

我的学习日志

目录

[我的学习日志](#)

[目录](#)

[算法设计](#)

[描述算法复杂度](#)

[常见算法](#)

[栈](#)

[栈的实现](#)

[栈的应用](#)

[后缀表达式](#)

[队列](#)

[队列的实现](#)

[队列假溢出](#)

[循环队列](#)

[递归](#)

[递归的特点](#)

[递归与栈的关系](#)

[数组](#)

[二分查找](#)

[移除元素](#)

[滑动窗口法](#)

[数列排序算法](#)

[插入排序](#)

[归并排序](#)

[快速排序](#)

[链表](#)

[链表的类型](#)

[链表存储方式](#)

[删除节点](#)

[性能分析](#)

[移除链表元素问题](#)

[设计链表](#)

[反转链表](#)

[哈希表](#)

[常见的三种哈希结构](#)

[两个数组的交集\(SET\)](#)

[两数之和的下标\(MAP\)](#)

[四数组相加](#)

[获取验证码 easy-captcha](#)

[Maven依赖](#)

[使用方法](#)

[Git版本管理](#)

[Git 与 SVN 区别](#)

[Git安装](#)

[Linux 平台上安装](#)

[Git 配置](#)

[Git使用入门](#)

[分支管理](#)

[保存点](#)

[cherry-pick](#)

[MYSQL详解](#)

注意规范

修改表

修改表名

增加表字段

修改表字段

删除表字段

MySql数据管理

外键(了解即可)

物理外键(不建议使用)

DML语言(全部记住)

添加

修改

删除

DQL查询语言

联表查询JoinON详解

自连接

分页和排序

子查询

常用函数

聚合函数及分组过滤

事务

1.事务简介

2.事务四大特征/原则

3.MYSQL 事务处理

索引

索引分类

索引使用

索引原则

权限管理和备份

用户管理

Mysql备份

数据库三大范式

Mybatis XML映射文件详解

基本SQL XML语句

Select语句 -查询

Insert、 Update、 Delete 语句

Foreach语句

Sql语句

#{}/\${} -字符串替换 语句

ResultMap -结果映射

id & result

Association -关联

Collection -集合

Discriminator -鉴定器

动态 SQL

if语句

choose、 when、 otherwise语句

trim、 where、 set语句

TK.mybatis框架使用

BaseMapper方法:

IdsMapper方法:

ConditionMapper方法:

JWT (JSON Web Token)跨域身份验证

结构解析

使用JWT

1.Maven依赖

2.利用数据，生成对应Token

3.使用对应Token，获取数据

SpringBoot+Mybatis整合

Maven依赖

操作方法

- 1.编写实体类
- 2.编写mapper接口
- 3.编写mapper xml映射文件
- 4.编写service接口
- 5.编写service实现类
- 6.编写controller文件
- 7.配置property文件(或者yml文件)

Mybatis逆向工程

操作方法

- 一.Maven依赖
- 二.配置generatorConfig.xml文件
- 三.编写生成器

PageHelper 分页

Maven依赖

操作方法

- 1.配置文件
- 2.编写对应需要分页controller层方法

Redis基础学习

Linu下安装

服务器

- 打开服务器
 访问服务器

基础知识

数据类型

- 1.String

 SET命令
 GET命令
 GETSET命令
 KEYS命令
 STRLEN命令
 MSET命令
- 2.Hash

 HMSET命令
 HGETALL命令
 HGET命令
 HDEL命令
 HLEN命令
- 3.List

 LPUSH命令
 LRANGE命令
 LSET命令
 LLEN命令
 LIINDEX命令
 LPOP/RPOP命令
 RPOPLPUSH命令
- 4.Set

- SADD命令
 SMEMBER命令
 SCARD命令
 SISMEMBER命令
 SREM命令
 SRANDMEMBER命令
 对比命令

 SDIFF命令
 SINTER命令

SUNION命令

5.Zset

ZADD命令

ZSCARD命令

ZRANGE/ZREVRANGE命令

三种特殊数据类型

Geospatial - 地理位置

Geoadd - 添加地理位置

Geopos - 查询地理位置

Geodist - 计算地理之间距离

Georadius - 寻找附近(通过经纬度)

Georadiusbymember - 寻找附近(通过指定元素)

Hyperloglog数据结构 - 基数统计

Bitmaps - 位图(位存储)

发布订阅

SUBSCRIBE命令

PUBLISH命令

UNSUBSCRIBE命令

事务

事务执行步骤

事务监控

Redis的持久化

RDB(Redis DataBase)

AOF(Append Only File)

Redis主从复制

配置环境

层层链路

哨兵模式

Jedis

Maven依赖

编码测试

SpringBoot+Redis

Maven依赖

配置方法

1.配置properties文件(yaml文件)中的redis环境参数

2.配置RedisConfig类

3.配置RedisUtil类

4.创建实体类

5.在Controller中使用 (读写)

MD5+Salt+Hash散列进行数据加密

主要方法

注册用户时

Shiro安全框架

主要功能

细分功能

Maven依赖

快速入门语句

SpringBoot继承Shiro

Maven依赖

创建Realm类

创建ShiroConfig配置类

Controller类

Shiro+Thymeleaf页面整合

常用标签:

The has Permission tag

The authenticated tag

The hasRole tag

权限、角色访问控制

方法一：直接在页面控制（以Thymeleaf为例）

方法二：Controller代码层中控制

方法三：代码注释控制

JAVA设计模式

OOP七大原则

单例模式

工厂模式

Swagger

简介

SpringBoot集成Swagger

配置Swagger

 Swagger配置扫描接口

 配置API的分组

 扫描实体类

Thymeleaf

 Maven依赖

 首次使用

 语法规则

 thymeleaf公共页面元素抽取

Vue

 环境配置

 vue.js基础

 Vue组件

 Axios异步通讯

 Slot插槽

 自定义事件

 WebPack打包

 Vue-router路由

 路由嵌套

 路由参数传递

 1.通过路由直接获取参数值

 2.通过props进行获取参数

 路由重定向

 参数检验

 ElementUI

 在单个页面设置Body背景颜色:

 单个页面设置显示标题

 1.修改项目title

 2.单个页面设置不同标题

 设置404页面

 路由钩子

 在钩子函数中使用异步请求

 Vue前端传输表单

 设置VUE页面切换动画

 Vue常见问题

 Vue3

 创建项目

 Setup()

