool

## RocketMQ(Apache)

<https://github.com/apache/rocketmq/tree/master/docs/cn>

- 概述:
  - RocketMQ是一个统一消息引擎,轻量级数据处理平台;
  - 主要优势是高可用,低延迟;
  - 消息存储使用高性能低延迟的文件存储;
  - 最初为阿里巴巴中间件团队开发,2016年底捐赠给apache
- 特点
  - 低延迟
    - Low Latency
    - More than 99.6% response latency within 1 millisecond under high pressure.
  - 财务导向
    - Finance Oriented
    - High availability with tracking and auditing features.
  - 行业可持续的
    - Industry Sustainable
  - 大数据友好
    - BigData Friendly
    - Batch transferring with versatile integration for flooding throughput.
  - 支持消息累积(在磁盘空间充足,不损失性能的情况下累积数据)
    - Massive Accumulation
    - Given sufficient disk space, accumulate messages without performance

loss.

- 原理和实现

- 基本概念

- <https://github.com/apache/rocketmq/blob/master/docs/cn/concept.md>

- 消息模型 (Message Model)

- 【RocketMQ主要由 Producer、Broker、Consumer 三部分组成】，其中Producer 负责生产消息，Consumer 负责消费消息，Broker 负责存储消息。【Broker 在实际部署过程中对应一台服务器，每个 Broker 可以存储多个Topic的消息，每个Topic的消息也可以分片存储于不同的Broker】。Message Queue 用于存储消息的物理地址，每个Topic中的消息地址存储于多个 Message Queue 中。ConsumerGroup 由多个 Consumer 实例构成。(同Kafka)

- 2.消息生产者 (Producer)

- 负责生产消息，一般由业务系统负责生产消息。一个消息生产者会把业务应用系统里产生的消息发送到broker服务器。【RocketMQ提供多种发送方式，同步发送、异步发送、顺序发送、单向发送。同步和异步方式均需要Broker返回确认信息，单向发送不需要。】

- 3.消息消费者 (Consumer)

- 负责消费消息，一般是后台系统负责异步消费。一个消息消费者会从Broker服务器拉取消息、并将其提供给应用程序。【从用户应用的角度而言提供了两种消费形式：拉取式消费、推动式消费。】

- 主题 (Topic)

- 表示一类消息的集合，每个主题包含若干条消息，每条消息只能属于一个主题，是RocketMQ进行消息订阅的基本单位。

- topic中设置messageQueue(ConsumerQueue)默认为8:

- readQueueNums:8

- writeQueueNums:8

- 代理服务器 (Broker Server)

- 消息中转角色，负责存储消息、转发消息。代理服务器在RocketMQ系统中负责接收从生产者发送来的消息并存储、同时为消费者的拉取请求作准备。代理服务器也存储消息相关的元数据，包括消费者组、消费进度偏移和主题和队列消息等。

- 命名服务 (Name Server)

- 命名服务充当路由消息的提供者。生产者或消费者能够通过名字服务查找各主题相应的Broker IP列表。多个Namesrv实例组成集群，但相互独立，没有信息交换。

- 拉取式消费 (Pull Consumer)

- Consumer消费的一种类型，应用通常主动调用Consumer的拉消息方法从Broker服务器拉消息、主动权由应用控制。一旦获取了批量消息，应用就会启动消费过程。

- 推送式消费 (Push Consumer)

- Consumer消费的一种类型，该模式下Broker收到数据后会主动推送给消费端，该消费模式一般实时性较高。

- 生产者组 (Producer Group)

- 同一类Producer的集合，这类Producer发送同一类消息且发送逻辑一

致。如果发送的是事务消息且原始生产者在发送之后崩溃，则Broker服务器会联系同一生产者组的其他生产者实例以提交或回溯消费。

- ■ 消费者组（Consumer Group）  
同一类Consumer的集合，这类Consumer通常消费同一类消息且消费逻辑一致。消费者组使得在消息消费方面，实现负载均衡和容错的目标变得非常容易。要注意的是，消费者组的消费者实例必须订阅完全相同的Topic。【RocketMQ 支持两种消息模式：集群消费（Clustering）和广播消费（Broadcasting）】。  
同一个Consumer Group下，通过增加Consumer实例的数量来提高并行度，集群消费模式下,超过订阅队列数(messagequeue)的Consumer实例无效。
- #集群消费模式下,一个消费者组内所有消费者分担消息的消费,此时,最大消费者数<=ConsumerQueueNums(也叫MessageQueueNums),多余的消费者不消费数据;小于时,分摊到已有的消费者上;
- #广播模式下,一个消费者组内每个消费者都消费全部数据;
- 集群消费模式样例:  
1  
[SEND] 同步发送消息结果: SendResult [sendStatus=SEND\_OK, msgId=C0A82B326DD918B4AAC24115970B0001, offsetMsgId=C0A82B3200002A9F000000000000E3683, messageQueue=MessageQueue [topic=JAVA\_MINDMAP\_TOPIC, brokerName=user.local, queueId=6], queueOffset=17]  
INPUT MESSAGE:  
[RECEIVE] Consumer:0, ConsumeMessageThread\_2 推送式消费  
Receive New Messages: [MessageExt [queueId=6, storeSize=195, queueOffset=17, sysFlag=0, bornTimestamp=1615619933963, bornHost=/192.168.43.50:56297, storeTimestamp=1615619933965, storeHost=/192.168.43.50:10911, msgId=C0A82B3200002A9F000000000000E3683, commitLogOffset=931459, bodyCRC=64810935, reconsumeTimes=0, preparedTransactionOffset=0, toString()=Message{topic='JAVA\_MINDMAP\_TOPIC', flag=0, properties={MIN\_OFFSET=0, MAX\_OFFSET=18, CONSUME\_START\_TIME=1615619933970, UNIQ\_KEY=C0A82B326DD918B4AAC24115970B0001, CLUSTER=DefaultCluster, WAIT=true, TAGS=TagA}, body=[49], transactionId='null'}]]  
[SEND] 异步发送消息成功:SendResult [sendStatus=SEND\_OK, msgId=C0A82B326DD918B4AAC24115970B0001, offsetMsgId=C0A82B3200002A9F000000000000E3746, messageQueue=MessageQueue [topic=JAVA\_MINDMAP\_TOPIC, brokerName=user.local, queueId=5], queueOffset=19]  
[RECEIVE] Consumer:0, ConsumeMessageThread\_3 推送式消费  
Receive New Messages: [MessageExt [queueId=5, storeSize=195, queueOffset=19, sysFlag=0, bornTimestamp=1615619933968, bornHost=/192.168.43.50:56297, storeTimestamp=1615619933969,



```
storeHost=/192.168.43.50:10911,  
msgId=C0A82B3200002A9F000000000000E3746,  
commitLogOffset=931654, bodyCRC=64810935, reconsumeTimes=0,  
preparedTransactionOffset=0,  
toString()=Message{topic='JAVA_MINDMAP_TOPIC', flag=0,  
properties={MIN_OFFSET=0, MAX_OFFSET=20,  
CONSUME_START_TIME=1615619933973,  
UNIQ_KEY=C0A82B326DD918B4AAC24115970B0001,  
CLUSTER=DefaultCluster, WAIT=true, TAGS=TagA}, body=[49],  
transactionId='null']}]
```

- ■ 集群消费 (Clustering)  
集群消费模式下,相同Consumer Group的每个Consumer实例平均分摊消息。消费者默认为集群消费。
- ■ 广播消费 (Broadcasting)  
广播消费模式下, 相同Consumer Group的每个Consumer实例都接收全量的消息。
- ■ 普通顺序消息 (Normal Ordered Message)  
普通顺序消费模式下, 消费者通过同一个消费队列收到的消息是有顺序的, 不同消息队列收到的消息则可能是无顺序的。
- ■ 严格顺序消息 (Strictly Ordered Message)  
严格顺序消息模式下, 消费者收到的所有消息均是有顺序的。
- ■ 消息 (Message)  
消息系统所传输信息的物理载体, 生产和消费数据的最小单位, 每条消息必须属于一个主题。RocketMQ中每个消息拥有唯一的Message ID, 且可以携带具有业务标识的Key。系统提供了通过Message ID和Key查询消息的功能。
- ■ 标签 (Tag)  
为消息设置的标志, 用于同一主题下区分不同类型的消息。来自同一业务单元的消息, 可以根据不同业务目的在同一主题下设置不同标签。标签能够有效地保持代码的清晰度和连贯性, 并优化RocketMQ提供的查询系统。消费者可以根据Tag实现对不同子主题的不同消费逻辑, 实现更好的扩展性。
- 架构(Architecture)  
<https://github.com/apache/rocketmq/blob/master/docs/cn/architecture.md>
  - 技术架构  
RocketMQ架构上主要分为四部分
    - Producer: 消息发布的角色, 支持分布式集群方式部署。Producer通过MQ的负载均衡模块选择相应的Broker集群队列进行消息投递, 投递的过程支持快速失败并且低延迟。【无状态集群】
    - Consumer: 消息消费的角色, 支持分布式集群方式部署。支持以push推, pull拉两种模式对消息进行消费。同时也支持集群方式和广播方式的消费, 它提供实时消息订阅机制, 可以满足大多数用户的需求。【无状态集群】
      - 集群消费模式下, 同一个Consumer Group下, 通过增加Consumer实例的数量来提高并行度, 超过订阅队列数的

Consumer实例无效。

- NameServer: NameServer是一个非常简单的Topic路由注册中心，其角色类似Dubbo中的zookeeper，支持Broker的动态注册与发现。主要包括两个功能：

#Broker管理，NameServer接受Broker集群的注册信息并且保存下来作为路由信息的基本数据。然后提供心跳检测机制，检查Broker是否还存活；

#路由信息管理，每个NameServer将保存关于Broker集群的整个路由信息和用于客户端查询的队列信息。然后Producer和Consumer通过NameServer就可以知道整个Broker集群的路由信息，从而进行消息的投递和消费。【NameServer通常也是集群的方式部署，各实例间相互不进行信息通讯。】Broker是向每一台NameServer注册自己的路由信息，所以每一个NameServer实例上面都保存一份完整的路由信息。当某个NameServer因某种原因下线了，Broker仍然可以向其它NameServer同步其路由信息，Producer,Consumer仍然可以动态感知Broker的路由的信息。【无状态集群】

- 配置文件信息保存在\$HOME/store/config/目录下  
topic保存在该目录下topics.json文件中
- BrokerServer: Broker主要负责消息的存储、投递和查询以及服务高可用保证，为了实现这些功能，Broker包含了多个重要子模块。
  - Remoting Module: 整个Broker的实体，负责处理来自clients端(Producer/Consumer)的请求。
  - Client Manager: 负责管理客户端(Producer/Consumer)和维护Consumer的Topic订阅信息
  - Store Service: 提供方便简单的API接口处理消息存储到物理硬盘和查询功能。
  - HA Service: 高可用服务，提供Master Broker 和 Slave Broker之间的数据同步功能。
  - Index Service: 根据特定的Message key对投递到Broker的消息进行索引服务，以提供消息的快速查询。

## ■ 部署架构

- RocketMQ 网络部署特点
  - NameServer是一个几乎无状态节点，可集群部署，节点之间无任何信息同步。
  - Broker部署相对复杂，Broker分为Master与Slave，一个Master可以对应多个Slave，但是一个Slave只能对应一个Master，【Master与Slave 的对应关系通过指定相同的BrokerName，不同的BrokerId 来定义，BrokerId为0表示Master，非0表示Slave。】Master也可以部署多个。每个Broker与NameServer集群中的所有节点建立长连接，定时注册Topic信息到所有NameServer。注意：当前RocketMQ版本在部署架构上支持一Master多Slave，但只有BrokerId=1的从服务器才会参与消息的读负载。
  - 1、单个Master节点：负载压力非常大，如果宕机的话，数据

可能会丢失

2、多个Master阶段：分摊存储数据，但是没有Slave节点的话，宕机的情况下数据可能会丢失

3、多Master和多Slave节点，同步形式实现主从数据同步，在生产者将消息存放到主再同步到备Broker中才返回ack确认消息投递成功

4、多Master和多Slave节点，异步形式实现主从数据同步，在生产者将消息存放到主，返回ack确认消息投递成功，异步同步到备Broker中，效率高，但是数据可能会丢失

- ■ Producer与NameServer集群中的其中一个节点（随机选择）建立长连接，定期从NameServer获取Topic路由信息，并向提供Topic服务的Broker Master建立长连接，且定时向Master发送心跳。Producer完全无状态，可集群部署。
- ■ Consumer与NameServer集群中的其中一个节点（随机选择）建立长连接，定期从NameServer获取Topic路由信息，并向提供Topic服务的Broker Master、Broker Slave建立长连接，且定时向Master、Slave发送心跳。Consumer既可以从Master订阅消息，也可以从Slave订阅消息，消费者在向Master拉取消息时，Master服务器会根据拉取偏移量与最大偏移量的距离（判断是否读老消息，产生读I/O），以及从服务器是否可读等因素建议下一次是从Master还是Slave拉取。
- 集群工作流程
  - ■ 启动NameServer，NameServer起来后监听端口，等待Broker、Producer、Consumer连上来，相当于一个路由控制中心。
  - ■ Broker启动，跟所有的NameServer保持长连接，定时(30s)发送心跳包。心跳包中包含当前Broker信息(IP+端口等)以及存储的所有Topic信息。注册成功后，NameServer集群中就有Topic跟Broker的映射关系。
  - ■ 收发消息前，先创建Topic，创建Topic时需要指定该Topic要存储在哪些Broker上(同时需指定nameserver)，也可以在发送消息时自动创建Topic。  
创建Topic命令：  

```
sh mqadmin updateTopic -b 192.168.10.150:10911 -n 192.168.10.150:9876 -t JAVA_MINDMAP_TOPIC
```
  - ■ Producer发送消息，启动时先跟NameServer集群中的其中一台建立长连接，并从NameServer中获取当前发送的Topic存在哪些Broker上，轮询从队列列表选择一个队列，然后与队列所在的Broker建立长连接从而向Broker发消息。(每30s向NameServer拉取一次Topic路由信息)
  - 发送消息时,先检查本地topicPublishInfoTable缓存中是否存在Topic信息,若不存在,则从NameServer中更新该Topic的信息(路由信息,MessageQueue列表);  
会检查2次Topic路由信息,第一次若获取失败(topic不存在情况下),会进行第二次创建topic操作,然后再同步信息(都是在客户端进行的)

- org.apache.rocketmq.client.impl.factory.MQClientInstance.this.updateTopicRouteInfoFromNameServer()进行topic信息同步
  - 使用轮训获取消息保存的MessageQueueId
 

```
public MessageQueue selectOneMessageQueue() {
    int index = this.sendWhichQueue.getAndIncrement();
    int pos = Math.abs(index) % this.messageQueueList.size();
    if (pos < 0) {
        pos = 0;
    }
    return (MessageQueue)this.messageQueueList.get(pos);
}
```
  - Consumer跟Producer类似，跟其中一台NameServer建立长连接，获取当前订阅Topic存在哪些Broker上，然后直接跟Broker建立连接通道，开始消费消息。
- 默认端口
  - BrokerServer:10911  
(HA Slave:10912)
  - NameServer:9876  
多个NameServer按;分割
- 设计
  - <https://github.com/apache/rocketmq/blob/master/docs/cn/design.md>
  - 消息存储
    - 消息存储架构  
消息存储架构中主要有三个跟消息存储相关的文件构成:  
CommitLog,ConsumerQueue,IndexFile
    - CommitLog:  
消息主体以及元数据的存储主体，存储Producer端写入的消息主体内容,消息内容不是定长的。单个文件大小默认1G，文件名长度为20位，左边补零，剩余为起始偏移量，比如00000000000000000000代表了第一个文件，起始偏移量为0，文件大小为1G=1073741824；当第一个文件写满了，第二个文件为00000000001073741824，起始偏移量为1073741824，以此类推。消息主要是顺序写入日志文件，当文件满了，写入下一个文件；  
文件保存位置:\$HOME/store/commitlog/\$20\_offsets
    - ll ~/store  
total 16  
-rw-r--r-- 1 user staff 0B 3 12 12:06 abort  
-rw-r--r-- 1 user staff 4.0K 3 12 14:08 checkpoint  
drwxr-xr-x 4 user staff 128B 3 12 13:47 commitlog  
drwxr-xr-x 11 user staff 352B 3 12 14:09 config  
drwxr-xr-x 3 user staff 96B 3 12 13:47  
consumequeue  
drwxr-xr-x 3 user staff 96B 3 12 13:47 index

```

-rw-r--r-- 1 user staff 4B 3 12 12:06 lock
■ ll ~/store/commitlog/
total 448
-rw-r--r-- 1 user staff 1.0G 3 12 13:47
00000000000000000000
-rw-r--r-- 1 user staff 1.0G 3 12 13:47
00000000001073741824
■ ConsumeQueue:
消息消费队列，引入的目的主要是提高消息消费的性能，由于
RocketMQ是基于主题topic的订阅模式，消息消费是针对主题
进行的，如果要遍历commitlog文件中根据topic检索消息是
非常低效的。Consumer即可根据ConsumeQueue来查找待
消费的消息。其中，ConsumeQueue（逻辑消费队列）作为
消费消息的索引，保存了指定Topic下的队列消息在
CommitLog中的起始物理偏移量offset，消息大小size和消息
Tag的HashCode值。consumequeue文件可以看成是基于
topic的commitlog索引文件，故【consumequeue文件夹的
组织方式如下：topic/queue/file三层组织结构，具体存储路
径为：
$HOME/store/consumequeue/{topic}/{queueId}/{fileNam
e},queueId为从0开始的序列,filename为20位的偏移量序
列】。同样consumequeue文件采取定长设计，每一个条目
共20个字节，分别为8字节的commitlog物理偏移量、4字节
的消息长度、8字节tag hashCode，单个文件由30W个条目组
成，可以像数组一样随机访问每一个条目，每个
ConsumeQueue文件大小约5.72M；
【ConsumerQueue大小(queue的数量)由Topic配置,默认大
小为8:
sh mqbroker -n localhost:9876 -p查看配置信息:
defaultTopicQueueNums=8 (包括
readQueueNums,writeQueueNums)
autoCreateTopicEnable=true】
■ ~/store/consumequeue
total 0
drwxr-xr-x 6 user staff 192B 3 12 13:47 TopicTest
~/store/consumequeue/TopicTest
total 0
drwxr-xr-x 3 user staff 96B 3 12 13:47 0
drwxr-xr-x 3 user staff 96B 3 12 13:47 1
drwxr-xr-x 3 user staff 96B 3 12 13:47 2
drwxr-xr-x 3 user staff 96B 3 12 13:47 3
ll ~/store/consumequeue/TopicTest/0
total 11720
-rw-r--r-- 1 user staff 5.7M 3 12 13:47
00000000000000000000

```

- IndexFile:

IndexFile（索引文件）提供了一种可以通过key或时间区间来查询消息的方法。【Index文件的存储位置是：\$HOME\store\index\${fileName}，文件名fileName是以创建时的时间戳命名的，固定的单个IndexFile文件大小约为400M，一个IndexFile可以保存 2000W个索引】，IndexFile的底层存储设计为在文件系统中实现HashMap结构，故rocketmq的索引文件其底层实现为hash索引。

- ll ~/store/index

- total 28072

- rw-r--r-- 1 user staff 401M 3 12 13:47

- 20210312134728376

- RocketMQ采用的是混合型的存储结构，即为Broker单个实例下所有的队列共用一个日志数据文件（即为CommitLog）来存储。RocketMQ的混合型存储结构(多个Topic的消息实体内容都存储于一个CommitLog中)针对Producer和Consumer分别采用了数据和索引部分相分离的存储结构，Producer发送消息至Broker端，然后Broker端使用同步或者异步的方式对消息刷盘持久化，保存至CommitLog中。只要消息被刷盘持久化至磁盘文件CommitLog中，那么Producer发送的消息就不会丢失。正因为如此，Consumer也就肯定有机会去消费这条消息。当无法拉取到消息后，可以等下一次消息拉取，同时服务端也支持长轮询模式，如果一个消息拉取请求未拉取到消息，Broker允许等待30s的时间，只要这段时间内有新消息到达，将直接返回给消费端。这里，RocketMQ的具体做法是，使用Broker端的后台服务线程—ReputMessageService不停地分发请求并异步构建ConsumeQueue（逻辑消费队列）和IndexFile（索引文件）数据。

- 消息默认保存时长: 72小时(3天),可在Broker端配置:  
fileReservedTime 72 以小时计算的文件保留时间

- 页缓存与内存映射

- 消息刷盘

- (1) 同步刷盘：只有在消息真正持久化至磁盘后RocketMQ的Broker端才会真正返回给Producer端一个成功的ACK响应。同步刷盘对MQ消息可靠性来说是一种不错的保障，但是性能上会有较大影响，一般适用于金融业务应用该模式较多。
  - (2) 异步刷盘：能够充分利用OS的PageCache的优势，只要消息写入PageCache即可将成功的ACK返回给Producer端。消息刷盘采用后台异步线程提交的方式进行，降低了读写延迟，提高了MQ的性能和吞吐量。

- 通信机制

(基于Netty)

- RocketMQ消息队列集群主要包括NameServer、Broker(Master/Slave)、Producer、Consumer4个角色，基本通讯流程如下：

- (1) Broker启动后需要完成一次将自己注册至NameServer的操作；随后每隔30s时间定时向NameServer上报Topic路由信息。
- (2) 消息生产者Producer作为客户端发送消息时候，需要根据消息的Topic从本地缓存的TopicPublishInfoTable获取路由信息。如果没有则更新路由信息会从NameServer上重新拉取，同时Producer会默认每隔30s向NameServer拉取一次路由信息。
- (3) 消息生产者Producer根据2) 中获取的路由信息选择一个队列（MessageQueue）进行消息发送；Broker作为消息的接收者接收消息并落盘存储。
- (4) 消息消费者Consumer根据2) 中获取的路由信息，并再完成客户端的负载均衡后，选择其中的某一个或者某几个消息队列来拉取消息并进行消费。
- 从上面1) ~3) 中可以看出在消息生产者, Broker和NameServer之间都会发生通信（这里只说了MQ的部分通信），因此如何设计一个良好的网络通信模块在MQ中至关重要，它将决定RocketMQ集群整体的消息传输能力与最终的性能。

rocketmq-remoting 模块是 RocketMQ消息队列中负责网络通信的模块，它几乎被其他所有需要网络通信的模块（诸如rocketmq-client、rocketmq-broker、rocketmq-namesrv）所依赖和引用。为了实现客户端与服务器之间高效的数据请求与接收，RocketMQ消息队列自定义了通信协议并在Netty的基础之上扩展了通信模块。

#### ■ 消息过滤

- RocketMQ分布式消息队列的消息过滤方式有别于其它MQ中间件，是在Consumer端订阅消息时再做消息过滤的。RocketMQ这么做是在于其Producer端写入消息和Consumer端订阅消息采用分离存储的机制来实现的，Consumer端订阅消息是需要通过ConsumeQueue这个消息消费的逻辑队列拿到一个索引，然后再从CommitLog里面读取真正的消息实体内容，所以说到底也是还绕不开其存储结构。其ConsumeQueue的存储结构如下，可以看到其中有8个字节存储的Message Tag的哈希值，基于Tag的消息过滤正式基于这个字段值的。
- 主要支持2种的过滤方式
  - (1) Tag过滤方式：Consumer端在订阅消息时除了指定Topic还可以指定TAG，如果一个消息有多个TAG，可以用||分隔。其中，Consumer端会将这个订阅请求构建成一个SubscriptionData，发送一个Pull消息的请求给Broker端。Broker端从RocketMQ的文件存储层—Store读取数据之前，会用这些数据先构建一个MessageFilter，然后传给Store。Store从ConsumeQueue读取到一条记录后，会用它记录的消息tag hash值去做过滤，由于在服务端只是根据hashcode进行判断，无法精确对tag原始字符串进行过滤，故在消息消费端拉取到消息后，还需要对消息的原始tag字符串进行比对，如果不同，则丢弃该消息，不进行消息消费。
  - (2) SQL92的过滤方式：这种方式的大致做法和上面的Tag过滤方式一样，只是在Store层的具体过滤过程不太一样，真正的

SQL expression 的构建和执行由rocketmq-filter模块负责的。每次过滤都去执行SQL表达式会影响效率，所以RocketMQ使用了BloomFilter避免了每次都去执行。SQL92的表达式上下文为消息的属性。

- 负载均衡

RocketMQ中的负载均衡都在Client端完成，具体来说的话，主要可以分为Producer端发送消息时候的负载均衡和Consumer端订阅消息的负载均衡

- 事务消息

Apache RocketMQ在4.3.0版中已经支持分布式事务消息，这里RocketMQ采用了2PC的思想来实现了提交事务消息，同时增加一个补偿逻辑来处理二阶段超时或者失败的消息

- 消息查询

RocketMQ支持按照下面两种维度（“按照Message Id查询消息”、“按照Message Key查询消息”）进行消息查询

- 按照MessageId查询消息

RocketMQ中的MessageId的长度总共有16字节，其中包含了消息存储主机地址（IP地址和端口），消息Commit Log offset。“按照MessageId查询消息”在RocketMQ中具体做法是：Client端从MessageId中解析出Broker的地址（IP地址和端口）和Commit Log的偏移地址后封装成一个RPC请求后通过Remoting通信层发送（业务请求码：VIEW\_MESSAGE\_BY\_ID）。Broker端走的是QueryMessageProcessor，读取消息的过程用其中的commitLog offset 和 size 去commitLog 中找到真正的记录并解析成一个完整的消息返回。

- 按照Message Key查询消息

“按照Message Key查询消息”，主要是基于RocketMQ的IndexFile索引文件来实现的。RocketMQ的索引文件逻辑结构，类似JDK中HashMap的实现。

“按照Message Key查询消息”的方式，RocketMQ的具体做法是，主要通过Broker端的QueryMessageProcessor业务处理器来查询，读取消息的过程就是用topic和key找到IndexFile索引文件中的一条记录，根据其中的commitLog offset从CommitLog文件中读取消息的实体内容。

- 特性

<https://github.com/apache/rocketmq/blob/master/docs/cn/features.md>

- 订阅与发布

消息的发布是指某个生产者向某个topic发送消息；消息的订阅是指某个消费者关注了某个topic中带有某些tag的消息，进而从该topic消费数据。

- 消息顺序

消息有序指的是一类消息消费时，能按照发送的顺序来消费。例如：一个订单产生了三条消息分别是订单创建、订单付款、订单完成。消费时要按照这个顺序消费才能有意义，但是同时订单之间是可以并行消费的。RocketMQ可以严格的保证消息有序。

顺序消息分为全局顺序消息与分区顺序消息，全局顺序是指某个Topic下的所



有消息都要保证顺序；部分顺序消息只要保证每一组消息被顺序消费即可。

- 全局顺序 对于指定的一个 Topic，所有消息按照严格的先入先出（FIFO）的顺序进行发布和消费。适用场景：性能要求不高，所有的消息严格按照 FIFO 原则进行消息发布和消费的场景
- 分区顺序 对于指定的一个 Topic，所有消息根据 sharding key 进行区块分区。同一个分区内的消息按照严格的 FIFO 顺序进行发布和消费。Sharding key 是顺序消息中用来区分不同分区的关键字段，和普通消息的 Key 是完全不同的概念。适用场景：性能要求高，以 sharding key 作为分区字段，在同一个区块中严格的按照 FIFO 原则进行消息发布和消费的场景。
- 消息过滤  
RocketMQ的消费者可以根据Tag进行消息过滤，也支持自定义属性过滤。消息过滤目前是在Broker端实现的，优点是减少了对于Consumer无用消息的网络传输，缺点是增加了Broker的负担、而且实现相对复杂。
- 消息可靠性  
RocketMQ支持消息的高可靠，影响消息可靠性的几种情况：
  - 1)Broker非正常关闭
  - 2)Broker异常Crash
  - 3)OS Crash
  - 4)机器掉电，但是能立即恢复供电情况
  - 5)机器无法开机（可能是cpu、主板、内存等关键设备损坏）
  - 6)磁盘设备损坏
- 1)、2)、3)、4) 四种情况都属于硬件资源可立即恢复情况，RocketMQ在这四种情况下能保证消息不丢，或者丢失少量数据（依赖刷盘方式是同步还是异步）。
- 5)、6)属于单点故障，且无法恢复，一旦发生，在此单点上的消息全部丢失。RocketMQ在这两种情况下，通过异步复制，可保证99%的消息不丢，但是仍然会有极少量的消息可能丢失。通过同步双写技术可以完全避免单点，同步双写势必会影响性能，适合对消息可靠性要求极高的场合，例如与Money相关的应用。注：RocketMQ从3.0版本开始支持同步双写。
- 至少一次  
至少一次(At least Once)指每个消息必须投递一次。Consumer先Pull消息到本地，消费完成后，才向服务器返回ack，如果没有消费一定不会ack消息，所以RocketMQ可以很好的支持此特性。
- 回溯消费  
回溯消费是指Consumer已经消费成功的消息，由于业务上需求需要重新消费，要支持此功能，Broker在向Consumer投递成功消息后，消息仍然需要保留。并且重新消费一般是按照时间维度，例如由于Consumer系统故障，恢复后需要重新消费1小时前的数据，那么Broker要提供一种机制，可以按照时间维度来回退消费进度。RocketMQ支持按照时间回溯消费，时间维度精确到毫秒。
- 事务消息  
RocketMQ事务消息（Transactional Message）是指应用本地事务和发送消息操作可以被定义到全局事务中，要么同时成功，要么同时失败。RocketMQ的事务消息提供类似 X/Open XA 的分布事务功能，通过事务消息能达到分布式事务的最终一致。
- 定时消息

定时消息（延迟队列）是指消息发送到broker后，不会立即被消费，等待特定时间投递给真正的topic。broker有配置项messageDelayLevel，默认值为“1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h”，18个level。可以配置自定义messageDelayLevel。注意，messageDelayLevel是broker的属性，不属于某个topic。发消息时，设置delayLevel等级即可：msg.setDelayLevel(level)。level有以下三种情况：  
level == 0，消息为非延迟消息  
1 <= level <= maxLevel，消息延迟特定时间，例如level==1，延迟1s  
level > maxLevel，则level== maxLevel，例如level==20，延迟2h  
定时消息会暂存在名为SCHEDULE\_TOPIC\_XXXX的topic中，并根据delayTimeLevel存入特定的queue，queueId = delayTimeLevel - 1，即一个queue只存相同延迟的消息，保证具有相同发送延迟的消息能够顺序消费。broker会调度地消费SCHEDULE\_TOPIC\_XXXX，将消息写入真实的topic。需要注意的是，定时消息会在第一次写入和调度写入真实topic时都会计数，因此发送数量、tps都会变高。

#### ■ ■ 消息重试

Consumer消费消息失败后，要提供一种重试机制，令消息再消费一次。

Consumer消费消息失败通常可以认为有以下几种情况：

\*由于消息本身的原因，例如反序列化失败，消息数据本身无法处理（例如话费充值，当前消息的手机号被注销，无法充值）等。这种错误通常需要跳过这条消息，再消费其它消息，而这条失败的消息即使立刻重试消费，99%也不成功，所以最好提供一种定时重试机制，即过10秒后再重试。

\*由于依赖的下游应用服务不可用，例如db连接不可用，外系统网络不可达等。遇到这种错误，即使跳过当前失败的消息，消费其他消息同样也会报错。这种情况建议应用sleep 30s，再消费下一条消息，这样可以减轻Broker重试消息的压力。

RocketMQ会为每个消费组都设置一个Topic名称为

“%RETRY%+consumerGroup”的重试队列（这里需要注意的是，这个Topic的重试队列是针对消费组，而不是针对每个Topic设置的），用于暂时保存因为各种异常而导致Consumer端无法消费的消息。考虑到异常恢复起来需要一些时间，会为重试队列设置多个重试级别，每个重试级别都有与之对应的重新投递延时，重试次数越多投递延时就越大。RocketMQ对于重试消息的处理是先保存至Topic名称为“SCHEDULE\_TOPIC\_XXXX”的延迟队列中，后台定时任务按照对应的时间进行Delay后重新保存至“%RETRY%+consumerGroup”的重试队列中。

#### ■ ■ 消息重投

生产者在发送消息时，同步消息失败会重投，异步消息有重试，oneway没有任何保证。消息重投保证消息尽可能发送成功、不丢失，但可能会造成消息重复，消息重复在RocketMQ中是无法避免的问题。消息重复在一般情况下不会发生，当出现消息量大、网络抖动，消息重复就会是大概率事件。另外，生产者主动重发、consumer负载变化也会导致重复消息。如下方法可以设置消息重试策略：

\*retryTimesWhenSendFailed:同步发送失败重投次数，默认为2，因此生产者会最多尝试发送retryTimesWhenSendFailed + 1次。不会选择上次失败的broker，尝试向其他broker发送，最大程度保证消息不丢。超过重投次数，抛出异常，由客户端保证消息不丢。当出现RemotingException、

MQClientException和部分MQBrokerException时会重投。

\*retryTimesWhenSendAsyncFailed:异步发送失败重试次数，异步重试不会选择其他broker，仅在同一个broker上做重试，不保证消息不丢。

\*retryAnotherBrokerWhenNotStoreOK:消息刷盘（主或备）超时或slave不可用（返回状态非SEND\_OK），是否尝试发送到其他broker，默认false。十分重要消息可以开启。

#### ■ 11.流量控制

生产者流控，因为broker处理能力达到瓶颈；消费者流控，因为消费能力达到瓶颈。

##### ■ 生产者流控：

\*commitLog文件被锁时间超过osPageCacheBusyTimeOutMills时，参数默认为1000ms，返回流控。

\*如果开启transientStorePoolEnable == true，且broker为异步刷盘的主机，且transientStorePool中资源不足，拒绝当前send请求，返回流控。

\*broker每隔10ms检查send请求队列头部请求的等待时间，如果超过waitTimeMillsInSendQueue，默认200ms，拒绝当前send请求，返回流控。

\*broker通过拒绝send 请求方式实现流量控制。

注意，生产者流控，不会尝试消息重投。

##### ■ 消费者流控：

\*消费者本地缓存消息数超过pullThresholdForQueue时，默认1000。

\*消费者本地缓存消息大小超过pullThresholdSizeForQueue时，默认100MB。

\*消费者本地缓存消息跨度超过consumeConcurrentlyMaxSpan时，默认2000。

消费者流控的结果是降低拉取频率。

#### ■ 12.死信队列

死信队列用于处理无法被正常消费的消息。当一条消息初次消费失败，消息队列会自动进行消息重试；达到最大重试次数后，若消费依然失败，则表明消费者在正常情况下无法正确地消费该消息，此时，消息队列不会立刻将消息丢弃，而是将其发送到该消费者对应的特殊队列中。

RocketMQ将这种正常情况下无法被消费的消息称为死信消息（Dead-Letter Message），将存储死信消息的特殊队列称为死信队列（Dead-Letter Queue）。

在RocketMQ中，可以通过使用console控制台对死信队列中的消息进行重发来使得消费者实例再次进行消费。

#### ■ 用法/样例

[https://github.com/apache/rocketmq/blob/master/docs/cn/RocketMQ\\_Example.md](https://github.com/apache/rocketmq/blob/master/docs/cn/RocketMQ_Example.md)

##### ■ RocketMQ-Startup (需设置JAVA\_HOME)

##### ■ 1.启动Nameserver,默认端口:9876

```
nohup sh bin/mqnamesrv &
tail -f ~/logs/rocketmqlogs/namesrv.log
The Name Server boot success...
```

- 若提示JAVA\_HOME报错,但实际JAVA\_HOME存在,  
则需要export JAVA\_HOME:

```

vim /etc/profile
#在尾部添加
JAVA_HOME=/data/jdk1.8.0_201
CLASSPATH=.:$JAVA_HOME/lib/tools.jar:$JAVA_HOME/lib/rt.jar~
PATH=$JAVA_HOME/bin:$HOME/bin:$HOME/.local/bin:$PATH
#这一句一定需要
export JAVA_HOME
source /etc/profile
#命令执行完成需要退出重新登录
- 2.启动BrokerServer,默认端口:10911(HA Slave:10912)
nohup sh bin/mqbroker -n localhost:9876 &
tail -f ~/logs/rocketmqlogs/broker.log
The broker[%s, 172.30.30.233:10911] boot success...
- 3.测试消息发送和接收
- 生产者示例:
export NAMESRV_ADDR=localhost:9876
sh bin/tools.sh org.apache.rocketmq.example.quickstart.Producer
SendResult [sendStatus=SEND_OK, msgId= ...
- 消费者示例
sh bin/tools.sh org.apache.rocketmq.example.quickstart.Consumer
ConsumeMessageThread_%d Receive New Messages: [MessageExt...
- 4.关闭服务
- 关闭Broker
sh bin/mqshutdown broker
The mqbroker(36695) is running...
Send shutdown request to mqbroker(36695) OK
- 关闭NameServer
sh bin/mqshutdown namesrv
The mqnamesrv(36664) is running...
Send shutdown request to mqnamesrv(36664) OK
- 管理控制台
- mqadmin:
mqadmin statsAll: 所有topic及consumer消费情况
http://rocketmq.apache.org/docs/cli-admin-tool/
https://www.cnblogs.com/zyguo/p/4962425.html

```

- MAVEN依赖
- 基本样例
  - 使用RocketMQ发送三种类型的消息：同步消息、异步消息和单向消息。其中前两种消息是可靠的，因为会有发送是否成功的应答。
- 顺序消息
  - 消息有序指的是可以按照消息的发送顺序来消费(FIFO)。RocketMQ可以严格的保证消息有序，可以分为分区有序或者全局有序。  
顺序消费的原理解析，在默认的情况下消息发送会采取Round Robin轮询方式把消息发送到不同的queue(分区队列)；而消费消息的时候从多个queue上拉取消息，这种情况发送和消费是不能保证顺序。但是如果控制发送的顺序

消息只依次发送到同一个queue中，消费的时候只从这个queue上依次拉取，则就保证了顺序。当发送和消费参与的queue只有一个，则是全局有序；如果多个queue参与，则为分区有序，即相对每个queue，消息都是有序的。

#### ■ 延时消息

- Message message = new Message("TestTopic", ("Hello scheduled message " + i).getBytes());  
// 设置延时等级3,这个消息将在10s之后发送(现在只支持固定的几个时间,详看delayTimeLevel)  
message.setDelayTimeLevel(3);  
// 发送消息  
producer.send(message);  
最大支持18个Level,最大的2小时;  
private String messageDelayLevel = "1s 5s 10s 30s 1m 2m 3m 4m 5m 6m 7m 8m 9m 10m 20m 30m 1h 2h";  
现在RocketMq并不支持任意时间的延时，需要设置几个固定的延时等级;

#### ■ 批量消息

- 批量发送消息能显著提高传递小消息的性能。限制是这些批量消息应该有相同的topic，相同的waitStoreMsgOK，而且不能是延时消息。此外，这一批消息的总大小不应超过4MB。

#### ■ 过滤消息

- Tag过滤  
DefaultMQPushConsumer consumer = new  
DefaultMQPushConsumer("CID\_EXAMPLE");  
consumer.subscribe("TOPIC", "TAGA || TAGB || TAGC");
- SQL92过滤:  
只有使用push模式的消费者才能用使用SQL92标准的sql语句，接口如下：  
public void subscribe(finalString topic, final MessageSelector  
messageSelector)  
消费者:  
consumer.subscribe("TopicTest", MessageSelector.bySql("a between 0  
and 3"));

#### ■ 事务消息

- 事务消息共有三种状态，提交状态、回滚状态、中间状态：  
#TransactionStatus.CommitTransaction: 提交事务，它允许消费者消费此消息。  
#TransactionStatus.RollbackTransaction: 回滚事务，它代表该消息将被删除，不允许被消费。  
#TransactionStatus.Unknown: 中间状态，它代表需要检查消息队列来确定状态。

#### ■ 相关问题:

- MQClientException: No route info of this topic,  
解决:检查broker是否设置了自动创建topic,新版本默认自动创建topic  
./mqbroker -n 127.0.0.1:9876 -p -p是查看配置信息  
查看autoCreateTopicEnable的值是否为true

```
nohup sh mqbroker -n 192.168.180.133:9876
```

```
autoCreateTopicEnable=true
```

解决:<http://rocketmq.apache.org/docs/faq/>:

Producer complains "No Topic Route Info", how to diagnose?