 总结

SpringBoot Result对象

 一：定义响应码枚举

 二：创建返回对象Result实体（泛型）

 四：返回结果数据格式封装 / 响应结果生成工具

 五：返回Result功能测试

 maven pom文件配置

 UUID（通用唯一识别码）

 UUID生成工具类

 RESTful项目登录模块的实现

- 配置跨域访问配置类
- 实现方法
- Ehcache 缓存
 - ehcache 和 redis 比较
 - Maven引入依赖
 - 项目整合
- Nuxt.js
 - 安装和运行
 - 修改默认端口号
 - 项目文件
 - 全局css文件设置
 - 路由功能
 - 动态路由跳转
 - 默认模板和默认模块
 - 默认模板
 - 默认布局
 - 自定义404页面
- Front-End 前端设计
 - position(定位)
 - static:静态定位
 - relative:相对定位
 - fixed:固定定位
 - absolute:绝对定位
 - sticky:粘性定位
 - 重叠元素
 - 浮动和清除
 - float 属性
 - clear 属性
 - Flexbox 布局模块
- TweenMax(GSAP)动画
 - 引用/导入
 - TweenLite和TweenMax区别
 - TweenMax对象.play()
 - 动画结构
 - TweenMax.to()
 - TweenMax.from()
 - TweenMax.fromTo()
 - TweenMax.staggerTo()
 - TweenMax.staggerFromTo()
 - TweenMax.delayedCall()
 - TweenMax.set()
 - 3D效果
 - 动画参数
 - repeat
 - delay
 - 实例属性
 - .timeline
- Locomotive Scroll(视差滚动)
 - 安装引用
 - 使用

算法设计

算法 (Algorithm) 是指用来操作数据、解决程序问题的一组方法。对于同一个问题，使用不同的算法，也许最终得到的结果是一样的，但在过程中消耗的资源和时间却会有很大的区别。

如何去衡量不同算法之间的优劣呢？

主要还是从算法所占用的「**时间**」和「**空间**」两个维度去考量。

- **时间维度**：是指执行当前算法所消耗的时间，我们通常用「**时间复杂度**」来描述。
- **空间维度**：是指执行当前算法需要占用多少内存空间，我们通常用「**空间复杂度**」来描述。

描述算法复杂度

在描述算法复杂度时，经常用到 $O(1)$, $O(n)$, $O(\log n)$, $O(n \log n)$ 来表示对应算法的时间复杂度，这里进行归纳一下它们代表的含义：

这是算法的时空复杂度的表示。不仅仅用于表示时间复杂度，也用于表示空间复杂度。

O后面的括号中有一个函数，指明某个算法的耗时/耗空间与数据增长量之间的关系。其中的**n**代表输入数据的量。

线性阶 $O(n)$ ，就代表数据量增大几倍，耗时也增大几倍。比如常见的遍历算法。

平方阶 $O(n^2)$ ，就代表数据量增大 n 倍时，耗时增大 n 的平方倍，这是比线性更高的时间复杂度。比如冒泡排序，就是典型的 $O(n^2)$ 的算法，对 n 个数排序，需要扫描 $n \times n$ 次。

对数阶 $O(\log n)$ ，当数据增大 n 倍时，耗时增大 $\log n$ 倍（这里的 \log 是以 2 为底的，比如，当数据增大 256 倍时，耗时只增大 8 倍，是比线性还要低的时间复杂度）。二分查找就是 $O(\log n)$ 的算法，每找一次排除一半的可能，256 个数据中查找只要找 8 次就可以找到目标。

线性对数阶 $O(n \log n)$ ，就是 n 乘以 $\log n$ ，当数据增大 256 倍时，耗时增大 $256 \times 8 = 2048$ 倍。这个复杂度高于线性低于平方。归并排序就是 $O(n \log n)$ 的时间复杂度。

常数阶 $O(1)$ 就是最低的时空复杂度了，也就是耗时/耗空间与输入数据大小无关，无论输入数据增大多少倍，耗时/耗空间都不变。哈希算法就是典型的 $O(1)$ 时间复杂度，无论数据规模多大，都可以在一次计算后找到目标（不考虑冲突的话）

常见算法

- 1、排序算法：快速排序、归并排序、计数排序
- 2、搜索算法：回溯、递归、剪枝
- 3、图论：最短路径、最小生成树、网络流建模
- 4、动态规划：背包问题、最长子序列、计数问题
- 5、基础技巧：分治、倍增、二分法、贪心算法