#手工创建topic(需指定broker,nameServer)

```
sh mqadmin updateTopic -b 192.168.10.150:10911 -n
```

```
192.168.10.150:9876 -t JAVA_MINDMAP_TOPIC
```

- 2.Producer同步发送消息时,首次发送消息可能会超时,导致失败
- 3.多个Consumer需设置instanceName,否则会报The consumer group has been created before, specify another name please.错误

解决:

a. consumer.setInstanceName("consumer-instance-1");

b. instanceName 默认为:

```
instanceName = System.getProperty("rocketmq.client.name",  
"DEFAULT");
```

#### ■ 最佳实践

[https://github.com/apache/rocketmq/blob/master/docs/cn/best\\_practice.md](https://github.com/apache/rocketmq/blob/master/docs/cn/best_practice.md)

- Broker配置(包括Broker角色,刷盘类型,消息保存时长,过期消息删除时间等)  
生产者配置;  
消费者配置;  
JVM配置;  
Linux内核配置;
- 消费速度慢的处理方式
  - 1 提高消费并行度
    - a. 增加消费队列和消费者实例  
同一个 ConsumerGroup 下, 通过增加 Consumer 实例数量来提高并行度 (需要注意的是超过订阅队列数的 Consumer 实例无效)。可以通过加机器, 或者在已有机器启动多个进程的方式。
    - b. 提高单个 Consumer 的消费并行线程, 通过修改参数 consumeThreadMin、consumeThreadMax实现。
  - 2 批量方式消费  
某些业务流程如果支持批量方式消费, 则可以很大程度上提高消费吞吐量, 例如订单扣款类应用, 一次处理一个订单耗时 1 s, 一次处理 10 个订单可能也只耗时 2 s, 这样即可大幅度提高消费的吞吐量, 通过设置 consumer的 consumeMessageBatchMaxSize 这个参数, 默认是 1, 即一次只消费一条消息, 例如设置为 N, 那么每次消费的消息数小于等于 N。
  - 3 跳过非重要消息
  - 4 优化每条消息消费过程

#### ■ 与kafka比较

<http://rocketmq.apache.org/docs/motivation/>

- <http://rocketmq.apache.org/rocketmq/how-to-support-more-queues-in-rocketmq/>

**RabbitMQ**

**ActiveMQ**

**服务默认端口**

---

**Mysql: 3306**

**Oracle: 1521**

**ProgreSQL: 5432**

**Redis: 6379**

**Mongo: 27017**

**zookeeper: 2181**

**kafka: 9092**

**Tomcat: 8080**

**Windows RDP: 3389**

**分布式框架**

---

**SpringCloud**

- Springboot

- <https://docs.spring.io/spring-boot/docs/2.3.1.RELEASE/reference/html/>

- 配置

- org.springframework.boot.context.config

- .ConfigFileApplicationListener

- 配置项 (1.5+和2.0+版本配置名称有区别)

- <https://docs.spring.io/spring-boot/docs/2.3.1.RELEASE/reference/html/appendix-application-properties.html>

- 核心配置

- logging.config 日志记录配置文件的位置。例如，用于logback的

- `classpath: logback.xml`。

- logging.level.\*

- 日志级别严重性映射。例

- 如，`logging.level.org.springframework = DEBUG`。

- spring.application.name 应用名称

- spring.autoconfigure.exclude 要排除的自动配置类。

同 SpringApplication(exclude = [])

- spring.banner.image.location  
banner图片文件的位置（也可以使用jpg或png）  
classpath:banner.gif
- spring.banner.location banner文字资源位置  
默认: classpath:banner.txt  
org.springframework.boot.SpringApplicationBannerPrinter
- spring.config.location 替换默认设置的配置文件位置,多个路径按,分割  
默认为: classpath:/,classpath:/config/,file:/,file:/config/  
匹配时取反序
- spring.config.name 替换默认配置文件名  
默认为application  
ConfigFileApplicationListener
- spring.profiles  
用逗号分隔的概要文件表达式列表，至少要匹配一个概要文件表达式才能包含在内的文档。
- spring.profiles.active 以逗号分隔的活动配置文件列表。可以被命令行开关覆盖
- spring.profiles.include  
无条件激活指定的逗号分隔的配置文件列表（如果使用YAML，则激活配置文件列表）
- JSON属性
  - spring.jackson.date-format 日期格式字符串或标准日期格式类名称。例如，yyyy-MM-dd HH:mm:ss。
  - spring.jackson.time-zone 格式化日期时使用的时区。例如，“America / Los\_Angeles”或“GMT + 10”
- 数据属性
  - spring.data.elasticsearch.client.reactive.endpoints  
要连接到的Elasticsearch端点的逗号分隔列表
  - spring.data.elasticsearch.client.reactive.password  
ES连接凭证密码
  - spring.data.elasticsearch.client.reactive.username  
ES连接用户名
  - spring.data.mongodb.authentication-database mongo认证数据库名称
  - spring.data.mongodb.auto-index-creation MongoDB是否创建索引
  - spring.data.mongodb.database  
MongoDB数据库名称
  - Neo4jredis
  - solr
- 服务器属性
  - server.address 服务绑定的地址
  - server.port 服务器http端口



- `server.compression.enabled` 是否启用响应压缩。 `false`
  - `server.compression.mime-types` 以逗号分隔的应压缩的MIME类型列表。
  - `server.error.path` 错误控制器的路径。
- `application.properties/.xml`
  - 包含应用相关的配置项，被应用程序上下文(ApplicationContext)加载，会被后续的配置文件覆盖掉相同属性的配置
- `bootstrap.properties`
  - 包含引导相关的外部配置项，被引导上下文(BootstrapContext)加载，BootstrapContext是ApplicationContext的父上下文，二者共用Environment且优先于application.properties加载。  
如: 依赖的外部的Zookeeper配置，  
`spring.application.name`  
`server.port`等配置
- `application-[profile].properties`
  - 分环境应用程序上下文配置文件，当设置`spring.profiles.active=xxx`时，会加载`application-xxx.properties`
- `bootstrap-[profile].properties`
- `application.yml`
  - 与`application.properties` 作用相同
- `bootstrap.yml`
  - 与`bootstrap.properties` 作用相同
- 命令行传入变量
  - 命令行传入变量可覆盖配置文件中的变量
- 系统环境变量
- 加载顺序
  - 不同名称配置文件加载顺序
    - `bootstrap`
    - `bootstrap-[profile]`
    - `application`
    - `application-[profile]`
    - 后边加载的配置会覆盖前边的配置
  - 同名称不同后缀加载顺序
    - `0 = "properties"`
    - `1 = "xml"`
    - `2 = "yml"`
    - `3 = "yaml"`
- 默认配置文件加载路径
  - `classpath:/,classpath:/config/,file:/,file:/config/`  
加载顺序为: 以上顺序的返序  
`ConfigFileApplicationListener#getSearchNames()`

- 特性: <https://docs.spring.io/spring-boot/docs/2.3.1.RELEASE/reference/html/spring-boot-features.html>
- Maven依赖

- SpringBoot依赖有2种方式

- 继承spring-boot-starter-parent

```
org.springframework.boot
spring-boot-starter-parent
2.1.3.RELEASE
```

该方法可以使用properties覆盖内部依赖版本，如：

```
<spring-data-releasetrain.version>Fowler-SR2</spring-data-releasetrain.version>
```

- 在DependencyManagement中scope=import 依赖spring-boot-dependency pom

```
org.springframework.data
spring-data-releasetrain
Fowler-SR2
import
pom
```

```
org.springframework.boot
spring-boot-dependencies
2.1.3.RELEASE
pom
import
```

该方式不能使用properties形式覆盖原始依赖版本。要达到同样效果，需要在dependencyMangement中spring-boot-dependencies前添加pom依赖

- <--SpringBoot依赖-->

```
org.springframework.boot
spring-boot-dependencies
${spring-boot.version}
pom
import
```

- - org.springframework.cloud
  - spring-cloud-dependencies
  - \${spring-cloud.version}
  - pom
  - import
- 所有springboot提供的starter 都以spring-boot-starter-[xx]开头
- server.port使用依赖
  - - org.springframework.cloud
    - spring-cloud-context
- 主方法
  - public static void main(String[] args) {
   
SpringApplication.run(Example.class, args);
   
}
  - 自定义SpringApplication
   
new
   
SpringApplication(MySpringConfiguration.class).bannerMode(Banner.Mode.OFF)
   
.run(args);
  - 在启动期间运行的任务应由CommandLineRunner和ApplicationRunner组件执行，而不是使用Spring组件生命周期回调（例如）@PostConstruct。
  - 访问应用程序参数:
   
如果您需要访问传递给的应用程序参数，则SpringApplication.run(...)可以注入org.springframework.boot.ApplicationArgumentsBean。该ApplicationArguments接口提供对原始String[]参数以及已解析option和non-option参数的访问
   
@Component
   
public class MyBean {
   
@Autowired
   
public MyBean(ApplicationArguments args) {
   
boolean debug = args.containsOption("debug");
   
List files = args.getNonOptionArgs();
   
// if run with "--debug logfile.txt" debug=true, files=["logfile.txt"]
   
}
   
}
  - 使用ApplicationRunner或CommandLineRunner
    - 如果启动后需要运行一些特定的代码SpringApplication，则可以实现ApplicationRunner或CommandLineRunner接口。这两个接口以相同的方式工作，并提供一个单一的run方法，该方法在SpringApplication.run(...)完成之前(refreshContext()后)就被调用。

这些CommandLineRunner接口提供了作为简单字符串数组的应用程序参数访问，而则ApplicationRunner使用ApplicationArguments了前面讨论的接口。以下示例显示了一个CommandLineRunnerwith run方法@Component

```
public class MyBean implements CommandLineRunner {  
    public void run(String... args) {  
        // Do something...  
    }  
}
```

- org.springframework.boot.SpringApplication#afterRefresh  
org.springframework.boot.SpringApplication#callRunners(ApplicationContext context, ApplicationArguments args)

先执行ApplicationRunner,再执行CommandLineRunner

- private void callRunners(ApplicationContext context, ApplicationArguments args) {  
 List runners = new ArrayList();  
 runners.addAll(context.getBeansOfType(ApplicationRunner.class).values());  
 runners.addAll(context.getBeansOfType(CommandLineRunner.class).values());  
 AnnotationAwareOrderComparator.sort(runners);  
 for (Object runner : new LinkedHashSet(runners)) {  
 if (runner instanceof ApplicationRunner) {  
 callRunner((ApplicationRunner) runner, args);  
 }  
 if (runner instanceof CommandLineRunner) {  
 callRunner((CommandLineRunner) runner, args);  
 }  
 }  
}

- 外部化配置

- 配置随机值

RandomValuePropertySource是用于注射的随机值（例如，进入机密或试验例）是有用的。它可以产生整数， longs， uuid或字符串 (org.springframework.boot.context.config.RandomValuePropertySource)

- my.secret=\${random.value}  
my.number=\${random.int}  
my.bignumber=\${random.long}  
my.uuid=\${random.uuid}  
my.number.less.than.ten=\${random.int(10)}  
my.number.in.range=\${random.int[1024,65536]}

- 访问命令行属性

- 默认情况下， SpringApplication将所有命令行选项参数（即以开头的参数--， 例如--server.port=9000）转换为a property并将其添加到Spring Environment。如前所述， 命令行属性始终优先于其他属

性源。

如果您不想将命令行属性添加到中Environment，则可以使用禁用它们SpringApplication.setAddCommandLineProperties(false)。

- Application Property Files
- YAML Shortcomings
  - 无法使用@PropertySource注解加载YAML文件
- org.springframework.context.Aware 接口
  - 可实现相关\*Aware接口，来获取相关配置或bean  
或使用@Autowired注入  
EnvironmentAware 获取Environment  
ApplicationContextAware 获取ApplicationContext  
BootstrapContextAware 获取BootstrapContext
- 注解  
org.springframework.boot.autoconfigure
  - @SpringBootApplication 声明为一个SpringBoot应用  
默认会加载DataSourceAutoConfiguration.class
    - 排除自动配置的类:
- @SpringBootApplication(exclude = {DataSourceAutoConfiguration.class})
- @SpringBootApplication(excludeName = "org.xx.xx")指定完整限定名
- 使用spring.autoconfigure.exclude属性来控制要排除的自动配置类的列表。
  - @SpringBootApplication注释可用于启用这三个功能，即：  
@EnableAutoConfiguration： 启用Spring Boot的自动配置机制  
@ComponentScan： 启用@Component(@Controller,@Service,@Repository)对应用程序所在的软件包的扫描（请参阅最佳实践）  
@Configuration： 允许在上下文中注册额外的bean或导入其他配置类
    - @EnableAutoConfiguration： 启用Spring Boot的自动配置机制， 自动加载,注入Spring配置
    - @ComponentScan(basePackages = {},) 指定Spring组件的扫描目录，包括扫描@Configuration
  - 不指定时，默认扫描Application所在包下的所有路径
    - Config
      - @Configuration 声明自动配置加载的注解
  - 由@Configuration的类中声明了一个或多个由@Bean声明的方法;加载配置文件@Value
    - @ConfigurationProperties(prefix = "")
  - 将Spring配置文件属性绑定到JavaBean中
- 需使用@Component加入Bean扫描
- 在扫描不到时，在任何  
@Configuration上使用@EnableConfigurationProperties(ValidatorConfig.class) 声明要自动配置的类
- 在@Bean上扫描，可从Environment中的配置注册到bean中；(注入Environment类)  
prefix: 配置属性前缀,按.分割，支持属性嵌套；  
Spring Boot提供了绑定@ConfigurationProperties类型并将其注册为Bean的基础架构；  
有时，带注释的类@ConfigurationProperties可能不适合扫描，例如，如果您正在开发

自己的自动配置或想要有条件地启用它们。在这些情况下，请使用  
@EnableConfigurationProperties注释指定要处理的类型列表。可以在任何  
@Configuration类上完成此操作；

验证:

```
@ConfigurationProperties(prefix="acme")
@Validated
public class AcmeProperties{
    @NotNull
    private InetAddress remoteAddress;
}
```

可使用@Validated添加验证

- 依赖:

```
org.springframework.boot
spring-boot-configuration-processor
```

- @Component

```
@ConfigurationProperties("acme")
public class AcmeProperties {}
```

或

```
@Configuration
@EnableConfigurationProperties(AcmeProperties.class)
@ConfigurationProperties("acme")
public class AcmeProperties {}
```

- 要使用配置属性扫描，请将@ConfigurationPropertiesScan注释添加到您的应用程序。通常，它将添加到带有注释的主应用程序类中，@SpringBootApplication但可以将其添加到任何@Configuration类中。默认情况下，将从声明注释的类的包中进行扫描。如果要定义要扫描的特定程序包

```
@SpringBootApplication
@ConfigurationPropertiesScan({ "com.example.app", "org.acme.another" })
public class MyApplication {
}
```

- 宽松的绑定

- acme.my-project.person.first-name

use in .properties and .yaml 中划线格式

- acme.myProject.person.firstName

驼峰格式

- acme.my\_project.person.first\_name

下划线格式

- ACME\_MYPROJECT\_PERSON\_FIRSTNAME 系统环境变量 大写格式

- 以上配置可以在properties,.yaml,环境变量,系统变量中使用

- 转换时间

(SpringBoot2+)

- Spring Boot为表达持续时间提供了专门的支持。

可以使用该注解进行类型转换

```
@DurationUnit(ChronoUnit.SECONDS)
```

配置文件值可以是:

- Long形式，使用ms作为默认单位

- ISO-8601格式

- 值和单位耦合的可读性高的单位(如 10s表示10秒)

- @ConfigurationProperties("app.system")

```
public class AppSystemProperties {
    @DurationUnit(ChronoUnit.SECONDS)
    private Duration sessionTimeout = Duration.ofSeconds(30);
}
```

- 转换日期

(SpringBoot2+)

- Spring Boot还可以使用java.time.Periodtype。

@PeriodUnit(ChronoUnit.SECONDS)

可以在应用程序属性中使用以下格式：

- 常规int表示形式（除非@PeriodUnit指定，否则使用天作为默认单位）

- 标准的ISO-8601格式使用java.time.Period

- 值和单位对耦合的更简单格式（例如，1y3d表示1年零3天）

org.springframework.boot.convert.PeriodToStringConverter

- 转换数据大小

(SpringBoot2+)

- Spring Framework的DataSize值类型表示字节大小

@DataSizeUnit(DataUnit.MEGABYTES)

应用程序属性中的以下格式可用：

- 常规long表示形式（除非@DataSizeUnit已指定a，否则使用字节作为默认单位）

2.值和单位耦合的更具可读性的格式（例如，10MB意味着10兆字节）

- @ConstructorBinding 构造函数绑定

@ConfigurationProperties("acme")

- @PropertySource("")

与@Configuration，

或 @ConfigurationProperties

一起使用指定配置文件

- @Value (“\${?:default}”) 注入properties里面的配置项，及执行部分SpEL表达式

- SpEL 支持算数运算符，逻辑运算符，三目运算符(a?:b),

a?.b等

<https://www.jianshu.com/p/e0b50053b5d3>

- 通过@value("#{}")获取springcontext容器中的值的信息。

如果我们想通过@value获取spring容器中的值（包括bean和bean的属性值），我们可以通过@value("#{bean名称}")或者@value("#{bean名称.属性名}",该属性要有setter方法)

可执行SpEL表达式

- 通过@value("\${}")获取properties文件中的值的信息。

可执行部分SpEL表达式

- \${}和#{}可嵌套使用:\${cacheTimeOut:#{60603\*1000}}

- @ImportResource 加载xml配置文件

- @Import 导入配置文件配置类

- @EnableDiscoveryClient 开启服务发现，可用于zookeeper和eureka

org.springframework.cloud.client.discovery.EnableDiscoveryClient

- zookeeper注册中心依赖

org.springframework.cloud  
spring-cloud-starter-zookeeper-discovery

- Feign(Ribbon+Hystrix)
- @EnableFeignClients 启用feign

org.springframework.cloud.netflix.feign.EnableFeignClients

- feign/hystrix依赖

org.springframework.cloud  
spring-cloud-starter-feign

org.springframework.cloud  
spring-cloud-starter-hystrix

- @EnableHystrix 启用断路器
- 同feign.hystrix.enabled=true
- hystrix.command.default.execution.timeout.enabled=true 设置Hystrix默认超时开启

- #配置部分接口,需要指定类名#方法名

hystrix.command.BaseStudyServiceClient#savePaperTaskSubmitInfo.execution.isolation.thread.timeoutInMilliseconds

(com.netflix.hystrix.HystrixCommandProperties#getProperty)

- hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds

设置Hystrix超时时间

- @EnableCircuitBreaker 开启断路器,可服务熔断
- @FeignClient(value="serverName",fallback=XXX.class) 设置接口为Feign请求
- @HystrixCommand注解方法失败后, 系统将切换到fallbackMethod方法执行。

指定回调方法

使用feign时, 无需设置改注解

- Eureka
- @EnableEurekaClient配置本应用将使用服务注册和服务发现
- @EnableEurekaServer 启动一个服务注册中心
- SpringMVC
- @RestControllerAdvice

@ControllerAdvice

- 增强Controller处理, 可实现:

#### ■ Controller 错误全局处理

#### ■ Controller全局数据绑定

#### 3.Controller 全局数据预处理

- @ExceptionHandler
- 设置异常处理方法, 作用于方法, 与@ControllerAdvice配合使用
- @InitBinder
- 数据预处理, 作用于方法, 与



@ControllerAdvice配合使用

- @ModelAttribute
- 数据绑定
- @RestController 声明Controller为RESTFul接口，并且各方法的返回值会自动处理为@ResponseBody

- @CrossOrigin 允许跨域
- @Controller 声明一个Controller类
- @RequestMapping

@PostMapping

@GetMapping

- @PathVariable 注入路径参数

@RequestParam 注入QueryString/Form参数

@RequestBody 注入RequestBody为对象

- @ResponseBody 设置返回值对象转为json
- SpringBoot设置时区
- 1. 接口JSON设置时区

spring.jackson.date-format=yyyy-MM-dd HH:mm:ss

spring.jackson.time-zone=Asia/Shanghai

- 2. 项目环境设置时区(可尝试在SpringApplication.run()前设置)

@PostConstruct

```
void setDefaultTimezone() {
```

```
    TimeZone.setDefault(TimeZone.getTimeZone("Asia/Shanghai"));
```

```
}
```

- JPA

- @Entity

@Table

- @Column 声明字段
- @Transient 声明不需要与数据库映射的字段，在保存的时候不需要保存进数据库

。

- 测试

- @SpringBootTest(webEnvironment = RANDOM\_PORT)

@ActiveProfiles("test")

@Slf4j

```
public abstract class TestBase {
```

```
    .....
```

```
}
```

- @Test声明一个方法为测试方法

- @Test

@Transactional

```
@WithMockUser(username = "user-id-18163138155", authorities =
```

```
"ROLE_TEACHER")
```

```
void should_import_student_success() throws Exception {
```

```
    .....
```

```
}
```

- @Transactional被声明的测试方法的数据会回滚，避免污染测试数据。
- @WithMockUser Spring Security 提供的，用来模拟一个真实用户，并且可以赋

予权限。

- Mybatis
  - @MapperScan Mybatis接口扫描
  - org.mybatis.spring.boot  
mybatis-spring-boot-starter  
\${mybatis.spring.boot.version}
- @Transactional 开启事务，非Mybatis
- Scheduling
  - @EnableScheduling 开启Scheduling
  - @Scheduled(cron = "\${}") 开启定时任务
- Mongo
  - @EnableMongoRepositories(basePackages = {""}) 开启Mongo Repository扫描
- Swagger 自动生成接口文档
  - @Api(value="",description="",tags="") 声明一个API组
  - @ApiOperation(value="",notes="",httpMethod="") 声明一个API
- Lombok 自动生成Bean Get/Set
  - @Data
  - @Setter
  - @Getter
  - @AllArgsConstructor
  - @NoArgsConstructor
  - @Slf4j 自动生成Slf4j日志对象

- Gateway(Zuul) 网关，使用各种过滤器实现
  - com.netflix.zuul.ZuulFilter  
public class PostFilter extends ZuulFilter{}
- Discovery
  - zookeeper
  - eureka
- Feign(Hystix+Ribbon)
- Spring-Retry(重试机制)

## RPC(RemoteProcedureCall)

- Dubbo  
<https://dubbo.apache.org/zh/docs/v2.7/user/preface/architecture/>  
支持编码API方式和Spring XML/注解2种调用方式

- 概念:  
Apache Dubbo 是一款高性能、轻量级的开源 Java 服务框架;  
Dubbo是一个分布式服务框架，致力于提供高性能和透明化的RPC远程服务调用方案，以及SOA服务治理方案。提供了六大核心能力：面向接口代理的高性能RPC调用，智能容错和负载均衡，服务自动注册和发现，高度可扩展能力，运行期流量调度，可视化的服务治理与运维;  
Dubbo由阿里开发,后加入Apache;

- Dubbo底层原理和实现机制简述:

Dubbo是一个RPC,SOA框架,默认协议(dubbo)采用单一长连接和NIO异步通讯;作为RPC,支持dubbo,hessian,injvm等传输协议,底层默认采用Netty长连接传输,是典型的生产者(Provider)/消费者(Consumer)模型;

作为SOA,具有服务治理功能,提供服务注册和发现,并提供容错和负载均衡机制,默认使用Zookeeper作为注册中心,Provider启动时,会将所有接口按Service全局限定名注册到zookeeper/dubbo/节点下,并订阅/dubbo/configurators;Consumer启动时订阅/dubbo/providers,./dubbo/routers,/dubbo/configurators,当服务发生变化时,将变更推送到Consumer端;

Provider和Consumer启动后会在内存中记录接口调用次数等信息,并定时每分钟发送到Monitor一次

- Dubbo初始化过程

- 启动Dubbo,即将服务装载容器中, 然后注册服务,和Spring启动过程类似;

#读取/解析配置文件

解析服务:

1) 基于dubbo.jar内的META-INF/spring.handlers配置, spring在遇到dubbo名称空间时, 会回调DubboNamespaceHandler类。

2) 所有的dubbo标签, 都统一用DubboBeanDefinitionParser进行解析, 基于一对一属性映射, 将XML标签解析为Bean对象。

在ServiceConfig.export 或者ReferenceConfig.get 初始化时, 将Bean对象转会为url格式, 将所以Bean属性转成url的参数。

然后将URL传给Protocol扩展点, 基于扩展点的Adaptive机制, 根据URL的协议头, 进行不同协议的服务暴露和引用。

a、只暴露服务端口

在没有使用注册中心的情况, 这种情况一般适用在开发环境下, 服务的调用这和提供在同一个IP上, 只需要打开服务的端口即可。

即, 当配置

ServiceConfig解析出的URL的格式为:

dubbo://service-host/com.xxx.TxxService?version=1.0.0

基于扩展点的Adaptive机制, 通过URL的“dubbo://”协议头识别, 直接调用DubboProtocol的export()方法, 打开服务端口。

b、向注册中心暴露服务:

和上一种的区别: 需要将服务的IP和端口一同暴露给注册中心。

ServiceConfig解析出的url格式为:

registry://registry-

host/com.alibaba.dubbo.registry.RegistryService?

export=URL.encode(“dubbo://service-host/com.xxx.TxxService?version=1.0.0”)

基于扩展点的Adaptive机制, 通过URL的“registry://”协议头识别, 调用RegistryProtocol的export方法, 将export参数中的提供者URL先注册到注册中心, 再重新传给Protocol扩展点进行暴露:

dubbo://service-host/com.xxx.TxxService?version=1.0.0

- Dubbo心跳机制  
(维持provider和consumer之间的长连接)  
org.apache.dubbo.remoting.exchange.support.header.HeartbeatHandler  
中实现

- dubbo心跳时间heartbeat默认是1s, 超过heartbeat时间没有收到消息, 就发送心跳消息(provider, consumer一样),如果连着3次(heartbeatTimeout为heartbeat\*3)没有收到心跳响应, provider会关闭channel, 而consumer会进行重连;不论是provider还是consumer的心跳检测都是通过启动定时任务的方式实现;

- 架构

- 架构图:<https://dubbo.apache.org/imgs/user/dubbo-architecture.jpg>  
架构简要说明:

- 注册中心Registry提供服务发现和服务注册能力,服务提供方Provider在启动初始化时注册(Register)服务到Registry,服务消费方Consumer在启动初始化时,订阅(Subscribe)需要使用的服务
- 服务提供者发生变化时,注册中心Registry异步通知服务消费方Consumer,更新内部服务者缓存信息
- 服务调用时,由服务消费方Consumer基于负载均衡算法从服务方列表中选择一个服务,并『同步』调用该服务接口,如果失败则使用另一台服务调用(默认为failover,重试2次,每次都会重新获取服务提供者列表,并重新选举)
- 消费提供者Provider和服务消费者Consumer,在内存中缓存各自服务调用次数和调用时间, 定时每分钟『异步』发送一次数据到监控中心Monitor

- 角色

- Provider 暴露服务的服务提供方,用于暴露服务和注册服务到注册中心
- Consumer 调用远程服务的的服务消费方
- Registry 服务注册与发现的注册中心
- Monitor 统计服务的调用次数和调用时间的监控中心
- Container 服务运行容器,一般为Spring容器
- 调用关系说明:

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

- Consumer调用Provider接口调用流程:

- @Reference注解对象调用,  
Dubbo会为@Service,@Reference类重新生成代理类,并重写  
InvocationHandler(org.apache.dubbo.rpc.proxy.InvokerInvocationHandler),若为自  
定义的Service方法,则再创建org.apache.dubbo.rpc.RpcInvocation对象,然后执行  
invoke(rpcInvocation)方法
- 路由和请求调用:  
invoke(rpcInvocation)过程中会进入  
org.apache.dubbo.rpc.cluster.support.wrapper.AbstractCluster\$InterceptorInvoker  
Node.invoke方法处理, 并先进入ConsumerContextClusterInterceptor拦截器处理,然  
后执行【org.apache.dubbo.rpc.cluster.support.AbstractClusterInvoker#invoke()】  
方法,经过RegistryDirectory.doList()方法的routerChain处理,返回invokers列表(即服务  
Provider列表),然后根据invokers创建LoadBalance负载均衡器(此处配置了默认负载均  
衡random),Dubbo默认在启动时加载了org/apache/dubbo/dubbo/2.7.6/dubbo-  
2.7.6.jar!/META-INF/dubbo/internal/org.apache.dubbo.rpc.cluster.LoadBalance中的  
4种负载均衡策略,然后继续执行doInvoke方法(实际执行的是AbstractClusterInvoker子  
类的doInvoke,默认为FailoverClusterInvoker),  
#实际发起连接请求  
org.apache.dubbo.rpc.protocol.AsyncToSyncInvoker#invoke方法中  
org.apache.dubbo.rpc.protocol.AbstractInvoker#invoke()=>  
org.apache.dubbo.rpc.protocol.dubbo.DubboInvoker.doInvoke();  
consumer发起请求时DefaultFuture传入的executor用于超时处理,实际默认使用  
NettyClient发起请求处理
- FailoverClusterInvoker中处理了重试次数,及默认的重试次数;并进行  
LoadBalance.doSelect()选择目标Provider,并进行请求,重试处理;其中会经过  
org.apache.dubbo.rpc.filter.ConsumerContextFilter处理;  
最后,通过org.apache.dubbo.rpc.AsyncRpcResult处理返回结果
- Provider在netty收到请求时,为请求的URL创建FixedThreadPool,见  
org.apache.dubbo.common.threadpool.manager  
.DefaultExecutorRepository#createExecutor
  - 依赖:  
<https://dubbo.apache.org/zh/docs/v2.7/user/dependencies/>
  - 用法  
<https://dubbo.apache.org/zh/docs/v2.7/user/quick-start/>  
<https://dubbo.apache.org/zh/docs/v2.7/user/examples/>  
<https://github.com/apache/dubbo/blob/master/dubbo-demo/pom.xml>
  - API 的方式(编码):  
<https://dubbo.apache.org/zh/docs/v2.7/user/configuration/api/>
  - XML(Spring)
    - Local Spring配置(local.xml),即服务提供方和调用方在一个项目中,  
为普通的Bean调用:

- 2.Remote Spring配置,即服务提供方和调用方在不同的项目中,为RPC调用:

在本地服务的基础上, 只需做简单配置, 即可完成远程化:

将上面的 local.xml 配置拆分成两份,

a. 将服务定义部分放在服务提供方 remote-provider.xml, 将服务引用部分放在服务消费方 remote-consumer.xml。

b. 并在提供方增加暴露服务配置 [dubbo:service](#), 在消费方增加引用服务配置 [dubbo:reference](#)。

remote-provider.xml:

```
<dubbo:service interface="com.xxx.XxxService" ref="xxxService" />
```

remote-consumer.xml:

```
<dubbo:reference id="xxxService" interface="com.xxx.XxxService" />
```

- MAVEN依赖:(SpringFramework,Zookeeper(+curator),Dubbo)

```
org.springframework  
spring-framework-bom  
5.2.7.RELEASE  
pom  
import
```

```
org.apache.dubbo  
dubbo  
2.7.6
```

```
org.apache.dubbo  
dubbo-dependencies-zookeeper  
2.7.6  
pom
```

org.springframework  
spring-beans

org.springframework  
spring-core

org.springframework  
spring-context

org.springframework  
spring-context-support

org.springframework  
spring-aop

org.apache.dubbo  
dubbo

org.apache.curator  
curator-recipes

org.apache.zookeeper  
zookeeper

■ 启动:

```
public class DubboServerProviderApplication {  
    public static void main(String[] args) throws IOException {  
        ClassPathXmlApplicationContext context = new  
        ClassPathXmlApplicationContext(new String[]{"dubbo-server-  
        provider.xml"});  
        context.start();  
        System.in.read();  
    }  
}
```

xml配置:

dubbo-client-consumer.xml:

```

<dubbo:reference id="xmlDemoService"
interface="org.kangspace.javamindmap.dubbodemo.service.xml.Xml
IDemoService" />
dubbo-server-provider.xml:

```

```

<dubbo:application name="dubbo-service-app" />

```

```

<dubbo:registry address="zookeeper://127.0.0.1:2181" />

```

```

<dubbo:protocol name="dubbo" port="20880" />

```

```

<dubbo:service
interface="org.kangspace.javamindmap.dubbodemo.service.xml.Xml
IDemoService"
    version="1.0.0" group="demo" timeout="5000"
    ref="xmlDemoService" />

```

- 注解(SpringBoot/Spring AnnotationConfigApplicationContext)

- Maven依赖(同XML方式,依赖  
SpringFramework,Zookeeper(+curator),Dubbo)

- 使用方式

- 1.Provider

```

public static void main(String[] args) throws IOException {
    AnnotationConfigApplicationContext context = new
    AnnotationConfigApplicationContext(ProviderConfiguration.class);
    context.start();
    String runningTip = "Dubbo Server is running...";
    log.info(runningTip);
    System.out.println(runningTip);
    System.in.read(); // 按任意键退出
}
/**

```

- 配置类

```

*/
@Configuration
@EnableDubbo(scanBasePackages =
"org.kangspace.javamindmap.dubbodemo.service.impl.annotation")
@EnableDubboConfig
@PropertySource("classpath:/dubbo-server-provider.properties")
public static class ProviderConfiguration{
}

```

- 2.Consumer



```

public static void main(String[] args) {
    AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(ConsumerConfiguration.class);
    context.start();
    AnnotationDemoServiceCallService annotationDemoServiceCallService =
context.getBean(AnnotationDemoServiceCallService.class);
    String hello = annotationDemoServiceCallService.sayHello();
    String runningTip = "Dubbo Client is running...";
    log.info(runningTip);
    System.out.println(runningTip);
    log.info("DubboClientAnnotationProviderApplication run : "+hello);
}
/**

```

#### ■ 配置类

```

*/
@Configuration
@EnableDubbo(scanBasePackages =
"org.kangspace.javamindmap.dubbodemo.client.annotation")
@EnableDubboConfig
@PropertySource("classpath:/dubbo-client-consumer.properties")
@ComponentScan(basePackages =
"org.kangspace.javamindmap.dubbodemo.client.annotation")
public static class ConsumerConfiguration{
}
    - dubbo.properties:
dubbo.application.name=dubbo-server-annotation-provider
dubbo.protocol.name=dubbo
dubbo.protocol.port=20881
dubbo.registry.address=zookeeper://127.0.0.1:2181
    - 类级别的注解:
@org.apache.dubbo.config.annotation.Service 定义Dubbo服务
字段级别注解:
@org.apache.dubbo.config.annotation.Reference 注入引用Dubbo服务

```

#### ■ 用法示例说明

##### ■ 启动时检查:

check=true,默认true;

Dubbo 缺省会在启动时检查依赖的服务是否可用，不可用时会抛出异常，阻止 Spring 初始化完成，以便上线时，能及早发现问题，默认 check="true";

可在xml,properties及@Reference(check=false)设置;

<https://dubbo.apache.org/zh/docs/v2.7/user/examples/preflight-check/>

##### ■ 关闭某个服务的启动时检查 (没有提供者时报错):

```

<dubbo:reference interface="com.foo.BarService"
check="false" />

```

dubbo.reference.com.foo.BarService.check=false(设置某个服务

的check值)

dubbo.reference.check=false(强制改变所有 reference 的 check 值, 就算配置中有声明, 也会被覆盖。)

- 关闭所有服务的启动时检查 (没有提供者时报错):

```
<dubbo:consumer check="false" />
```

dubbo.consumer.check=false(设置 check 的缺省值, 如果配置中有显式的声明, 则不影响)

- 关闭注册中心启动时检查 (注册订阅失败时报错):

```
<dubbo:registry check="false" />
```

dubbo.registry.check=false(前面两个都是指订阅成功, 但提供者列表是否为空是否报错, 如果注册订阅失败时, 也允许启动, 需使用此选项, 将在后台定时重试)

- 集群容错模式:

<https://dubbo.apache.org/zh/docs/v2.7/user/examples/fault-tolerant-strategy/>

- Failover Cluster, 失败自动切换;默认模式;当出现失败, 重试其它服务器。通常用于读操作, 但重试会带来更长延迟。可通过 retries="2" 来设置重试次数(不含第一次)。
- Failfast Cluster 快速失败, 只发起一次调用, 失败立即报错。通常用于非幂等性的写操作, 比如新增记录。
- Failsafe Cluster 失败安全, 出现异常时, 直接忽略。通常用于写入审计日志等操作。
- Failback Cluster 失败自动恢复, 后台记录失败请求, 定时重发。通常用于消息通知操作。
- Forking Cluster 并行调用多个服务器, 只要一个成功即返回。通常用于实时性要求较高的读操作, 但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数。

6.Broadcast Cluster 广播调用所有提供者, 逐个调用, 任意一台报错则报错。通常用于通知所有提供者更新缓存或日志等本地资源信息。

集群模式配置:

```
<dubbo:service cluster="failsafe" /> 或
```

```
@Service( cluster="failsafe" )
```

```
<dubbo:reference cluster="failsafe" />或
```

```
@Reference( cluster="failsafe" )
```

- Failover Cluster配置(优先级: method>reference>service):

```
<dubbo:service retries="2" />或
```

```
<dubbo:reference retries="2" />或
```

[dubbo:reference](#)

```
<dubbo:method name="findFoo" retries="2" />
```

[/dubbo:reference](#)