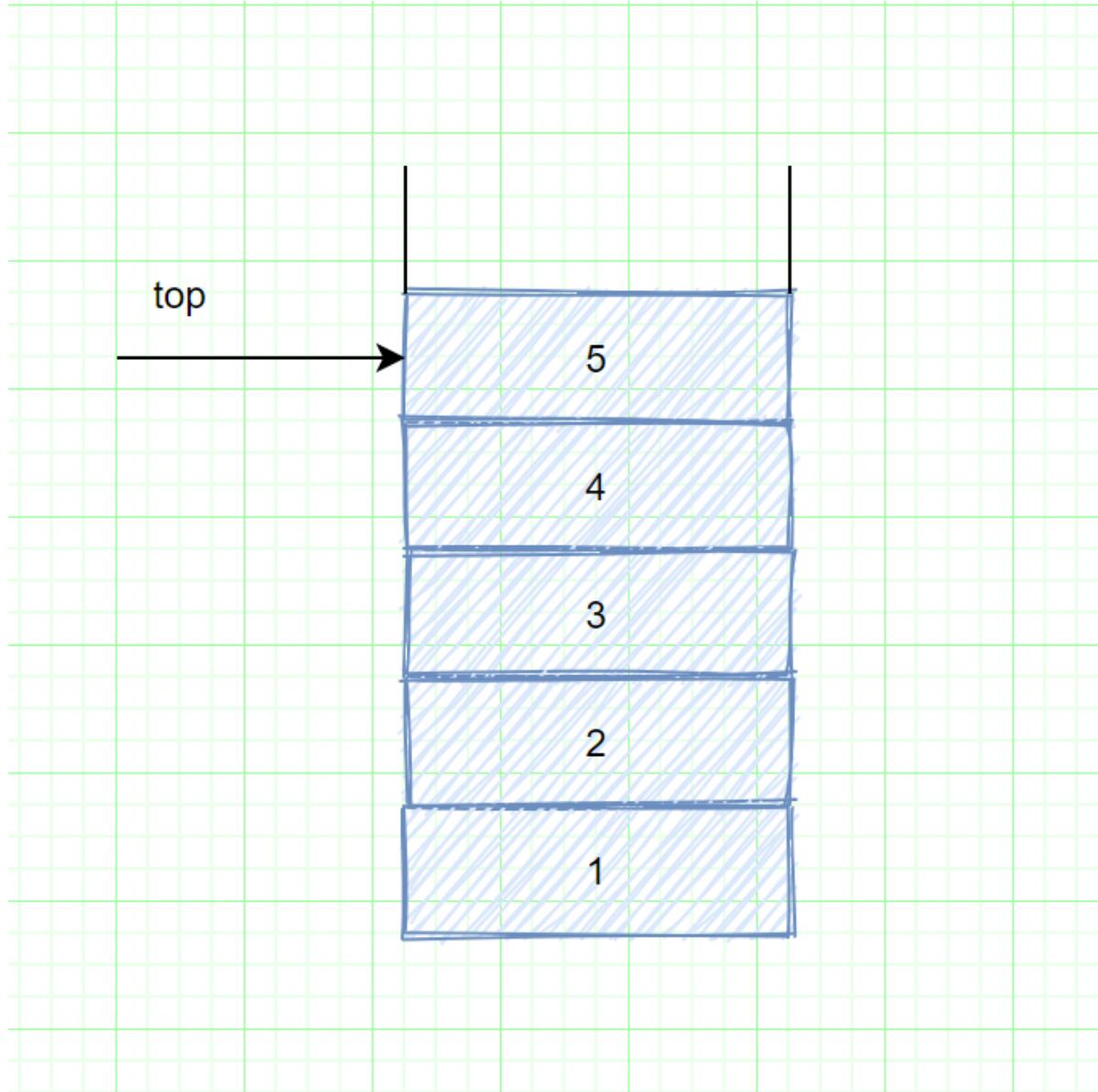
数据结构：

- 1、数组和链表
- 2、栈与队列
- 3、树和图
- 4、哈希表
- 5、大/小根堆，可并堆
- 6、字符串：字典树、后缀树

栈

栈 (stack) 是限制插入和删除只能在一个位置上进行的表，该位置是表的末端叫做栈的顶 (top)，对栈的基本操作有push(进栈)和pop(出栈)，前者相当于插入，后者则是删除最后插入的元素。

一般的模型是，存在某个元素位于栈顶，而该元素是唯一可见元素。



栈的实现

因为栈是一个表，因此能够实现表的方法都可以实现栈，ArrayList和LinkedList都可以支持栈操作。

```
Stack<TreeNode> stack = new Stack<TreeNode>();
```

栈的应用

栈在现实中应用场景很多，大家在刷题时就可以注意到，很多题目都可以用栈来解决的。

比较常用的情景，数字表达式的求值。

后缀表达式

又被称为逆波兰表示.

普通四则运算中括号也是其中的一部分，先乘除后加减使运算变的复杂.使用后缀表达式则可以无需考虑括号.

将中缀表达式(普通)转为后缀表达式:

中缀: $9 + (3 - 1) * 3 + 10 / 2$

后缀: $9\ 3\ 1\ -\ 3\ *\ +\ 10\ 2\ /$

转换规则:

- 1.从左到右遍历中缀表达式的每个数字和符号，若是数字就输出（直接成为后缀表达式的一部分，不进入栈）
- 2.若是符合则判断其与栈顶符号的优先级，是右括号 或 重复出现且低于栈顶的元素，则栈顶元素依次出栈并输出，直到重复元素出栈完毕，当前再元素入栈。
- 3.遵循以上两条直到输出后缀表达式为止。



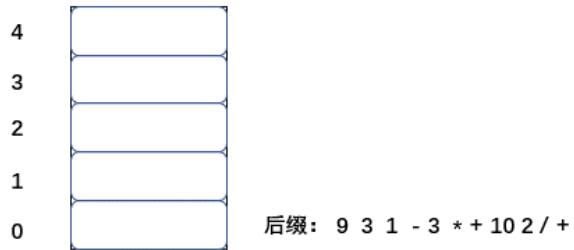
计算规则:

中缀: $9 + (3 - 1) * 3 + 10 / 2 = 20$

后缀: $9\ 3\ 1\ -\ 3\ *\ +\ 10\ 2\ /$

1.从左到右遍历表达式的每个数字和符号，如果是数字就进栈

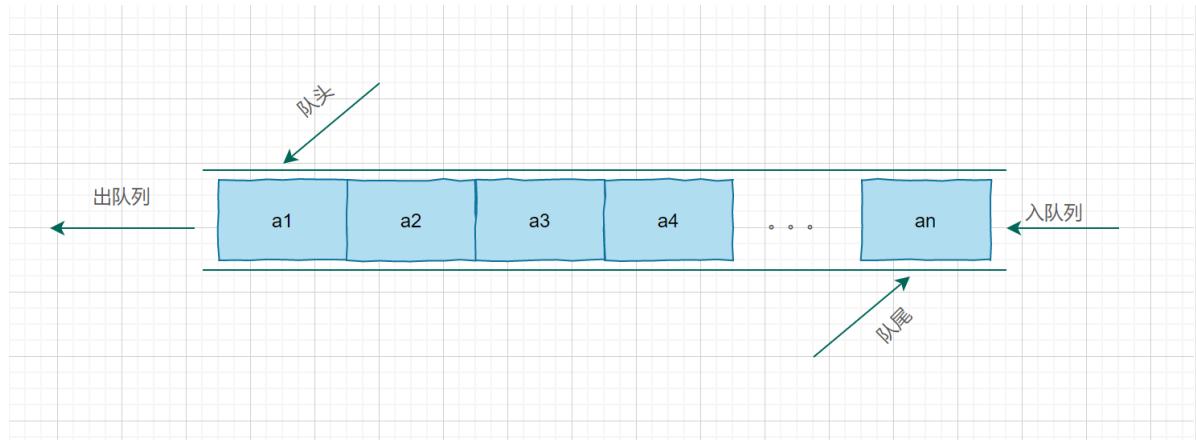
2.如果是符号就将栈顶的两个数字出栈，用**下位元素与栈顶元素进行计算**,并将结果入栈，一直到获得最终结果。



队列

像栈一样，队列（queue）也是表。

然而使用队列时插入在一端进行而删除在另一端进行，遵守先进先出的规则。所以队列的另一个名字是（FIFO）。



队列的实现

队列我们在树的层次遍历时经常使用，后面我们写到树的时候会给大家整理框架。队列同样也可以由数组和LinkedList实现，刷题时比较常用的方法是

```
Queue<TreeNode> queue = new LinkedList<TreeNode>();
```

队列假溢出

数组长度为5，我们放入了[1,2,3,4,5]我们将1, 2出队，此时如果继续加入6时，因为数组末尾元素已经被占用，再向后加则会溢出，但是我们的下标0, 和下标1还是空闲的。所以我们把这种现象叫做“假溢出”。

例如，我们在学校里面排队洗澡一人一个格，当你来到澡堂发现前面还有两个格，但是后面已经满了，你是去前面洗，还是等后面格子的哥们洗完再洗？肯定是去前面的格子洗。除非澡堂的所有格子都满了。我们才会等。