或对应注解模式

- Broadcast Cluster:

broadcast.fail.percent=20 代表了当 20% 的节点调用失败就抛出异常, 不再调用其他节点  
@reference(cluster = "broadcast", parameters = {"broadcast.fail.percent", "20"})

- 负载均衡:

在集群负载均衡时, Dubbo 提供了多种均衡策略, 缺省为 random 随机调用。

负载均衡配置:

服务端:

```
<dubbo:service interface="..." loadbalance="roundrobin" />
```

客户端:

```
<dubbo:reference interface="..." loadbalance="roundrobin" />
```

服务端方法级别:

```
<dubbo:service interface="...">
```

```
<dubbo:method name="..." loadbalance="roundrobin"/>
```

[/dubbo:service](#)

客户端方法级别:

```
<dubbo:reference interface="...">
```

```
<dubbo:method name="..." loadbalance="roundrobin"/>
```

[/dubbo:reference](#)

或对应的注解模式配置

- Random LoadBalance : 随机, 按权重设置随机概率;默认策略
- RoundRobin LoadBalance : 轮询, 按公约后的权重设置轮询比率
- LeastActive LoadBalance : 最少活跃调用数, 相同活跃数的随机, 活跃数指调用前后

计数差

- ConsistentHash LoadBalance : 一致性 Hash, 相同参数的请求总是发到同一提供

者。

当某一提供者挂时, 原本发往该提供者的请求, 基于虚拟节点, 平摊到其它提供者, 不会引起剧烈变动。

缺省只对第一个参数 Hash, 如果要修改, 请配置 `<dubbo:parameter key="hash.arguments" value="0,1" />`

缺省用 160 份虚拟节点, 如果要修改, 请配置 `<dubbo:parameter key="hash.nodes" value="320" />`

算法参见: [http://en.wikipedia.org/wiki/Consistent\\_hashing](http://en.wikipedia.org/wiki/Consistent_hashing)

- Dubbo线程模型:

通过不同的派发策略和不同的线程池配置的组合来应对不同的场景:

```
<dubbo:protocol name="dubbo" dispatcher="all" threadpool="fixed" threads="100" />
```

- Dispatcher:

all 所有消息都派发到线程池, 包括请求, 响应, 连接事件, 断开事件, 心跳等。

direct 所有消息都不派发到线程池, 全部在 IO 线程上直接执行。

message 只有请求响应消息派发到线程池, 其它连接断开事件, 心跳等消息, 直接在 IO 线程上执行。

execution 只有请求消息派发到线程池, 不含响应, 响应和其它连接断开事件, 心跳等消息, 直接在 IO 线程上执行。

connection 在 IO 线程上, 将连接断开事件放入队列, 有序逐个执行, 其它消息派发到线程池。

- ThreadPool:

fixed 固定大小线程池, 启动时建立线程, 不关闭, 一直持有。(缺省)

cached 缓存线程池, 空闲一分钟自动删除, 需要时重建。

limited 可伸缩线程池, 但池中的线程数只会增长不会收缩。只增长不收缩的目的是为了避免收缩时突然来了大流量引起的性能问题。

eager 优先创建Worker线程池。在任务数量大于corePoolSize但是小于maximumPoolSize时, 优先创建Worker来处理任务。当任务数量大于maximumPoolSize时, 将任务放入阻塞队

列中。阻塞队列充满时抛出RejectedExecutionException。(相比于cached:cached在任务数量超过maximumPoolSize时直接抛出异常而不是将任务放入阻塞队列)

- 直连提供者:

Dubbo 中点对点的直连方式

- 通过 XML / 注解 配置:

```
<dubbo:reference id="xxxService" interface="com.alibaba.xxx.XxxService"
url="dubbo://localhost:20890" />
```

或

```
@Reference(url="")
```

- 通过 -D 参数指定:

```
java -Dcom.alibaba.xxx.XxxService=dubbo://localhost:20890
```

key 为服务名, value 为服务提供者 url, 此配置优先级最高, 1.0.15 及以上版本支持

- 通过文件映射:

```
java -Ddubbo.resolve.file=xxx.properties
```

文件内容:

然后在映射文件 xxx.properties 中加入配置, 其中 key 为服务名, value 为服务提供者

URL:

```
com.alibaba.xxx.XxxService=dubbo://localhost:20890
```

- 只订阅不注册:

禁用注册配置:

```
<dubbo:registry address="10.20.153.10:9090" register="false" />
```

或者

```
<dubbo:registry address="10.20.153.10:9090?register=false" />
```

或在配置文件中配置:

```
dubbo.registry.register=false
```

- 多协议:

不同服务在性能上适用不同协议进行传输, 比如大数据用短连接协议, 小数据大并发用长连接协议;

配置:

```
<dubbo:protocol name="dubbo" port="20880" />
```

```
<dubbo:protocol name="rmi" port="1099" />
```

```
<dubbo:service interface="com.alibaba.hello.api.HelloService" version="1.0.0"
ref="helloService" protocol="dubbo" />
```

```
<dubbo:service interface="com.alibaba.hello.api.DemoService" version="1.0.0"
ref="demoService" protocol="rmi" />
```

```
<dubbo:protocol name="hessian" port="8080" />
```

```
<dubbo:service id="helloService" interface="com.alibaba.hello.api.HelloService"
version="1.0.0" protocol="dubbo,rmi" />
```

- 多注册中心:

Dubbo 支持同一服务向多注册中心同时注册, 或者不同服务分别注册到不同的注册中心上去, 甚至可以同时引用注册在不同注册中心上的同名服务;

- 多注册中心注册:

比如: 中文站有些服务来不及在青岛部署, 只在杭州部署, 而青岛的其它应用需要引用此服

务，就可以将服务同时注册到两个注册中心。

```
<dubbo:application name="world" />

<dubbo:registry id="hangzhouRegistry" address="10.20.141.150:9090" />
<dubbo:registry id="qingdaoRegistry" address="10.20.141.151:9010" default="false"
/>

<dubbo:service interface="com.alibaba.hello.api.HelloService" version="1.0.0"
ref="helloService" registry="hangzhouRegistry,qingdaoRegistry" />
```

- 不同服务使用不同注册中心：  
比如：CRM 有些服务是专门为国际站设计的，有些服务是专门为中文站设计的。

```
<dubbo:application name="world" />

<dubbo:registry id="chinaRegistry" address="10.20.141.150:9090" />
<dubbo:registry id="intlRegistry" address="10.20.154.177:9010" default="false" />

<dubbo:service interface="com.alibaba.hello.api.HelloService" version="1.0.0"
ref="helloService" registry="chinaRegistry" />

<dubbo:service interface="com.alibaba.hello.api.DemoService" version="1.0.0"
ref="demoService" registry="intlRegistry" />
```

- 多注册中心引用  
比如：CRM 需同时调用中文站和国际站的 PC2 服务，PC2 在中文站和国际站均有部署，接口及版本号都一样，但连的数据库不一样。

```
<dubbo:application name="world" />

<dubbo:registry id="chinaRegistry" address="10.20.141.150:9090" />
<dubbo:registry id="intlRegistry" address="10.20.154.177:9010" default="false" />

<dubbo:reference id="chinaHelloService"
interface="com.alibaba.hello.api.HelloService" version="1.0.0" registry="chinaRegistry"
/>

<dubbo:reference id="intlHelloService" interface="com.alibaba.hello.api.HelloService"
version="1.0.0" registry="intlRegistry" />

<dubbo:registry address="10.20.141.150:9090|10.20.154.177:9010" />
```

#### - 服务分组:

使用服务分组区分服务接口的不同实现

当一个接口有多种实现时, 可以用 group 区分。

服务

```
<dubbo:service group="feedback" interface="com.xxx.IndexService" />
```

```
<dubbo:service group="member" interface="com.xxx.IndexService" />
```

引用

```
<dubbo:reference id="feedbackIndexService" group="feedback"
interface="com.xxx.IndexService" />
```

```
<dubbo:reference id="memberIndexService" group="member"
interface="com.xxx.IndexService" />
```

任意组:

```
<dubbo:reference id="barService" interface="com.foo.BarService" group="" />
```

#### - 静态服务:

将 Dubbo 服务标识为非动态管理模式,需手工处理服务上下线

```
<dubbo:registry address="10.20.141.150:9090" dynamic="false" />
```

或

```
<dubbo:registry address="10.20.141.150:9090?dynamic=false" />
```

#### - 分组聚合:

合并所有分组

```
<dubbo:reference interface="com.xxx.MenuService" group="" merger="true" />
```

合并指定分组:

```
<dubbo:reference interface="com.xxx.MenuService" group="aaa,bbb" merger="true"
/>
```

指定方法合并结果, 其它未指定的方法, 将只调用一个 Group:

```
<dubbo:reference interface="com.xxx.MenuService" group="">
```

```
<dubbo:method name="getMenuItems" merger="true" />
```

[/dubbo:reference](#)

#### - 参数验证:

参数验证功能是基于 JSR303 实现的, 用户只需标识 JSR303 标准的验证 annotation, 并通过声明 filter 来实现验证, 即 `javax.validation.validation-api` 和 `org.hibernate.hibernate-validator`

#### - 结果缓存:

配置:

```
<dubbo:reference interface="com.foo.BarService" cache="lru" />
```

或

```
<dubbo:reference interface="com.foo.BarService">
```

```
<dubbo:method name="findBar" cache="lru" />
```

[/dubbo:reference](#)

或对应的注解配置

#### - 缓存类型:

`lru` 基于最近最少使用原则删除多余缓存, 保持最热的数据被缓存。

`threadlocal` 当前线程缓存, 比如一个页面渲染, 用到很多 portal, 每个 portal 都要去查用户信息, 通过线程缓存, 可以减少这种多余访问。

`jcache` 与 JSR107 集成, 可以桥接各种缓存实现。

#### - 泛化调用:

<https://dubbo.apache.org/zh/docs/v2.7/user/examples/generic-reference/>

- 上下文信息:

<https://dubbo.apache.org/zh/docs/v2.7/user/examples/context/>

上下文中存放的是当前调用过程中所需的环境信息。所有配置信息都将转换为 URL 的参数，参见 schema 配置参考手册 中的对应URL参数一列。

RpcContext 是一个 ThreadLocal 的临时状态记录器，当接收到 RPC 请求，或发起 RPC 请求时，RpcContext 的状态都会变化。比如：A 调 B，B 再调 C，则 B 机器上，在 B 调 C 之前，RpcContext 记录的是 A 调 B 的信息，在 B 调 C 之后，RpcContext 记录的是 B 调 C 的信息。

- 隐式参数:

通过 Dubbo 中的 Attachment 在服务消费方和提供方之间隐式传递参数

可以通过 RpcContext 上的 setAttachment 和 getAttachment 在服务消费方和提供方之间进行参数的隐式传递。

- 在服务消费方端设置隐式参数

setAttachment 设置的 KV 对，在完成下面一次远程调用会被清空，即多次远程调用要多次设置。

RpcContext.getContext().setAttachment("index", "1"); // 隐式传参，后面的远程调用都会隐式将这些参数发送到服务器端，类似cookie，用于框架集成，不建议常规业务使用

xxxService.xxx(); // 远程调用

// ...

- 异步执行:

Dubbo 服务提供方的异步执行

- 1.定义 CompletableFuture 签名的接口

```
public interface AsyncService {
```

```
    CompletableFuture sayHello(String name);
```

```
}
```

```
public class AsyncServiceImpl implements AsyncService {
```

```
    @Override
```

```
    public CompletableFuture sayHello(String name) {
```

```
        RpcContext savedContext = RpcContext.getContext();
```

```
        // 建议为supplyAsync提供自定义线程池，避免使用JDK公用线程池
```

```
        return CompletableFuture.supplyAsync(() -> {
```

```
            System.out.println(savedContext.getAttachment("consumer-key1"));
```

```
            try {
```

```
                Thread.sleep(5000);
```

```
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
```

```
            }
```

```
            return "async response from provider.";
```

```
        });
```

```
    }
```

```
}
```

通过 return CompletableFuture.supplyAsync()，业务执行已从 Dubbo 线程切换到业务线程，避免了对 Dubbo 线程池的阻塞。

- 使用AsyncContext:

Dubbo 提供了一个类似 Servlet 3.0 的异步接口AsyncContext，在没有 CompletableFuture 签名接口的情况下，也可以实现 Provider 端的异步执行。

服务接口定义：

```
public interface AsyncService {
    String sayHello(String name);
}
```

服务暴露，和普通服务完全一致:

服务实现:

```
public class AsyncServiceImpl implements AsyncService {
    public String sayHello(String name) {
        final AsyncContext asyncContext = RpcContext.startAsync();
        new Thread() -> {
            // 如果要使用上下文，则必须要放在第一句执行
            asyncContext.signalContextSwitch();
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // 写回响应
            asyncContext.write("Hello " + name + ", response from provider.");
        }.start();
        return null;
    }
}
```

- 异步调用:

在 Dubbo 中发起异步调用,使用 CompletableFuture 签名的接口;

需要服务提供者事先定义 CompletableFuture 签名的服务

- 1. xml引用

```
<dubbo:reference id="asyncService" timeout="10000"
interface="com.alibaba.dubbo.samples.async.api.AsyncService"/>
```

调用:

```
// 调用直接返回CompletableFuture
CompletableFuture future = asyncService.sayHello("async call request");
// 增加回调
future.whenComplete((v, t) -> {
    if (t != null) {
        t.printStackTrace();
    } else {
        System.out.println("Response: " + v);
    }
});
// 早于结果输出
System.out.println("Executed before response return.");
```

- 2. 使用 RpcContext

在 consumer.xml 中配置:

```
<dubbo:reference id="asyncService"
interface="org.apache.dubbo.samples.governance.api.AsyncService">
    <dubbo:method name="sayHello" async="true" />
```



## [/dubbo:reference](#)

调用代码:

```
// 此调用会立即返回null
asyncService.sayHello("world");
// 拿到调用的Future引用, 当结果返回后, 会被通知和设置到此Future
CompletableFuture helloFuture = RpcContext.getContext().getCompletableFuture();
// 为Future添加回调
helloFuture.whenComplete((retValue, exception) -> {
    if (exception == null) {
        System.out.println(retValue);
    } else {
        exception.printStackTrace();
    }
});或
CompletableFuture future = RpcContext.getContext().asyncCall(
    () -> {
        asyncService.sayHello("oneway call request1");
    }
);
future.get();
```

- 本地调用:

本地调用使用了 injvm 协议, 是一个伪协议, 它不开启端口, 不发起远程调用, 只在 JVM 内直接关联, 但执行 Dubbo 的 Filter 链。

- 配置:

定义 injvm 协议

```
<dubbo:protocol name="injvm" />
```

设置默认协议

```
<dubbo:provider protocol="injvm" />
```

设置服务协议

```
<dubbo:service protocol="injvm" />
```

优先使用 injvm

```
<dubbo:consumer injvm="true" .../>
```

```
<dubbo:provider injvm="true" .../>
```

或

```
<dubbo:reference injvm="true" .../>
```

```
<dubbo:service injvm="true" .../>
```

- 参数回调:

通过参数回调从服务器端调用客户端逻辑:

参数回调方式与调用本地 callback 或 listener 相同, 只需要在 Spring 的配置文件中声明哪个参数是 callback 类型即可。Dubbo 将基于长连接生成反向代理, 这样就可以从服务器端调用客户端逻辑

- 事件通知:

在调用之前、调用之后、出现异常时的事件通知

在调用之前、调用之后、出现异常时, 会触发 oninvoke、onreturn、onthrow 三个事件, 可以配置当事件发生时, 通知哪个类的哪个方法。

- 本地伪装:

在 Dubbo 中利用本地伪装实现服务降级

- 并发控制:

Dubbo 中的并发控制

- 配置样例

样例 1

限制 com.foo.BarService 的每个方法，服务器端并发执行（或占用线程池线程数）不能超过 10 个:

```
<dubbo:service interface="com.foo.BarService" executes="10" />
```

样例 2

限制 com.foo.BarService 的 sayHello 方法，服务器端并发执行（或占用线程池线程数）不能超过 10 个:

```
<dubbo:service interface="com.foo.BarService">  
<dubbo:method name="sayHello" executes="10" />
```

[/dubbo:service](#)

样例 3

限制 com.foo.BarService 的每个方法，每客户端并发执行（或占用连接的请求数）不能超过 10 个:

```
<dubbo:service interface="com.foo.BarService" actives="10" />
```

或

```
<dubbo:reference interface="com.foo.BarService" actives="10" />
```

样例 4

限制 com.foo.BarService 的 sayHello 方法，每客户端并发执行（或占用连接的请求数）不能超过 10 个:

```
<dubbo:service interface="com.foo.BarService">  
<dubbo:method name="sayHello" actives="10" />
```

[/dubbo:service](#)

或

```
<dubbo:reference interface="com.foo.BarService">  
<dubbo:method name="sayHello" actives="10" />
```

[/dubbo:service](#)

如果 [dubbo:service](#) 和 [dubbo:reference](#) 都配了 actives, [dubbo:reference](#) 优先, 参见: 配置的覆盖策略。

- Load Balance 均衡

配置服务的客户端的 loadbalance 属性为 leastactive, 此 Loadbalance 会调用并发数最小的 Provider (Consumer端并发数)。

```
<dubbo:reference interface="com.foo.BarService" loadbalance="leastactive" />
```

或

```
<dubbo:service interface="com.foo.BarService" loadbalance="leastactive" />
```

- 连接控制

Dubbo 中服务端和客户端的连接控制

服务端连接控制

限制服务器端接受的连接不能超过 10 个 1:

```
<dubbo:provider protocol="dubbo" accepts="10" />
```

或

```
<dubbo:protocol name="dubbo" accepts="10" />
```

客户端连接控制

限制客户端服务使用连接不能超过 10 个 2:

```
<dubbo:reference interface="com.foo.BarService" connections="10" />
```

或

```
<dubbo:service interface="com.foo.BarService" connections="10" />
```

如果 [dubbo:service](#) 和 [dubbo:reference](#) 都配了 connections, [dubbo:reference](#) 优先, 参见: 配置的覆盖策略

- 延迟连接

在 Dubbo 中配置延迟连接

延迟连接用于减少长连接数。当有调用发起时, 再创建长连接。

```
<dubbo:protocol name="dubbo" lazy="true" />
```

提示

该配置只对使用长连接的 dubbo 协议生效。

- 粘滞连接(默认false)

为有状态服务配置粘滞连接

粘滞连接用于有状态服务, 尽可能让客户端总是向同一提供者发起调用, 除非该提供者挂了, 再连另一台。

粘滞连接将自动开启延迟连接, 以减少长连接数。

```
<dubbo:reference id="xxxService" interface="com.xxx.XxxService" sticky="true" />
```

Dubbo 支持方法级别的粘滞连接, 如果你想进行更细粒度的控制, 还可以这样配置。

```
<dubbo:reference id="xxxService" interface="com.xxx.XxxService">
```

```
<dubbo:method name="sayHello" sticky="true" />
```

[/dubbo:reference](#)

- TLS

通过 TLS 保证传输安全

- 主机绑定

- 查找顺序

缺省主机 IP 查找顺序:

通过 `LocalHost.getLocalHost()` 获取本机地址。

如果是 127. 等 loopback 地址, 则扫描各网卡, 获取网卡 IP。

- 主机配置

注册的地址如果获取不正确, 比如需要注册公网地址, 可以:

可以在 `/etc/hosts` 中加入: 机器名 公网 IP, 比如:

```
test1 205.182.23.201
```

在 `dubbo.xml` 中加入主机地址的配置:

```
<dubbo:protocol host="205.182.23.201">
```

或在 `dubbo.properties` 中加入主机地址的配置:

```
dubbo.protocol.host=205.182.23.201
```

- 端口配置

缺省主机端口与协议相关:

协议 端口

dubbo 20880

rmi 1099

http 80

hessian 80

webservice 80

memcached 11211

redis 6379

可以按照下面的方式配置端口：

在 dubbo.xml 中加入主机地址的配置：

```
<dubbo:protocol name="dubbo" port="20880">
```

或在 dubbo.properties 中加入主机地址的配置：

```
dubbo.protocol.dubbo.port=20880
```

- 主机配置

- 背景

在 Dubbo 中，Provider 启动时主要做两件事情，一是启动 server，二是向注册中心注册服务。启动 server 时需要绑定 socket，向注册中心注册服务时也需要发送 socket 唯一标识服务地址；

查看代码发现，在 org.apache.dubbo.config.ServiceConfig#findConfigedHosts() 中，通过 InetAddress.getLocalHost().getHostAddress() 获取默认 host。其返回值如下：

未联网时，返回 127.0.0.1

在阿里云服务器中，返回私有地址，如：172.18.46.234

在本机测试时，返回公有地址，如：30.5.10.11

- 日志适配

- 日志适配

在 Dubbo 中适配日志框架

自 2.2.1 开始，dubbo 开始内置 log4j、slf4j、jcl、jdk 这些日志框架的适配[1]，也可以通过以下方式显式配置日志输出策略：

命令行

```
java -Ddubbo.application.logger=log4j
```

在 dubbo.properties 中指定

```
dubbo.application.logger=log4j
```

在 dubbo.xml 中配置

```
<dubbo:application logger="log4j" />
```

- 访问日志

配置 Dubbo 的访问日志

如果你想记录每一次请求信息，可开启访问日志，类似于 apache 的访问日志。注意：此日志量比较大，请注意磁盘容量。

将访问日志输出到当前应用的 log4j 日志：

```
<dubbo:protocol accesslog="true" />
```

将访问日志输出到指定文件：

```
<dubbo:protocol accesslog="http://10.20.160.198/wiki/display/dubbo/foo/bar.log" />
```

- 服务容器

- 使用 Dubbo 中的服务容器

服务容器是一个 standalone 的启动程序，因为后台服务不需要 Tomcat 或 JBoss 等 Web 容器的功能，如果硬要用 Web 容器去加载服务提供方，增加复杂性，也浪费资源。

服务容器只是一个简单的 Main 方法，并加载一个简单的 Spring 容器，用于暴露服务。

服务容器的加载内容可以扩展，内置了 spring、jetty、log4j 等加载，可通过容器扩展点进行扩展。配置在 java 命令的 -D 参数或者 dubbo.properties 中。

容器类型

【Spring Container】

自动加载 META-INF/spring 目录下的所有 Spring 配置。

配置 spring 配置加载位置：

dubbo.spring.config=classpath:META-INF/spring.xml

Jetty Container

启动一个内嵌 Jetty，用于汇报状态。

配置：

dubbo.jetty.port=8080：配置 jetty 启动端口

dubbo.jetty.directory=/foo/bar：配置可通过 jetty 直接访问的目录，用于存放静态文件

dubbo.jetty.page=log,status,system：配置显示的页面，缺省加载所有页面

Log4j Container

自动配置 log4j 的配置，在多进程启动时，自动给日志文件按进程分目录。

配置：

dubbo.log4j.file=/foo/bar.log：配置日志文件路径

dubbo.log4j.level=WARN：配置日志级别

dubbo.log4j.subdirectory=20880：配置日志子目录，用于多进程启动，避免冲突

容器启动

缺省只加载 spring

java org.apache.dubbo.container.Main

通过 main 函数参数传入要加载的容器

java org.apache.dubbo.container.Main spring jetty log4j

通过 JVM 启动参数传入要加载的容器

java org.apache.dubbo.container.Main -Ddubbo.container=spring,jetty,log4j

通过 classpath 下的 dubbo.properties 配置传入要加载的容器

dubbo.container=spring,jetty,log4j

- 只注册

只注册不订阅

如果有两个镜像环境，两个注册中心，有一个服务只在其中一个注册中心有部署，另一个注册中心还没来得及部署，而两个注册中心的其它应用都需要依赖此服务。这个时候，可以让服务提供者方只注册服务到另一注册中心，而不从另一注册中心订阅服务。

禁用订阅配置

<dubbo:registry id="hzRegistry" address="10.20.153.10:9090" />

<dubbo:registry id="qdRegistry" address="10.20.141.150:9090" subscribe="false" />

或者

<dubbo:registry id="hzRegistry" address="10.20.153.10:9090" />

<dubbo:registry id="qdRegistry" address="10.20.141.150:9090?subscribe=false" />

- 配置说明

<https://dubbo.apache.org/zh/docs/v2.7/user/configuration>

<https://dubbo.apache.org/zh/docs/v2.7/user/references/xml/dubbo-application/>

xml与properties配置对应关系:

<https://dubbo.apache.org/zh/docs/v2.7/user/recommend/>

- XML

<https://dubbo.apache.org/zh/docs/v2.7/user/configuration/xml/>

- [dubbo:service/](#) 服务配置用于暴露一个服务，定义服务的元信息，一个服务可以用多个协议暴露，一个服务也可以注册到多个注册中心

[dubbo:reference/](#) 2 引用配置用于创建一个远程服务代理，一个引用可以指向多个注册中心

[dubbo:protocol/](#) 协议配置用于配置提供服务的协议信息，协议由提供方指定，消费方被动接受

[dubbo:application/](#) 应用配置 用于配置当前应用信息，不管该应用是提供者还是消费者  
[dubbo:module/](#) 模块配置 用于配置当前模块信息，可选  
[dubbo:registry/](#) 注册中心配置 用于配置连接注册中心相关信息  
[dubbo:monitor/](#) 监控中心配置 用于配置连接监控中心相关信息，可选  
[dubbo:provider/](#) 提供方配置 当 ProtocolConfig 和 ServiceConfig 某属性没有配置时，采用此缺省值，可选  
[dubbo:consumer/](#) 消费方配置 当 ReferenceConfig 某属性没有配置时，采用此缺省值，可选  
[dubbo:method/](#) 方法配置 用于 ServiceConfig 和 ReferenceConfig 指定方法级的配置信息  
[dubbo:argument/](#) 参数配置

- 不同粒度配置的覆盖关系:

以 timeout 为例，下图显示了配置的查找顺序，其它 retries, loadbalance, actives 等类似:

方法级优先，接口级次之，全局配置再次之。

如果级别一样，则消费方优先，提供方次之。

其中，服务提供方配置，通过 URL 经由注册中心传递给消费方。

- 动态配置中心

配置中心 (v2.7.0) 在 Dubbo 中承担两个职责:

- 外部化配置。启动配置的集中式存储（简单理解为 dubbo.properties 的外部化存储）。
- 服务治理。服务治理规则的存储与通知。

- 属性配置:

Dubbo 可以自动加载 classpath 根目录下的 dubbo.properties，但是你同样可以使用 JVM 参数来指定路径: -Ddubbo.properties.file=xxx.properties。

- 映射规则

可以将 xml 的 tag 名和属性名组合起来，用 ‘.’ 分隔。每行一个属性。

dubbo.application.name=foo 相当于 <dubbo:application name="foo" />

dubbo.registry.address=10.20.153.10:9090 相当于 <dubbo:registry address="10.20.153.10:9090" />

如果在 xml 配置中有超过一个的 tag，那么你可以使用 ‘id’ 进行区分。如果你不指定 id，它将作用于所有 tag。

dubbo.protocol.rmi.port=1099 相当于 <dubbo:protocol id="rmi" name="rmi" port="1099" />

dubbo.registry.china.address=10.20.153.10:9090 相当于 <dubbo:registry id="china" address="10.20.153.10:9090" />

如下，是一个典型的 dubbo.properties 配置样例。

dubbo.application.name=foo

dubbo.application.owner=bar

dubbo.registry.address=10.20.153.10:9090

- 重写与优先级:

优先级从高到低:

JVM -D 参数: 当你部署或者启动应用时，它可以轻易地重写配置，比如，改变 dubbo 协议端口;

XML: XML 中的当前配置会重写 dubbo.properties 中的;

Properties: 默认配置，仅仅作用于以上两者没有配置时。

如果在 classpath 下有超过一个 dubbo.properties 文件，比如，两个 jar 包都各自包含

了 dubbo.properties, dubbo 将随机选择一个加载, 并且打印错误日志。  
如果 id 没有在 protocol 中配置, 将使用 name 作为默认属性。

- 注解配置
- 服务提供方

Service注解暴露服务

```
@Service
public class AnnotationServiceImpl implements AnnotationService {
    @Override
    public String sayHello(String name) {
        return "annotation: hello, " + name;
    }
}
```

增加应用共享配置

```
#dubbo-provider.properties
dubbo.application.name=annotation-provider
dubbo.registry.address=zookeeper://127.0.0.1:2181
dubbo.protocol.name=dubbo
dubbo.protocol.port=20880
```

指定Spring扫描路径

```
@Configuration
@EnableDubbo(scanBasePackages =
"org.apache.dubbo.samples.simple.annotation.impl")
@PropertySource("classpath:/spring/dubbo-provider.properties")
static public class ProviderConfiguration {
}
```

服务消费方

Reference注解引用服务

```
@Component("annotationAction")
public class AnnotationAction {
    @Reference
    private AnnotationService annotationService;
    public String doSayHello(String name) {
        return annotationService.sayHello(name);
    }
}
```

增加应用共享配置

```
#dubbo-consumer.properties
dubbo.application.name=annotation-consumer
dubbo.registry.address=zookeeper://127.0.0.1:2181
dubbo.consumer.timeout=3000
```

指定Spring扫描路径

```
@Configuration
@EnableDubbo(scanBasePackages =
"org.apache.dubbo.samples.simple.annotation.action")
@PropertySource("classpath:/spring/dubbo-consumer.properties")
@ComponentScan(value =
```





为:dubbo://192.168.43.50:20881/org.kangspace.javamindmap.dubbodemo.service.annotation.AnnotationDemoService?anyhost=true&application=dubbo-server-annotation-provider&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&interface=org.kangspace.javamindmap.dubbodemo.service.annotation.AnnotationDemoService&methods=sayHello&pid=4704&release=2.7.6&side=provider&timestamp=1614771817022) 节点,同时在/x.x.x.service下创建configurators节点;  
Provider Url中保存的是接口完全限定路径及方法名(如:methods=sayHello,sayNo)和各种配置项

- 一个接口中同名方法,不同参数在Zookeeper中的表示:

同名方法在url中只保存一个方法名

- 2. Consumer启动后会向Zookeeper订阅Provider信息,同时会将自己的信息注册到Zookeeper的/dubbo/x.x.x.service/consumers/... 【临时】节点上;  
节点

值:consumer%3A%2F%2F192.168.43.50%2F%2Fg.kangspace.javamindmap.dubbodemo.service.annotation.AnnotationDemoService%3Fapplication%3Ddubbo-server-annotation-provider%26category%3Dconsumers%26check%3Dfalse%26dubbo%3D2.0.2%26init%3Dfalse%26interface%3Dorg.kangspace.javamindmap.dubbodemo.service.annotation.AnnotationDemoService%26methods%3DsayHello%2CsayNo%26pid%3D5132%26release%3D2.7.6%26side%3Dconsumer%26sticky%3Dfalse%26timeout%3D3000%26timestamp%3D1614827842060,解码

后:consumer://192.168.43.50/org.kangspace.javamindmap.dubbodemo.service.annotation.AnnotationDemoService?application=dubbo-server-annotation-provider&category=consumers&check=false&dubbo=2.0.2&init=false&interface=org.kangspace.javamindmap.dubbodemo.service.annotation.AnnotationDemoService&methods=sayHello,sayNo&pid=5132&release=2.7.6&side=consumer&sticky=false&timeout=3000&timestamp=1614827842060

- 配置来源及优先级顺序(由高到低)

- JVM System Properties, -D 参数
- Externalized Configuration, 外部化配置(XML等)
- ServiceConfig、ReferenceConfig 等编程接口采集的配置
- 本地配置文件 dubbo.properties

- 相关问题

- 1.启动服务提示: [Dubbo]invalid constant type: 18

原因: Dubbo所依赖的Spring所使用的javassist是一个老版本的, 并不支持Lambda, 需要使用高版本, 比如3.22.0-GA

处理: 使用新版本javassist

```
org.javassist
javassist
3.22.0-GA
```

- 2.使用AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(ConsumerConfiguration.class);启动服务时, ConsumerConfiguration.class必须是static, 否则会

报.NoSuchBeanDefinitionException: No qualifying bean of type  
'xxx.ConsumerConfiguration'错误

- 3.[DUBBO] qos-server can not bind localhost:22222:

QOS配置导致有端口无法绑定:

a.xml配置

```
<dubbo:application name="dubbo-service-app" >  
  <dubbo:parameter key="qos.enable" value="false"/>  
  <dubbo:parameter key="qos.accept.foreign.ip" value="false"/>  
  <dubbo:parameter key="qos.port" value="22222"/>  
</dubbo:application>
```

[/dubbo:application](#)

b.properties配置

dubbo.application.qos.enable=true

dubbo.application.qos.port=33333

dubbo.application.qos.accept.foreign.ip=false

c.-D配置

-Ddubbo.application.qos.enable=true

-Ddubbo.application.qos.port=33333

-Ddubbo.application.qos.accept.foreign.ip=false

- 4. Dubbo中如何保证分布式事务

可在请求链路RpcContext.setAttachment()中添加自定义ID变量来标记同一事物

- 5.注册中心宕机后,Consumer依然可以和Provider通讯,因为Consumer中缓存了

Provider配置列表

- 6.dubbo工作原理

第一层: service层, 接口层, 给服务提供者和消费者来实现的

第二层: config层, 配置层, 主要是对dubbo进行各种配置的

第三层: proxy层, 服务代理层, 透明生成客户端的stub和服务端的skeleton

第四层: registry层, 服务注册层, 负责服务的注册与发现

第五层: cluster层, 集群层, 封装多个服务提供者的路由以及负载均衡, 将多个实例组合成一个服务

第六层: monitor层, 监控层, 对rpc接口的调用次数和调用时间进行监控

第七层: protocol层, 远程调用层, 封装rpc调用

第八层: exchange层, 信息交换层, 封装请求响应模式, 同步转异步

第九层: transport层, 网络传输层, 抽象mina和netty为统一接口

第十层: serialize层, 数据序列化层

工作流程:

1) 第一步, provider向注册中心去注册,并暴露服务

2) 第二步, consumer从注册中心订阅服务, 注册中心通知consumer订阅的服务列表

3) 第三步, consumer调用provider

4) 第四步, consumer和provider都异步的通知监控中心

- 7.Dubbo优雅停机:

Dubbo通过JDK的ShutdownHook实现优雅停机,所以需要使用kill PID才能优雅停机

- 8.Dubbo启动时,若Provider未启动,可配置check=false,不检查提供者,check默认为

true

- 管理控制台DubboAdmin

- Apache Thrift

# 分布式事务

<https://www.cnblogs.com/mayundalao/p/11798502.html>

<https://blog.csdn.net/squirrelanimal0922/article/details/97238041>

- LCN

- <https://www.codingapi.com/docs/>

- 模式

- LCN

- 1.原理介绍:

- LCN模式是通过代理Connection的方式实现对本地事务的操作，然后在由TxManager统一协调控制事务。当本地事务提交回滚或者关闭连接时将会执行假操作，该代理的连接将由LCN连接池管理。

- 2.模式特点:

- 该模式对代码的嵌入性为低。

- 该模式仅限于本地存在连接对象且可通过连接对象控制事务的模块。

- 该模式下的事务提交与回滚是由本地事务方控制，对于数据一致性上有较高的保障。

- 该模式缺陷在于代理的连接需要随事务发起方一共释放连接，增加了连接占用的时间。

- TCC

- 1.原理介绍:

- TCC事务机制相对于传统事务机制（X/Open XA Two-Phase-Commit），其特征在于它不依赖资源管理器(RM)对XA的支持，而是通过对（由业务系统提供的）业务逻辑的调度来实现分布式事务。主要由三步操作，Try: 尝试执行业务、Confirm:确认执行业务、Cancel: 取消执行业务。

- 2.模式特点:

- 该模式对代码的嵌入性高，要求每个业务需要写三种步骤的操作。

- 该模式对有无本地事务控制都可以支持使用面广。

- 数据一致性控制几乎完全由开发者控制，对业务开发难度要求高。

- TXC

- 1.原理介绍:

- TXC模式命名来源于淘宝，实现原理是在执行SQL之前，先查询SQL的影响数据，然后保存执行的SQL快走信息和创建锁。当需要回滚的时候就采用这些记录数据回滚数据库，目前锁实现依赖redis分布式锁控制。

- 2.模式特点:

- 该模式同样对代码的嵌入性低。

- 该模式仅限于对支持SQL方式的模块支持。

- 该模式由于每次执行SQL之前需要先查询影响数据，因此相比LCN模式消耗资源与时间要多。

- 该模式不会占用数据库的连接资源。

- 类似Seata AT模式

- Seata

- Simple Extensible Autonomous Transaction Architecture) 是阿里巴巴开源的简单可扩展自动事务架构的分布式事务中间件

- 模式

- AT( Automatic (Branch) Transaction Mode)

<https://seata.io/zh-cn/docs/dev/mode/at-mode.html>

- 前提

1.基于支持本地 ACID 事务的关系型数据库。

2.Java 应用，通过 JDBC 访问数据库。

整体机制

两阶段提交协议的演变：

一阶段：业务数据和回滚日志记录在同一个本地事务中提交，释放本地锁和连接资源。

1.解析 SQL：得到 SQL 的类型（UPDATE），表（product），条件（where name = 'TxC'）等相关的信息。

2.查询前镜像：根据解析得到的条件信息，生成查询语句，定位数据。

插入回滚日志：把前后镜像数据以及业务 SQL 相关的信息组成一条回滚日志记录，插入到 UNDO\_LOG 表中3.执行业务 SQL：更新这条记录的 name 为 'GTS'。

4.查询后镜像：根据前镜像的结果，通过 主键 定位数据

5.插入回滚日志：把前后镜像数据以及业务 SQL 相关的信息组成一条回滚日志记录，插入到 UNDO\_LOG 表中

6.提交前，向 TC 注册分支：申请 product 表中，主键值等于 1 的记录的 全局锁。

7.本地事务提交：业务数据的更新和前面步骤中生成的 UNDO LOG 一并提交。

8.将本地事务提交的结果上报给 TC。

二阶段：

提交异步化，非常快速地完成。

回滚通过一阶段的回滚日志进行反向补偿。

二阶段-回滚

1.收到 TC 的分支回滚请求，开启一个本地事务，执行如下操作。

2.通过 XID 和 Branch ID 查找到相应的 UNDO LOG 记录。

3.数据校验：拿 UNDO LOG 中的后镜与当前数据进行比较，如果有不同，说明数据被当前全局事务之外的动作做了修改。这种情况，需要根据配置策略来做处理，详细的说明在另外的文档中介绍。

4.根据 UNDO LOG 中的前镜像和业务 SQL 的相关信息生成并执行回滚的语句：

update product set name = 'TxC' where id = 1;

5.提交本地事务。并把本地事务的执行结果（即分支事务回滚的结果）上报给 TC。

二阶段-提交

- 收到 TC 的分支提交请求，把请求放入一个异步任务的队列中，马上返回提交成功的结果给 TC。

2.异步任务阶段的分支提交请求将异步和批量地删除相应 UNDO LOG 记录。

- AT 模式基于 支持本地 ACID 事务 的关系型数据库：

一阶段 prepare 行为：在本地事务中，一并提交业务数据更新和相应回滚日志记录。

二阶段 commit 行为：马上成功结束，自动 异步批量清理回滚日志。

二阶段 rollback 行为：通过回滚日志，自动 生成补偿操作，完成数据回滚。

- TCC

<https://seata.io/zh-cn/docs/dev/mode/tcc-mode.html>

- 一个分布式的全局事务，整体是 两阶段提交 的模型。全局事务是由若干分支事务组成的，分支事务要满足 两阶段提交 的模型要求，即需要每个分支事务都具备自己的：

一阶段 prepare 行为

二阶段 commit 或 rollback 行为

- TCC 模式，不依赖于底层数据资源的事务支持：

一阶段 prepare 行为：调用 自定义 的 prepare 逻辑。

二阶段 commit 行为：调用 自定义 的 commit 逻辑。

二阶段 rollback 行为：调用 自定义 的 rollback 逻辑。

所谓 TCC 模式，是指支持把 自定义 的分支事务纳入到全局事务的管理中。

- TCC模式由事务发起端在分支事务调用try(prepare)方法锁定资源,并将结果通过TM提交到TC,TC在收到所有try的结果后,通过TM调用各分支事务中的confirm或candle方法来处理提交和回滚。(TCC模式默认try成功confirm一定成功)

#### ■ ACID事务四要素

- 原子性 (Atomicity)
- 一致性 (Consistency)
- 隔离性 (Isolation)
- 持久性 (Durability)

#### ■ CAP原则

- 一个分布式系统中，一致性 (Consistency)、可用性 (Availability)、分区容忍性 (Partition tolerance)。CAP 原则指的是，这三个要素最多只能同时实现两点，不可能三者兼顾。

#### ■ 二阶段提交(Two-Phase Commit 2PC)

依赖数据库本地事务

- 两阶段提交2PC是分布式事务中最强大的事务类型之一，两段提交就是分两个阶段提交，第一阶段询问各个事务数据源是否准备好，第二阶段才真正将数据提交给事务数据源。

为了保证该事务可以满足ACID，就要引入一个协调者 (Coordinator)。其他的节点被称为参与者 (Participant)。协调者负责调度参与者的行为，并最终决定这些参与者是否要把事务进行提交或回滚。

##### ■ 1.准备阶段

协调者询问参与者事务是否执行成功，参与者发回事务执行结果。

##### ■ 2.提交阶段

如果事务在每个参与者上都执行成功，事务协调者发送通知让参与者提交事务；否则，协调者发送通知让参与者回滚事务。

需要注意的是，在准备阶段，参与者执行了事务，但是还未提交。只有在提交阶段接收到协调者发来的通知后，才进行提交或者回滚。

##### ■ 存在的问题

1.同步阻塞 所有事务参与者在等待其它参与者响应的时候都处于同步阻塞状态，无法进行其它操作。

2.单点问题 协调者在 2PC 中起到非常大的作用，发生故障将会造成很大影

响。特别是在阶段二发生故障，所有参与者会一直等待状态，无法完成其它操作。

3.数据不一致 在阶段二，如果协调者只发送了部分 Commit 消息，此时网络发生异常，那么只有部分参与者接收到 Commit 消息，也就是说只有部分参与者提交了事务，使得系统数据不一致。

4.太过保守 任意一个节点失败就会导致整个事务失败，没有完善的容错机制。

- 优点：尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。（其实也不能100%保证强一致）。

- 实现框架

- LCN中LCN,TXC模式
- Seata中AT模式

- 三阶段提交(Three-Phase Commit 3PC)

- 三阶段提交是在二阶段提交上的改进版本，3PC最关键要解决的就是协调者和参与者同时挂掉的问题，所以3PC把2PC的准备阶段再次一分为二，这样三阶段提交
- 优点：相比二阶段提交，三阶段提交降低了阻塞范围，在等待超时后协调者或参与者会中断事务。避免了协调者单点问题。阶段 3 中协调者出现问题时，参与者会继续提交事务。

缺点：数据不一致问题依然存在，当在参与者收到 preCommit 请求后等待 do commit 指令时，此时如果协调者请求中断事务，而协调者无法与参与者正常通信，会导致参与者继续提交事务，造成数据不一致。

- TCC补偿事务(Try-Confirm-Cancel)

需要实现幂等

- TCC 其实就是采用的补偿机制，其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：

Try 阶段主要是对业务系统做检测及资源预留

Confirm 阶段主要是对业务系统做确认提交，Try阶段执行成功并开始执行

Confirm阶段时，默认 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。

Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。

- 优点：跟2PC比起来，实现以及流程相对简单了一些，但数据的一致性比2PC也要差一些

性能提升：具体业务来实现控制资源锁的粒度变小，不会锁定整个资源。

数据最终一致性：基于 Confirm 和 Cancel 的幂等性，保证事务最终完成确认或者取消，保证数据的一致性。

缺点：缺点还是比较明显的，在2,3步中都有可能失败。TCC属于应用层的一种补偿方式，所以需要程序员在实现的时候多写很多补偿的代码，在一些场景中，一些业务流程可能用TCC不太好定义及处理。

- 实现框架

- LCN中TCC模式
- Seata中TCC模式

- 本地事务表(异步确保)

- 本地消息表与业务数据表处于同一个数据库中，这样就能利用本地事务来保证在对这两个表的操作满足事务特性，并且使用了消息队列来保证最终一致性。  
在分布式事务操作的一方完成写业务数据的操作之后向本地消息表发送一个消息，本地事务能保证这个消息一定会被写入本地消息表中。  
之后将本地消息表中的消息转发到 Kafka 等消息队列中，如果转发成功则将消息从本地消息表中删除，否则继续重新转发。  
在分布式事务操作的另一方从消息队列中读取一个消息，并执行消息中的操作。

- 处理流程：

- 服务消费者把业务数据和消息一同提交，发起事务。
- 消息经过MQ发送到服务提供方，服务消费者等待处理结果。
- 服务提供方接收消息，完成业务逻辑并通知消费者已处理的消息。

- 容错处理情况如下：

当步骤1处理出错，事务回滚，相当于什么都没有发生。

当步骤2、3处理出错，由于消息保存在消费者表中，可以重新发送到MQ进行重试。

如果步骤3处理出错，且是业务上的失败，服务提供者发送消息通知消费者事务失败，且此时变为消费者发起回滚事务进行回滚逻辑

- 优点：一种非常经典的实现，避免了分布式事务，实现了最终一致性。  
从应用设计开发的角度实现了消息数据的可靠性，消息数据的可靠性不依赖于消息中间件，弱化了对 MQ 中间件特性的依赖。  
缺点：与具体的业务场景绑定，耦合性强，不可公用。消息数据与业务数据同库，占用业务系统资源。业务系统在使用关系型数据库的情况下，消息服务性能会受到关系型数据库并发性能的局限。  
消息表会耦合到业务系统中，如果没有封装好的解决方案，会有很多杂活需要处理。

- MQ事务消息(最终一致性)

<https://blog.csdn.net/hosaos/article/details/90050276>

- 以RocketMQ 中间件为例，其思路大致为：  
第一阶段Prepared消息，会拿到消息的地址。第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。  
也就是说在业务方法内要想消息队列提交两次请求，一次发送消息和一次确认消息。如果确认消息发送失败了RocketMQ会定期扫描消息集群中的事务消息，这时候发现了Prepared消息，它会向消息发送者确认，所以生产方需要实现一个check接口，RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。

- 条件：

- a) 需要补偿逻辑
- b) 业务处理逻辑需要幂等

- 处理流程：

- c) 消费者向MQ发送half消息。
- d) MQ Server将消息持久化后，向发送方ack确认消息发送成功。
- e) 消费者开始执行事务逻辑。
- f) 消费者根据本地事务执行结果向MQ Server提交二次确认或者回滚。
- g) MQ Server收到commit状态则将half消息标记可投递状态。



h) 服务提供者收到该消息，执行本地业务逻辑。返回处理结果。

优点：

消息数据独立存储，降低业务系统与消息系统之间的耦合。

吞吐量优于本地消息表方案。

缺点：

一次消息发送需要两次网络请求(half消息 + commit/rollback)。

需要实现消息回查接口。

- 优点： 实现了最终一致性，不需要依赖本地数据库事务。

缺点： 实现难度大，主流MQ不支持，RocketMQ事务消息部分代码也未开源。

- Sagas事务模型（最终一致性）

- Saga模式是一种分布式异步事务，一种最终一致性事务，是一种柔性事务，有两种不同的方式来实现saga事务

## 架构模型

---

### DDD(Domain-Driven Design) 驱动领域设计

- UserInterface layer 用户接口层
- Application layer 应用层
- Domain layer 领域层(业务逻辑层)
- Infrastructure layer 基础设施层(为以上各层提供通用的技术能力:为应用层传递消息，为领域层提供持久化机制，为用户界面层绘制屏幕组件)

## 定时任务

---

### 分布式定时任务

- ElasticJob(当当开源)

## A/B机打卡

## Oauth

---

### OAuth2.0

- OAuth2.0是OAuth协议的延续版本，但不向前兼容，2012年10月正式发布为RFC 6749
- OAuth2.0是授权行业的标准协议，致力于简化客户端开发人员工作，同时为Web应用，桌面应用，移动电话等提供特定的授权流程。  
<https://oauth.net/2/>
- 参与者
  - 服务提供方：用户使用服务提供方存储受保护的资源
  - 用户：存放在服务提供方的受保护资源的拥有者
  - 客户端：需要访问服务提供方资源的第三方应用。认证前，客户端需想服务提供者申请客户端认证



- OAuth2.0 基本认证流程

(授权码方式)

- ■ 通过AppId, AppSecret请求授权地址跳转授权页, 生成Auth Code, 请求中可带state字段, scope类型
- ■ 通过用户Auth Code和AppId,AppSecret, 请求生成AccessToken (短期, 有时效性, 有权限范围scope)
- ■ 通过AccessToken 获取用户OpenId, 获取用户信息等其他API

- OAuth2.0分为2个角色

- Authorization Server负责获取用户授权并分发token
- Resource Server 负责处理API Call

- OAuth2.0的四种授权方式

- ■ 授权码(authorization-code)
- 指第三方应用先申请授权码, 然后再根据授权码获得令牌
  - authorize: response\_type=code
  - token: grant\_type=authorization\_code
- 如: A网站需要B网站的授权,

- 访问 B网站授权接口获得授权码

<https://b.com/oauth2/authorize?>

response\_type=code&

client\_id=CLIENT\_ID&

redirect\_uri=CALLBACK\_URL&

scope=read

- B网站跳转用户登录授权页, 授权成功后跳转回redirect\_url A网站, 同时返回授权码

[https://a.com/callback?code=AUTHORIZATION\\_CODE](https://a.com/callback?code=AUTHORIZATION_CODE)

- A网站后端拿到code后, 使用code和client\_id,client\_secret 申请令牌

<https://b.com/oauth2/token?>

client\_id=CLIENT\_ID&

client\_secret=CLIENT\_SECRET&

grant\_type=authorization\_code&

code=AUTHORIZATION\_CODE&

redirect\_uri=CALLBACK\_URL

响应数据:

```
{
  "access_token":"ACCESS_TOKEN",
  "token_type":"bearer",
  "expires_in":2592000,
  "refresh_token":"REFRESH_TOKEN",
  "scope":"read",
  "uid":100101,
  "info":{"...}
}
```

- 案例: 微信网页授权

- ■ 隐藏式(implicit)
- 允许直接向前端颁发令牌。这种方式没有授权码这个中间步骤, 所以称为 (授

权码)"隐藏式"(implicit)。

authorize: response\_type=token

■ 如:

- A 网站提供一个 链接, 要求用户跳转到B网站, 授权用户数据给A完整使用。

<https://b.com/oauth2/authorize?>

response\_type=token&

client\_id=CLIENT\_ID&

redirect\_uri=CALLBACK\_URL&

scope=read

- 用户跳转到B网站, 登录同意给予A网站授权。这时, B网站就会跳回redirect\_uri参数的指定网站, 并且把令牌作为URL参数, 传给A网站

[https://a.com/callback#token=ACCESS\\_TOKEN](https://a.com/callback#token=ACCESS_TOKEN)

注意, 令牌的位置是 URL 锚点 (fragment), 而不是查询字符串 (querystring), 这是因为 OAuth 2.0 允许跳转网址是 HTTP 协议, 因此存在"中间人攻击"的风险, 而浏览器跳转时, 锚点不会发到服务器, 就减少了泄漏令牌的风险。

这种方式把令牌直接传给前端, 是不安全的。因此, 只适用于一些安全要求不高的场景, 并且令牌有效期非常短, 通常就是会话期间 (session) 有效, 浏览器关掉, 令牌就失效。

- 密码式(password)

- 如果你高度信任某个应用, RFC 6749 也允许用户把用户名和密码, 直接告诉该应用。该应用就使用你的密码, 申请令牌, 这种方式称为"密码式" (password)。

grant\_type=password

■ 如:

- A网站要求用户提供B网站的用户名和密码。拿到以后, A就直接向B请求令牌

<https://oauth.b.com/token?>

grant\_type=password&

username=USERNAME&

password=PASSWORD&

client\_id=CLIENT\_ID

- B网站验证身份通过后, 直接给出令牌。注意, 这时不需要跳转, 而是把令牌放在JSON数据中, 作为Http响应, A以此拿到令牌

这种方式需要用户给出自己的用户名密码, 风险很大, 因此只适用于其他授权方式都无法采用的情况, 而且必须是用户高度信任的应用

- 客户端凭证式 (client credentials)

- 适用于没有前端的命令行应用, 即在命令行下请求令牌。

grant\_type=client\_credentials

■ 如:

- A应用在命令行向B发出请求

<https://oauth.b.com/token?>

grant\_type=client\_credentials&

client\_id=CLIENT\_ID&

client\_secret=CLIENT\_SECRET

- B网站验证通过后，直接返回令牌  
这种方式给出的令牌是针对第三方应用的，不是针对于用户，即有可能多个用户共享同一个令牌
- A拿到令牌后，请求B接口时，都必须在请求头带认证字段"Authorization"，值为"Bearer token"  
curl -H "Authorization: Bearer ACCESS\_TOKEN" \  
"https://api.b.com"  
- 案例：微信公众号
- 更新令牌
  - OAuth2.0允许用户更新令牌，具体方法：  
B网站颁发令牌时，一次颁发2个令牌，一个用户获取数据access\_token，一个用于刷新令牌refres\_token。  
令牌到期前使用refresh\_token发送请求，更新令牌  
token: grant\_type:refresh\_token  
[https://b.com/oauth/token?](https://b.com/oauth/token?grant_type=refresh_token&client_id=CLIENT_ID&client_secret=CLIENT_SECRET&refresh_token=REFRESH_TOKEN)  
grant\_type=refresh\_token&  
client\_id=CLIENT\_ID&  
client\_secret=CLIENT\_SECRET&  
refresh\_token=REFRESH\_TOKEN

## OpenId

- 概念：去中心化的网上身份认证系统，  
我们可以通过 URI（又叫 URL 或网站地址）来认证一个网站的唯一身份
- 原理简述：假设你已经拥有一个在A网站注册获得的OpenID帐号，B网站支持 OpenID帐号登录使用，而且你从未登录过。此时你在B网站的相应登录界面输入你的OpenID帐号进行登录的时候，浏览器会自动转向A网站的某个页面 进行身份验证。这时你只要输入你在A网站注册时候提供的密码登录A网站，对B网站进行验证管理（永久允许、只允许一次或者不允许）后，页面又会自动转到B 网站。

## 日常工具

---

### 开发

- Maven
  - mvn clean 清理项目  
mvn package -P [profiles,] -pl [project1,] 打包,-P指定活动的profile, -pl指定需要打包的项目  
mvn -DskipTests 跳过测试,编译测试类  
mvn -Dmaven.test.skip=true 跳过测试,不编译测试类  
mvn deploy -Dpgp.passphrase 发布jar包到nexus repository(需要生成pgp密钥及maven.setting设置server)  
mvn install 安装jar包到本地
  -

org.apache.maven.plugins  
maven-compiler-plugin

org.apache.maven.plugins  
maven-source-plugin

package

jar-no-fork

org.apache.maven.plugins  
maven-javadoc-plugin

package

jar

UTF-8

org.apache.maven.plugins  
maven-gpg-plugin

sign-artifacts  
verify

sign

org.apache.maven.plugins

maven-release-plugin  
2.5.3

true                      tag@{project.version}                      false  
-DskipTests

oss.sonatype.org  
oss.sonatype.org<https://oss.sonatype.org/content/repositories/snapshots/>

oss.sonatype.org  
oss.sonatype.org<https://oss.sonatype.org/service/local/staging/deploy/maven2/>

- mvn release:prepare 准备发布,设置发布版本,tag名称,新版本  
mvn release:perform 打出tag,并发布到远程仓库  
mvn release:rollback 回滚发布操作
- BOM
  - Bill Of Materials, 译作材料清单,BOM本身并不是一种特殊的文件格式, 而是一个普通的POM文件, 只是在这个POM中, 我们罗列的是一个工程的所有依赖和其对应的版本。
  - BOM关键信息:  
pom打包方式是pom文件  
下定义的各种依赖的版本
  - 使用方式
    - 1.通过parent引用  
4.0.0  
  
org.springframework.boot  
spring-boot-starter-parent  
2.3.4.RELEASE  
  
com.niceshot  
spring-cloud-learn  
0.0.1-SNAPSHOT  
spring-cloud-learn  
Demo project for Spring Boot
    - 2.通过dependencyManagement引用

parent只能指定一个BOM。如果想引入一个或多个BOM，这个时候，就需要使用配置。

即 不光可以用来定义BOM本身的依赖清单，也可以用作BOM的引入。因为dependencyManagement本身是做依赖管理的，Jar是一种依赖，BOM当然也是一种依赖。

```
org.springframework
spring-framework-bom
5.2.7.RELEASE
pom
import
```

## ■ Git

### ■ 创建密钥:

```
ssh-keygen -r rsa -C "email"
```

### ■ 修改已提交记录的提交信息

#修改上一次提交的信息

```
git rebase -i HEAD~
```

#将需要修改的提交前的pick改为edit

-- 修改作者

```
git commit -amend --author="name" --no-edit
```

-- 修改提交备注

```
git commit -m ""
```

-- 继续处理

```
git rebase --continue
```

-- 提交

```
git push --force
```

### ■ git checkout branchName 检出分支

```
git checkout -b branchName 创建并切换分支
```

```
git revert [-n] HAS某个提交的修改,不影响其他提交
```

```
git reset [--hard] HASH 回退到某个提交
```

```
git branch -f branchName HASH|branchName 将分支指向(重置)到某个分支
```

同 

```
git reset branchName
```

 (分支重置,期间的记录不再显示)

```
git rebase hash|branchName 将分支以 某个分支或某次提交为父分支
```

```
git rebase -i [startPoint(HASH|branchName)] endPoint 合并多次提交(前开后闭),可重新排序提交记录(可以在任意分支上操作 会新增1次提交记录)
```

同 

```
git cherry-pick hash
```

 将某次提交(提交树上的任意一次提交,除了HEAD的上游分支 可跨分支)合并到当前分支(会新增一次提交记录)

```
git clone 克隆远程仓库
```

```
git push [-f] 提交到远程分支
```

```
git push origin 提交关联到远程分支 并设置改远程分支为默认push分支
```

```
git push -u origin 提交关联到远程分支及远程仓库, 并设置默认仓库和默认push分
```