使用循环队列可以解决假溢出问题。

循环队列

用来解决假溢出的方法就是后面满，就再从头开始，也就是头尾相接的循环，我们把队列的这种头尾相接的顺序存储结构成为循环队列。

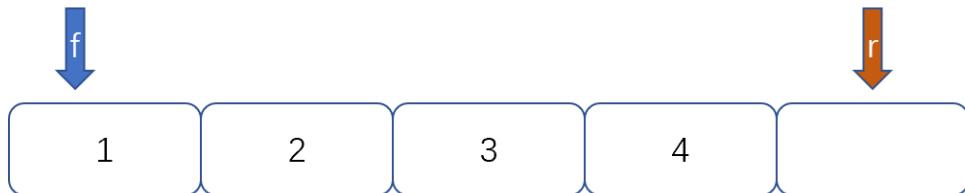


队列为空时 $\text{front} == \text{rear}$ ，队列满时也是 $\text{front} == \text{rear}$ ，区分满和空：

当队列为空时， $\text{front} == \text{rear}$, 当队列满是我们保留一个元素空间，也就是说，队列满时，数组内还有一个空间。再根据以下公式则能够判断队列满没满了。

$(\text{rear}+1) \% \text{queuesize} == \text{front}$

其中 queuesize ,代表队列的长度，上图为5。我们来判断上面两张图是否满。 $(4+1) \% 5 == 0$,
 $(1+1) \% 5 == 3$ 所以两种情况都是满的。



递归

递归，在计算机科学中是指一种通过重复将问题分解为同类的子问题而解决问题的方法。简单来说，递归表现为函数调用函数本身。

递归最恰当的比喻，就是查词典。我们使用的词典，本身就是递归，为了解释一个词，需要使用更多的词。当你查一个词，发现这个词的解释中某个词仍然不懂，于是你开始查这第二个词，可惜，第二个词里仍然有不懂的词，于是查第三个词，这样查下去，直到有一个词的解释是你完全能看懂的，那么递归走到了尽头，然后你开始后退，逐个明白之前查过的每一个词，最终，你明白了最开始那个词的意思。

简单的代码说明：

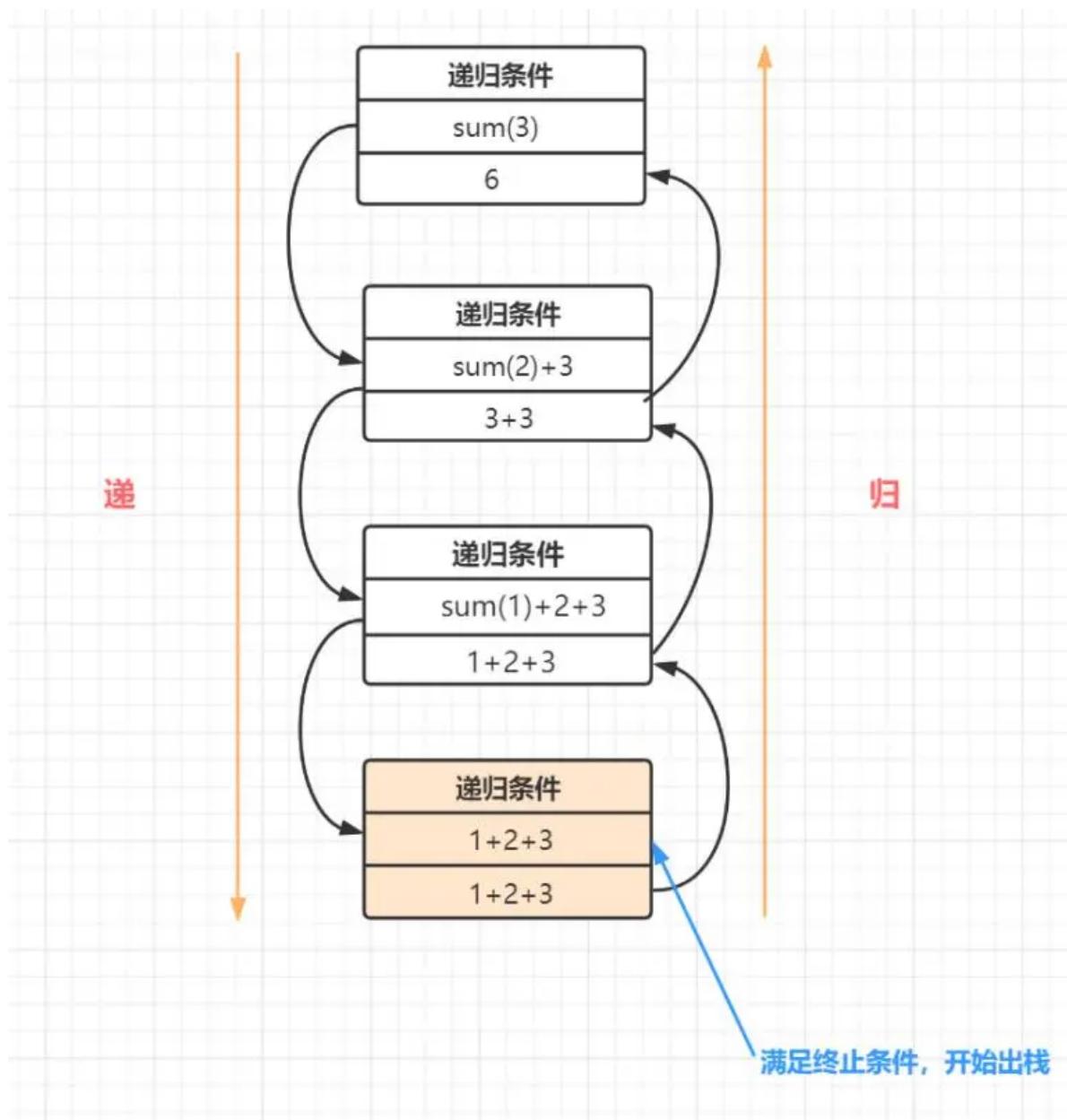
```
public int sum(int n) {  
    //终止条件  
    if (n <= 1) {  
        return 1;  
    }  
    //自身调用  
    return sum(n - 1) + n;  
}
```

递归的特点

递归有两个显著的特征，终止条件和自身调用：

- 自身调用：原问题可以分解为子问题，子问题和原问题的求解方法是一致的，即都是调用自身的同一个函数。
- 终止条件：递归必须有一个终止的条件，即不能无限循环地调用本身。

递归与栈的关系



$\text{sum}(5)$
 $=> 5 + \text{sum}(4)$
 $=> 5 + (4 + \text{sum}(3))$
 $=> 5 + (4 + (3 + \text{sum}(2)))$
 $=> 5 + (4 + (3 + (2 + \text{sum}(1))))$
 $=> 5 + (4 + (3 + (2 + 1)))$
 $=> 5 + (4 + (3 + 3))$
 $=> 5 + (4 + 6)$
 $=> 5 + 10$
 $=> 15$

递

归

数组

二分查找

二分查询法就是讲数据进行两份化处理. 左边区域left 右边区域right. 再取中间值, 进行比较, 判断要求数据在哪个区域

大家写二分法经常写乱, 主要是因为**对区间的定义没有想清楚, 区间的定义就是不变量**。要在二分查找的过程中, 保持不变量, 就是在while寻找中每一次边界的处理都要坚持根据区间的定义来操作, 这就是**循环不变量规则**。