支

- Document4j(WordToPdf)

## 工具类(Jar)

- Guava(com.google.guava/guava)
  - LoadingCache(实现类似1.7-的ConcurrentHashMap,使用分段锁实现本地缓存)

## DNS

- --查询TXT记录  
nslookup -q=TXT kangspace.org

## JavaScript/ES6/jQuery

---

## Interview

---

### Http/Https原理和流程

- Http原理  
HyperText Transfer Protocol
  - 应用层协议,基于TCP/IP协议,C/S架构,基于请求/响应范式的,明文传输  
无状态协议: 客户端和服务端不需要建立长久连接  
默认端口: 80
    - TCP/IP 4层模型
      - IP 网络层
      - TCP 传输层  
TCP能保证数据包顺序传输
    - TCP报文
      - ACK、SYN和FIN这些大写的单词表示标志位, 其值要么是1, 要么是0  
ack、seq小写的单词表示序号
      - ACK: 确认序号有效
      - SYN: 发送新连接
        - SYN=1 表示这是一个连接请求, 或连接接受报文  
SYN这个标志位只有在TCP建产连接时才会被置1, 握手完成后SYN标志位被置0。
      - FIN: 释放连接
    - TCP3次握手建立连接
      - 第一次握手: 客户端发送SYN包(syn=1)到服务器, 并设置发送序号seq为X,进入SYN\_SEND状态, 等待服务器确认;  
SYN:1 seq:X
      - 第二次握手: 服务器收到SYN包, 必须确认客户的SYN(ack=X+1), 同时自己也 发送一个SYN包 (syn=1), 即SYN+ACK包, 设置发送序号seq为Y  
此时服务器进入SYN\_RECV状态;

SYN:1 ACK:1 seq:Y ack: X+1

- 第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=Y+1)，并设置发送序号seq为Z,此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。

ACK: 1 ack: Y+1 SEQ:Z

- 握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去。
- 原因：3次握手完成两个重要的功能，既要双方做好发送数据的准备工作(双方都知道彼此已准备好)，也要允许双方就初始序列号进行协商，这个序列号在握手过程中被发送和确认。

#### ■ TCP4次挥手关闭连接

- 主动方发送FIN报文，并设置发送序号seq为u（等于前面已经传送过来的数据的最后一个字节的序号加1），客户端进入FIN-WAIT-1（终止等待1）状态

FIN: 1 seq:U

- 被动方收到FIN后 置发送序号seq为V,发送确认序号ack=U+1,此时，服务端就进入了CLOSE-WAIT（关闭等待）状态

ACK:1 seq:V ack:U+1

- 被动方发送FIN+ACK，置发送序号seq为W, 确认需要ack为U+1，被动方进入LAST-ACK状态，客户端进入FIN-WAIT-2（终止等待2）状态

FIN:1 ACK:1 seq:W ack:U+1

- 主动方收到被动方FIN+ACK后，置发送需要seq为U+1,ack为W+1，发送给被动方，进入2MSL TIME-WAIT,然后连接断开。被动方收到ACK后断开连接

ACK:1 seq:U+1 ack:W+1

- 主动方发送第四次挥手后，会等待2MSL时间

MSL即Maximum Segment Lifetime，也就是报文最大生存时间

- 等待2MSL的意义？

第一，为了保证A发送的最有一个ACK报文段能够到达B。这个ACK报文段有可能丢失，因而使处在LAST-ACK状态的B收不到对已发送的FIN和ACK 报文段的确认。B会超时重传这个FIN和ACK报文段，而A就能在2MSL时间内收到这个重传的ACK+FIN报文段。接着A重传一次确认。

第二，就是防止上面提到的已失效的连接请求报文段出现在本连接中，A在发送完最有一个ACK报文段后，再经过2MSL，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失

- TCP是全双工模式，当client发出FIN报文段时，只是表示client已经没有数据要发送了，client告诉server，它的数据已经全部发送完毕了；但是，这个时候client还是可以接受来server的数据；当server返回ACK报文段时，表示它已经知道client没有数据发送了，但是server还是可以发送数据到client的；当server也发送了FIN报文段时，这个时候就表示server也没有数据要发送了，就会告诉client，我也没有数据要发送了，如果收到client确认报文段，之后彼此就会愉快的中断这次TCP连接。



- 原因: 由于TCP连接是全双工的, 因此每个方向都必须单独进行关闭。这个原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个FIN只意味着这一方向上没有数据流动, 一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭, 而另一方执行被动关闭。

- 提问

- 为什么连接的时候是三次握手, 关闭的时候却是四次挥手?

- 3次握手 是为了保证client和Server都做好发送和接收数据的准备
- 4次挥手 是确认了Client和Server 双方都不再发送数据, 都释放了连接
- 这是因为服务端的LISTEN状态下的SOCKET当收到SYN报文的连接请求后, 它可以把ACK和SYN(ACK起应答作用, 而SYN起同步作用)放在一个报文里来发送。但关闭连接时, 当收到对方的FIN报文通知时, 它仅仅表示对方没有数据发送了; 但未必所有的数据都全部发送给对方了, 所以你可能未必会马上会关闭SOCKET,也即你可能还需要发送一些数据给对方之后, 再发送FIN报文给对方来表示你同意现在可以关闭连接了, 所以它这里的ACK报文和FIN报文多数情况下都是分开发送的。

- 为什么不能用两次握手进行连接?

- 3次握手完成两个重要的功能, 既要双方做好发送数据的准备工作(双方都知道彼此已准备好), 也要允许双方就初始序列号进行协商, 这个序列号在握手过程中被发送和确认。

现在把三次握手改成仅需要两次握手, 死锁是可能发生的。作为例子, 考虑计算机S和C之间的通信, 假定C给S发送一个连接请求分组, S收到了这个分组, 并发送了确认应答分组。按照两次握手的协定, S认为连接已经成功地建立了, 可以开始发送数据分组。可是, C在S的应答分组在传输中被丢失的情况下, 将不知道S 是否已准备好, 不知道S建立什么样的序列号, C甚至怀疑S是否收到自己的连接请求分组。在这种情况下, C认为连接还未建立成功, 将忽略S发来的任何数据分组, 只等待连接确认应答分组。而S在发出的分组超时后, 重复发送同样的分组。这样就形成了死锁。

类似SYN攻击

- 如果已经建立了连接, 但是客户端突然出现故障了怎么办

- 服务端有超时计时器维持, 当超过维持时间后会断开连接

- 发送了FIN只是表示这端不能继续发送数据(应用层不能再调用send发送), 但是还可以接收数据

- 应用层如何知道对端关闭

- 通常, 在最简单的阻塞模型中, 当你调用recv时, 如果返回0, 则表示对端关闭。在这个时候通常的做法就是也调用close, 那么TCP层就发送FIN, 继续完成四次握手。如果你不调用close, 那么对端就会处于FIN\_WAIT\_2状态, 而本端则会处于CLOSE\_WAIT状态。

- [https://blog.csdn.net/qg\\_38950316/article/details/81087809](https://blog.csdn.net/qg_38950316/article/details/81087809)  
<https://www.cnblogs.com/kindnull/p/10307333.html>

- TCP/IP 与 Socket之间的关系

- socket是对TCP/IP协议的封装，Socket本身并不是协议，而是一个调用接口(API)，通过Socket，我们才能使用TCP/IP协议
- Http工作过程  
(一次HTTP操作称为一个事务)
  - URL地址解析:  
protocol ://[username:password@]hostname[:port] / path /  
[;parameters][?query]#fragment

DNS解析域名为IP

- 2. 封装HTTP请求数据包

把以上部分结合本机自己的信息，封装成一个HTTP请求数据包

- 3.封装成TCP包，建立TCP连接（TCP的三次握手）
- 4. 客户机发送请求命令

建立连接后，客户机发送一个请求给服务器，请求方式的格式为：请求类型、统一资源标识符（URL）、协议版本号，后边为MIME信息包括请求修饰符、客户机信息和可能的内容。

GET示例：

GET <http://baidu.com> HTTP/1.1

Connection: Keep-Alive

Accept: /

User-Agent: Microsoft-CryptoAPI/10.0

Host: baidu.com

POST示例：

POST <http://tracking.miui.com/track/v1> HTTP/1.1

gzip: 0

Content-Type: application/x-www-form-urlencoded

User-Agent: Dalvik/2.1.0 (Linux; U; Android 9; MIX 2 MIUI/V11.0.2.0.PDECNXM)

Host: tracking.miui.com

Connection: Keep-Alive

Accept-Encoding: gzip

Content-Length: 2626

...请求实体

- 5. 服务器响应

服务器接到请求后，给予相应的响应信息，其格式为一个状态行，包括信息的协议版本号、状态码，状态码描述，后边是MIME信息包括服务器信息、实体信息和可能的内容。

实体消息是服务器向浏览器发送头信息后，它会发送一个空白行来表示头信息的发送到此为结束，接着它就以Content-Type应答头信息所描述的格式发送用户所请求的实际数据

HTTP/1.1 200 OK

Server: NWS\_Oversea\_Domestic\_Mid

Connection: keep-alive

Date: Wed, 04 Dec 2019 02:39:19 GMT

Last-Modified: Wed, 04 Dec 2019 01:16:44 GMT

Content-Type: application/ocsp-response

Content-Length: 1574

...实体主体数据

- 6. 服务器关闭TCP连接

一般情况下，一旦Web服务器向浏览器发送了请求数据，它就要关闭TCP连接，然后如果浏览器或者服务器在其头信息加入了这行代码

Connection:keep-alive

TCP连接在发送后将仍然保持打开状态，于是，浏览器可以继续通过相同的连接发送请求。保持连接节省了为每个请求建立新连接所需的时间，还节约了网络带宽。

在收到任意一方Connection:close 后关闭连接

- 响应码

- 1xx

- 100 Continue

服务器未拒绝

- 101 Switching Protocols

转换协议

- 2xx

- 202 Accepted 供处理的请求已被接受，但是处理未完成。

- 203 Non-authoritative Information 文档已经正常地返回，但一些应答头可能不正确，因为使用的是文档的拷贝。

- 204 No Content 没有新文档。浏览器应该继续显示原来的文档。如果用户定期地刷新页面，而Servlet可以确定用户文档足够新，这个状态代码是很有用的。

- 205 Reset Content 没有新文档。但浏览器应该重置它所显示的内容。用来强制浏览器清除表单输入内容。

- 206 Partial Content 客户发送了一个带有Range头的GET请求，服务器完成了它。

- 3xx

- 300 Multiple Choices 多重选择。链接列表。用户可以选择某链接到达目的地。最多允许五个地址。

- 301 Moved Permanently 所请求的页面已经转移至新的url。

- 302 Found 所请求的页面已经临时转移至新的url。

如果是Post时会被静默重定向为GET

- 303 See Other 所请求的页面可在别的url下被找到。

HTTP1.1定义浏览器对303状态码的处理跟原来浏览器对HTTP1.0的302状态码的处理方法一样

- 304 Not Modified 未按预期修改文档。客户端有缓冲的文档并发出了一个条件性的请求（一般是提供If-Modified-Since头表示客户只想比指定日期更新的文档）。服务器告诉客户，原来缓冲的文档还可以继续使用。

- 305 Use Proxy 客户请求的文档应该通过Location头所指明的代理服务器提取。

- 306 Unused 此代码被用于前一版本。目前已不再使用，但是代码依然被保留。

- 307 Temporary Redirect 被请求的页面已经临时移至新的url。Post不会被转为Get,会询问用户是否在新URI上发起Post

- 4xx

- 400 Bad Request 服务器未能理解请求。

- 401 Unauthorized 被请求的页面需要用户名和密码。

- 402 Payment Required 此代码尚无法使用。

- 403 Forbidden对被请求页面的访问被禁止。

- 404 Not Found 服务器无法找到被请求的页面。

- 405 Method Not Allowed 请求中指定的方法不被允许。

- 5xx

- 500 Internal Server Error 请求未完成。服务器遇到不可预知的情况

- 501 Not Implemented 请求未完成。服务器不支持所请求的功能。

- 502 Bad Gateway 请求未完成。服务器从上游服务器收到一个无效的响应。

- 503 Service Unavailable 请求未完成。服务器临时过载或宕机。
- 504 Gateway Timeout 网关超时。
- 505 HTTP Version Not Supported 服务器不支持请求中指定的HTTP协议版本。

## ■ Https原理

### ■ Http+SSL/TLS

SSL: SecureSocketLayer

TLS: SSL3.0 Transport Layer Security

- 默认端口号: 443
- TLS 握手在 TCP握手之后进行

### ■ 非对称加密

- 使用非对称加密（RSA）的公钥对（对称加密）密钥进行加密  
对随机数-预主密钥加密

### ■ 对称加密

- 使用对称加密(AES,DES,3DES等)对传输数据进行加密  
使用(随机数1,随机数)

### ■ 数字摘要

- MD5, SHA1等

### ■ 数字签名技术

- 数字签名建立在公钥加密体制基础上，是公钥加密技术的另一类应用。它把公钥加密技术和数字摘要结合起来，形成了实用的数字签名技术。

### ■ 数字证书内容

- 包括了加密后服务器的公钥、权威机构的信息、服务器域名，还有经过CA私钥签名之后的证书内容（经过先通过Hash函数计算得到证书数字摘要，然后用权威机构私钥加密数字摘要得到数字签名），签名计算方法以及证书对应的域名。

### ■ 问题

- 如何保证服务器给客户端下发的公钥是真正的公钥，而不是中间人伪造的公钥？
  - 当客户端收到这个证书之后，使用本地配置的权威机构的公钥对证书进行解密得到服务端的公钥和证书的数字签名，数字签名经过CA公钥解密得到证书信息摘要。
  - 2.用证书签名的方法计算一下当前证书的信息摘要，与收到的信息摘要作对比，如果一样，表示证书一定是服务器下发的，没有被中间人篡改过。因为中间人虽然有权威机构的公钥，能够解析证书内容并篡改，但是篡改完成之后中间人需要将证书重新加密，但是中间人没有权威机构的私钥，无法加密，强行加密只会导致客户端无法解密，如果中间人强行乱修改证书，就会导致证书内容和证书签名不匹配。

### ■ 优势

- 信任主机的问题.
- 防止通讯过程中的数据的泄密和被篡改

### ■ 劣势

- 在黑客攻击、拒绝服务工具、服务器劫持方面起不到作用

- SSL证书的信用链体系并不安全，特别是在某些国家可以控制CA根证书的情况下，中间人攻击一样可行
  - HTTPS连接缓存不如HTTP高效，流量成本高
  - HTTPS连接服务器端资源占用高很多，支持访客多的网站需要投入更大的成本
- HTTPS协议握手阶段比较费时，对网站的响应速度有影响，影响用户体验
- SSL连接建立过程
  - Client 向 Server发送 Hello , 随机码1,客户端支持的加密算法列表  
Client -> Hello ,random number1 ,cipher suites ->Server
  - Server接收数据后响应client，返回随机数2和匹配的加密算法  
Server -> random number2,matched cipher suites - client
  - 随即 Server给Client发送数字证书报文。  
Server -> server certificate -> Client
    - 客户端解析证书，验证公钥是否有效，如颁发机构，过期时间等，若异常，则弹出警告提示证书存在问题，若正常，则生成随机值（预主密钥）
  - 认证证书:
- 验证证书是否有效，通过浏览器/操作系统内置信任的根证书校证书是否有效,先从系统(Windows)或者浏览器内置（Firefox）检查证书链是否正确(通过OCSP/CRL结合内置的truststore验证),
- 浏览器尝试查CRL和OCSP
- 检查证书有效期
  - 4.检查网站域名是否与证书域名一致
    - IE和Chrome是通过内置在Windows系统中的TrustStore来管理根证书（当然自己也可以手动导入自签证书，浏览不会认可的因为有OCSP和CRL--之后细讲）；而Firefox则是内置在自己的浏览中。
  - CRL(证书吊销列表)和OCSP(在线证书检查)
    - 5. 客户端通过认证后，通过随机数1，随机数2，预主密钥组装会话密钥，然后通过证书的公钥加密 预主密钥  
client -> assemble session secret key = random number1+random2+premaster;  
encrypt (premaster secret) with public key ->Server
    - 6. 传送加密信息：公钥加密的预主密钥
    - 7. 服务器用私密解密6中的预主密钥，然后 通过随机数1，随机数2,预主密钥 组装会话密钥
    - 8. 客户端通过会话密钥加密一条消息发送给服务端，主要验证服务端是否正常接收客户端加密数据
    - 9. 服务端通过会话密钥加密一条消息回传给客户端，如果客户端能正常接收则SSL层连接建立完成
- Https工作过程
  - 同Http工作过程，只是在TCP 握手后增加了TLS层握手
- Websocket原理

- ws: 80  
wss: 443
- Request
  - GET /chat HTTP/1.1  
Host: server.example.com  
Upgrade: websocket  
Connection: Upgrade  
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==  
Sec-WebSocket-Protocol: chat, superchat  
Sec-WebSocket-Version: 13  
Origin: <http://example.com>
- Response
  - HTTP/1.1 101 Switching Protocols  
Upgrade: websocket  
Connection: Upgrade  
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=  
Sec-WebSocket-Protocol: chat
- 是真正的全双工方式，建立连接后客户端与服务器端是完全平等的，可以互相主动请求。而HTTP长连接基于HTTP，是传统的客户端对服务器发起请求的模式。
- tomcat实现websocket
  - @ServerEndpoint(value = "/WSChat")
  - @OnOpen  
public void onOpen(Session session) {}
  - @OnClose  
public void onClose() {}
  - @OnMessage  
public void onMessage(Session session, String message) {}
- js 实现websocket
  - websocket = new WebSocket(wsUri,"protocol");
  - websocket.onopen =function(event){};
  - websocket.onmessage =function(event){}
  - websocket.onerror =function(event){};
  - websocket.onclose =function(event){};
  - Socket.readyState
    - 只读属性 readyState 表示连接状态，可以是以下值：
      - 0 - 表示连接尚未建立。
      - 1 - 表示连接已建立，可以进行通信。
      - 2 - 表示连接正在进行关闭。
      - 3 - 表示连接已经关闭或者连接不能打开。
  - Socket.bufferedAmount
    - 只读属性 bufferedAmount 已被 send() 放入正在队列中等待传输，但是还没有发出的 UTF-8 文本字节数。

- 浏览器网页同时最大发起http请求数
  - 15-20个

## 微信公众号/小程序开发

## 业务场景实践方案

---

### 高并发场景方案

### 秒杀场景方案

### 轮询

- ajax 轮询
- long poll
  - 发起请求后，服务端等待有数据后返回，然后再次发起请求
- websocket

## 压测相关

---

### Jemeter

### LoadRunner

## 运维监控

---

### PINPOINT请求链路监控

- pinpoint-controller
  - pinpoint-controller,pinpoint-web,pinpoint-agent[javaagent]
- pinpoint-web
- pinpoint-agent[javaagent]  
最终程序启动使用--javaagent配置pinpoint-agent

## AWS

---

### Serverless

- Lambda

**DynamoDB**

**DocumentDB**

**Elemental MediaConverter**

**S3**

**大数据**

---

**ElasticSearch**

**Hive**

**Spark**

**Scala**

**推送**

---

**友盟**

**极光**

**Linux**

---

**查询占用端口:**

`netstat -anp 端口号`

`lsof -i 端口号`

**查看日志文件某个字符串出现的行数**

`cat x.log | grep xx | cut -d '分隔符' -?f | uniq -c | sort -n -r`

**uniq 对行去重 -c 在列前统计出现次数**