写二分法, 区间的定义一般为两种, 左闭右闭即[`left, right`], 或者左闭右开即[`left, right`)。

例题:给定一个 n 个元素有序的 (升序) 整型数组 `nums` 和一个目标值 `target` , 写一个函数搜索 `nums` 中的 `target`, 如果目标值存在返回下标, 否则返回 -1。

示例 1:

```
输入: nums = [-1,0,3,5,9,12], target = 9
输出: 4
解释: 9 出现在 nums 中并且下标为 4
```

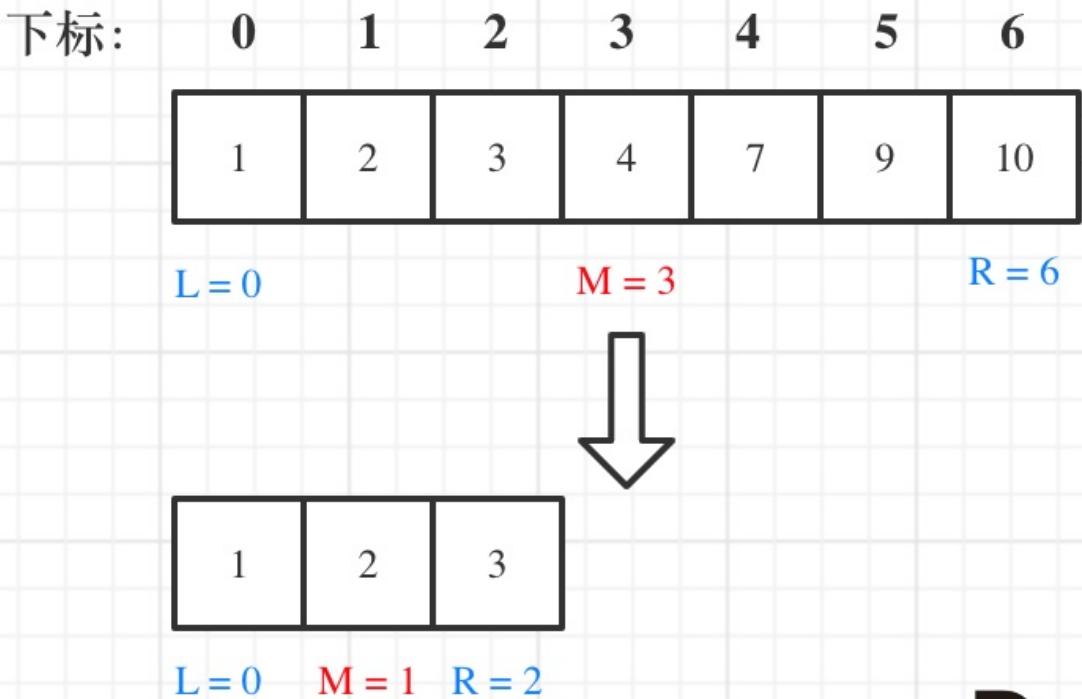
示例 2:

```
输入: nums = [-1,0,3,5,9,12], target = 2
输出: -1
解释: 2 不存在 nums 中因此返回 -1
```

提示:

- 你可以假设 `nums` 中的所有元素是不重复的。
- n 将在 [1, 10000]之间。
- `nums` 的每个元素都将在 [-9999, 9999]之间。

假设存在数组: 1,2,3,4,7,9,10中查找元素2.



```

public static int twoLinkSearch(int[] nums, int target) {
    //当 target 小于nums[0] 或者nums[nums.length - 1]时返回-1
    if (target < nums[0] || target > nums[nums.length - 1]) {
        return -1;
    }
    int left = 0;
    int right = nums.length - 1;
    while (left <= right) {
        int mid = left + ((right - left) / 2); // 防止溢出 等同于(left + right)/2
        if (nums[mid] == target)
            return mid;
        else if (nums[mid] < target) // 中间数小于目标值，则目标值在右半区
            left=mid+1;
        else if (nums[mid]>target) // 中间数大于目标值，则目标值在左半区
            right=mid-1;
    }
    return -1;
}

```

移除元素

要知道数组的元素在内存地址中是连续的，不能单独删除数组中的某个元素，只能覆盖。

例题：给你一个数组 `nums` 和一个值 `val`，你需要 原地 移除所有数值等于 `val` 的元素，并返回移除后数组的新长度。

提示：不要使用额外的数组空间，你必须仅使用 O(1) 额外空间并原地修改输入数组。

示例 1：

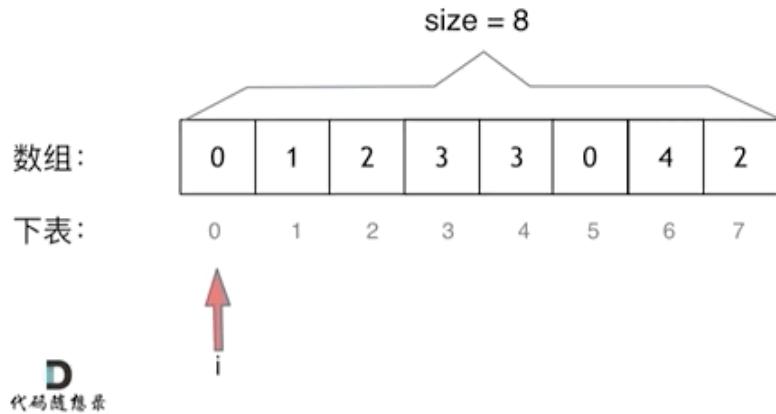
给定 `nums = [3,2,2,3]`, `val = 3`, 函数应该返回新的长度 2, 并且 `nums` 中的前两个元素均为 2。你不需要考虑数组中超出新长度后面的元素。

示例 2：

给定 $\text{nums} = [0,1,2,2,3,0,4,2]$, $\text{val} = 2$, 函数应该返回新的长度 5, 并且 nums 中的前五个元素为 0, 1, 3, 0, 4。

暴力循环解法:

$\text{nums} = [0,1,2,3,3,0,4,2]$, 删除元素2,

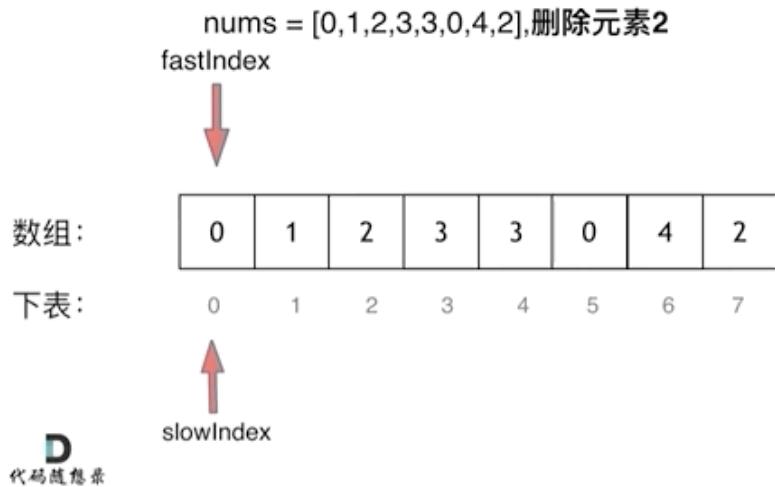


使用两层for循环，一个for循环遍历数组元素，第二个for循环更新数组。

```
// 时间复杂度: O(n^2)
// 空间复杂度: O(1)
public static int foreachDeletevar(int[] nums, int target){
    int length = nums.length;
    for (int i=0;i<length;i++){
        if (nums[i]==target){ // 发现需要移除的元素，就将后面数组集体向前移动一位，进行覆盖操作
            for (int j=i+1;j<length;j++){
                nums[j-1]=nums[j];
            }
            i--; // 因为下表i以后的数值都向前移动了一位，所以i也向前移动一位
            length--; // 此时数组的大小-1
        }
    }
    return length;
}
```

双指针法:

双指针法（快慢指针法）：通过一个快指针和慢指针在一个for循环下完成两个for循环的工作。



双指针法（快慢指针法）在数组和链表的操作中是非常常见的，很多考察数组、链表、字符串等操作的面试题，都使用双指针法。

```
// 时间复杂度: O(n)
// 空间复杂度: O(1)
public static int fastSlowIndexDeleteVar(int[] nums, int target) {
    int fast = 0;
    int slow;
    for (slow = 0; fast < nums.length; fast++) {
        if (nums[fast] != target) {
            nums[slow] = nums[fast];
            slow++;
        }
    }
    return slow;
}
```

滑动窗口法

所谓滑动窗口，就是不断的调节子序列的起始位置和终止位置，从而得出我们想要的结果。通常使用在求和的数组上。

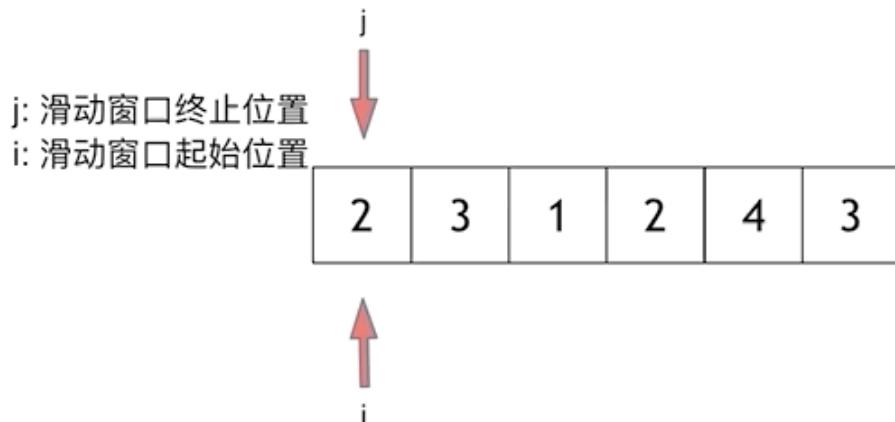
滑动窗口也可以理解为双指针法的一种！只不过这种解法更像是一个窗口的移动，所以叫做滑动窗口更适合一些。

例题：给定一个含有 n 个正整数的数组和一个正整数 s，找出该数组中满足其和 $\geq s$ 的长度最小的连续子数组，并返回其长度。如果不存在符合条件的子数组，返回 0。

示例：

输入：s = 7, nums = [2,3,1,2,4,3] 输出：2 解释：子数组 [4,3] 是该条件下的长度最小的子数组。

使用滑动窗口法：



本题中实现滑动窗口，主要确定如下三点：

- 窗口内是什么？
- 如何移动窗口的起始位置？
- 如何移动窗口的结束位置？

窗口就是满足其和 $\geq s$ 的长度最小的连续子数组。

窗口的起始位置如何移动：如果当前窗口的值大于s了，窗口就要向前移动了（也就是该缩小了）。

窗口的结束位置如何移动：窗口的结束位置就是遍历数组的指针，窗口的起始位置设置为数组的起始位置就可以了。

```
public static int minSubLen(int[] nums, int target) {
    int start = 0;
    int sum = 0;
    int result = Integer.MAX_VALUE;
    for (int end = 0; end < nums.length; end++) {
        sum += nums[end];
        while (sum >= target){ //判断结果是否大于或等于目标值
            result = Math.min(result, end - start + 1); //判断结果的数组长度,如果更小
            //则替代原结果
            sum -= nums[start++]; //减掉开始指针的值，并将开始指针向前移动
        }
    }
    return result == Integer.MAX_VALUE ? 0 : result;
}
```

数列排序算法

进行数列排序算法：

插入排序

6 5 3 1 8 7 2 4

每次将一个数字插入一个有序的数组里，成为一个长度更长的有序数组，有限次操作以后，数组整体有序。

```
/**  
 * 时间复杂度: O(N^2)  
 * 空间复杂度: O(1)  
 */  
public int[] sortArray(int[] nums) {  
    int len = nums.length;  
    // 循环不变量：将 nums[i] 插入到区间 [0, i) 使之成为有序数组  
    for (int i = 1; i < len; i++) {  
        int temp = nums[i]; // 提前将其当前指针的值存储  
        int j = i; // 用作遍历当前指针前的元素的指针  
        // 判断前一个数是否大于当前数，注意边界 j > 0  
        while (j > 0 && nums[j - 1] > temp) {  
            nums[j] = nums[j - 1];  
            j--;  
        }  
        nums[j] = temp;  
    }  
    return nums;  
}
```

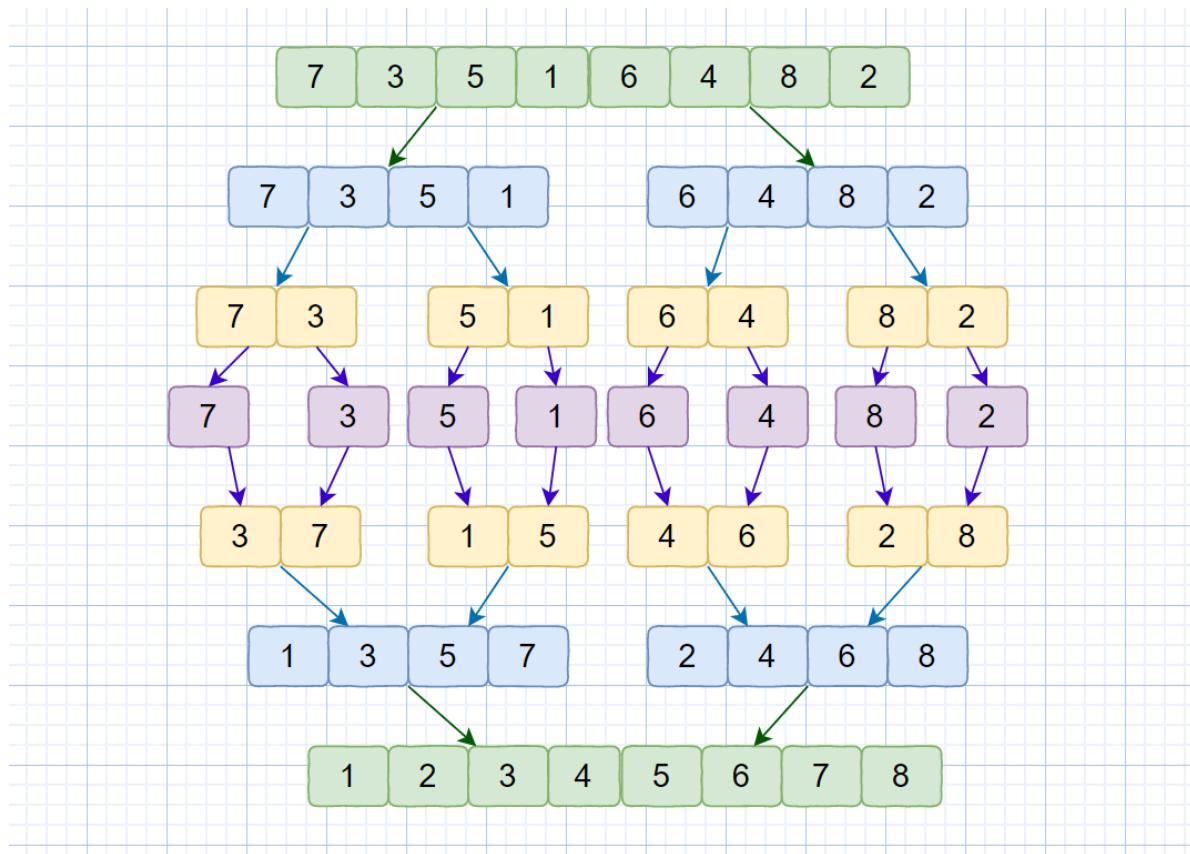
在小区间内执行排序任务的时候，可以转向使用「插入排序」。

归并排序

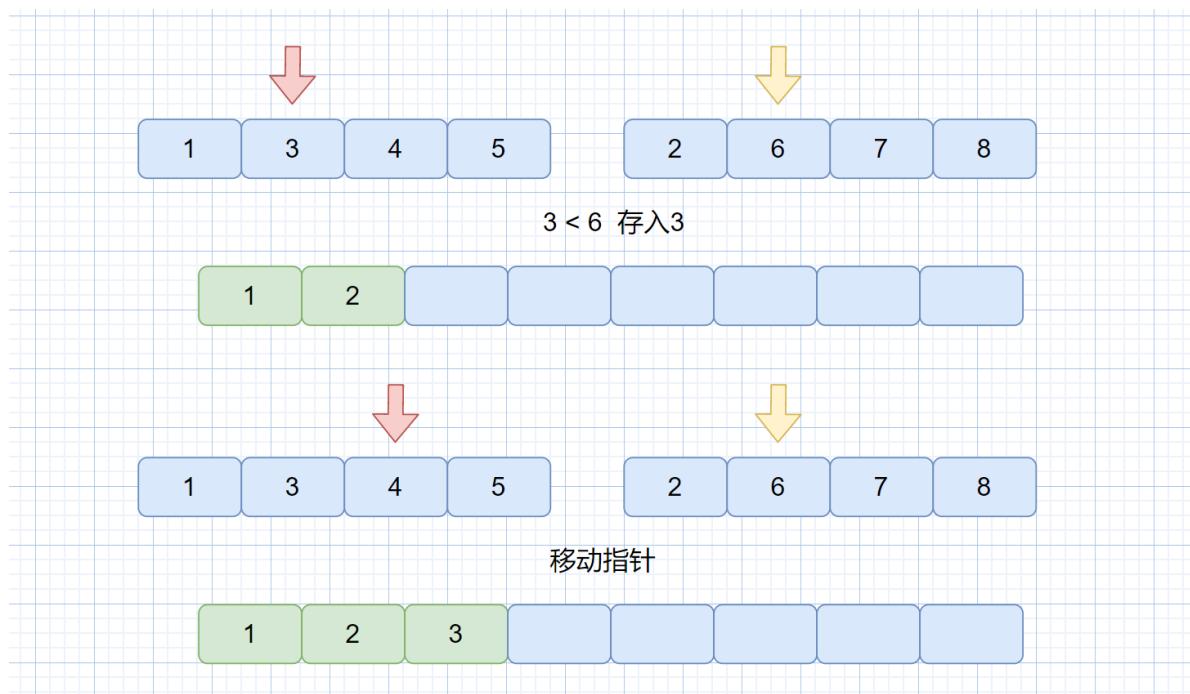
归并这个词语的含义就是合并，并入的意思。

归并排序使用的就是分治思想。顾名思义就是分而治之，将一个大问题分解成若干个小的子问题来解决。

将原数组分区域，分别排序，借助额外空间，合并分区域的有序数组，得到更长的有序数组。



而其中的归并主要实现思想为:



通过双指针从左自右比较两个指针指向的值,将较小的一方存入大集合中,存入之后较小一方的指针向前移动,并继续比较,直到某一小集合的元素全部都存到大集合中。

1 3 4 5

2 6 7 8

需要将6,7,8放入大集合中

1 2 3 4 5

6 7 8

1 3 4 5

2 6 7 8

1 2 3 4 5 6 7 8

当某一小集合元素全部放入大集合中(指针到尾部无法继续后), 则需将另一小集合中剩余的所有元素存到大集合中.

```
/*
 * 对数组 nums 的子区间 [left, right] 进行归并排序
 * @param nums
 * @param left
 * @param right
 * @param temp
 */
private void mergeSort(int[] nums, int left, int right, int[] temp) {
    // 对小区间的数组进行插入排序
    if (right - left <= INSERTION_SORT_THRESHOLD) {
        insertionSort(nums, left, right);
        return;
    }
    int mid = left + (right - left) / 2;
    // 分成两部分进行继续归并排序
    mergeSort(nums, left, mid, temp);
    mergeSort(nums, mid + 1, right, temp);
    // 如果第一部分的尾部 小于或等于 第二部分的头部 则无需合并排序
    if (nums[mid] <= nums[mid + 1]) {
        return;
    }
    // 合并小区间数组排序
    mergeOfTwoSortedArray(nums, left, mid, right, temp);
}

/**
 * 合并两个有序数组: 先把值复制到临时数组, 再合并回去
 * @param nums
 * @param left
 * @param mid
 * @param right
 * @param temp
 */
```

```

    public void mergeOfTwoSortedArray(int[] nums, int left, int mid, int right,
    int[] temp) {
        System.arraycopy(nums, left, temp, left, right + 1 - left); // 备份原数组
        int i = left; //指针i从已排序第一部分头部开始
        int j = mid + 1; //指针j从已排序第二部分头部开始
        for (int k = left; k <= right; k++) {
            if (i == mid + 1) { //指针i到达已排序第一部分尾部
                nums[k] = temp[j];
                j++;
            } else if (j == right + 1) { //指针j到达已排序第二部分尾部
                nums[k] = temp[i];
                i++;
            } else if (temp[i] <= temp[j]) {
                // 注意写成 < 就丢失了稳定性(相同元素原来靠前的排序以后依然靠前)
                nums[k] = temp[i];
                i++;
            } else {
                // temp[i] > temp[j]
                nums[k] = temp[j];
                j++;
            }
        }
    }
}

```

算法名称	最好时间复杂度	最坏时间复杂度	平均时间复杂度	空间复杂度	是否稳定
归并排序	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	稳定

快速排序

快速排序也是一种分治的排序算法.它将数组通过切分(partition)分成两个子数组,将两部分独立地排序.快排和归并排序是互补的.

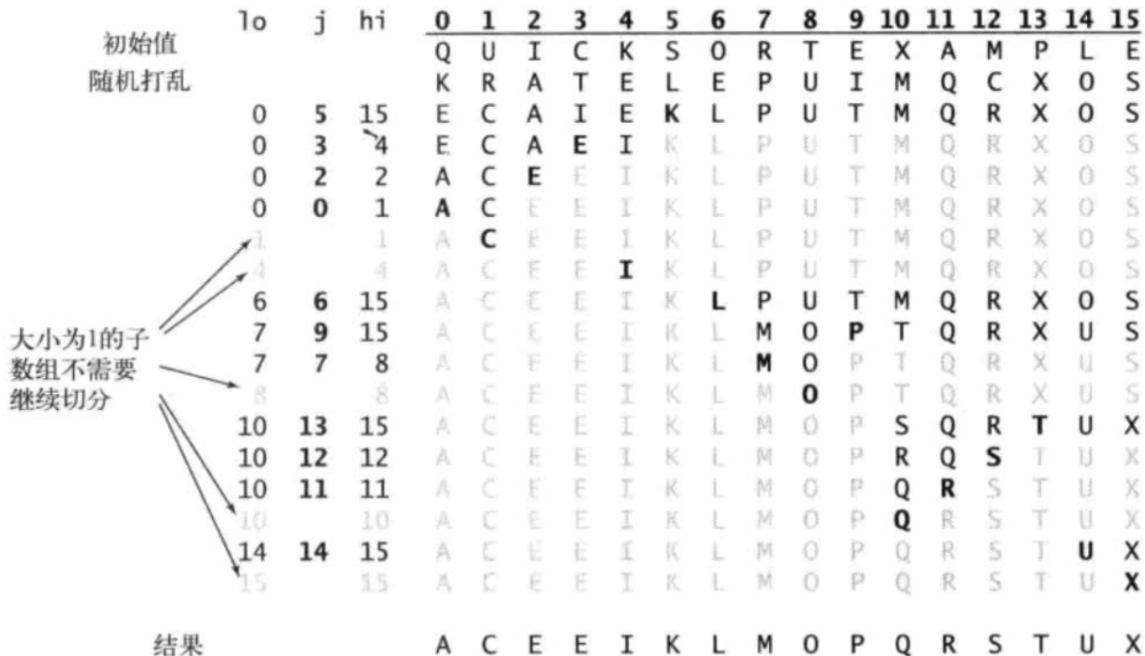
快速排序基本思想:

- 1.先从数组中找一个基准数
- 2.让其他比它大的元素移动到数列一边, 比他小的元素移动到数列另一边, 从而把数组拆解成两个部分。
- 3.再对左右区间重复第二步, 直到各区间只有一个数。

在快速排序中,首先随机打乱数组排序.然后取出切分元素,它把数组大于小于它的进行划分.



然后分别将左右部分循环进行快速排序,



一边循环到只有大小一为止,排序到最后直接合并即可(不需要像归并排序最后还需要合并排序).

- 版本 1: 基本版本: 把等于切分元素的所有元素分到了数组的同一侧, 但可能会造成递归树倾斜;

```
/*
 * 快速排列
 * @param nums
 * @param left
 * @param right
 */
public void quickSort(int[] nums, int left, int right) {
    // 判断是否 长度是否为0 ,为0则为排序完毕的元素
    if (right - left + 1 <= 0){
        return;
    }
    //获取切点元素位置
    int pIndex = partition(nums, left, right);
    quickSort(nums, left, pIndex - 1); //左侧
    quickSort(nums, pIndex + 1, right); //右侧
}

/**
 * 切分数组
 *
```

```

* @param nums
* @param left
* @param right
*/
private int partition(int[] nums, int left, int right) {
    //获取随机下标数(切分元素)
    int randomIndex = new Random().nextInt(right - left + 1) + left;
    swap(nums, left, randomIndex); //将头部与切分元素进行交换

    // 切分元素值
    int pivot = nums[left];
    // 切分元素下标位置
    int less = left;
    for (int i = left + 1; i <= right; i++) {
        if (nums[i] < pivot) { //指针的值小于切分值
            less++;
            swap(nums, i, less); //切分值与其互换位置
        }
    }
    swap(nums, left, less);
    return less;
}

/**
 * 交换两个值位置
 */
private void swap(int[] nums, int index1, int index2) {
    int temp = nums[index1];
    nums[index1] = nums[index2];
    nums[index2] = temp;
}

```

版本2:双指针法 - 把等于切分元素的所有元素等概率地分到了数组的两侧，避免了递归树倾斜，递归树相对平衡；

```

/**
 * 快速排列
 *
 * @param nums
 * @param left
 * @param right
 */
public void quickSort(int[] nums, int left, int right) {
    if (left < right) {
        int pIndex = partition(nums, left, right);
        quickSort(nums, left, pIndex - 1);
        quickSort(nums, pIndex + 1, right);
    }
}

public int partition(int[] nums, int left, int right) {
    int randomIndex = left + new Random().nextInt(right - left + 1);
    swap(nums, randomIndex, left);

    int pivot = nums[left];
    // 两个指针:一个从前到后,一个从后向前
    int less = left + 1;

```

```

int greater = right;
while (true){
    while (less<=right&&nums[less]<pivot){
        less++;
    }
    while (greater>left&& nums[greater]>pivot){
        greater--;
    }
    // 当小指针大于或等于大指针退出循环
    if (less>=greater){
        break;
    }
    // 当less指针大于等于 且greater指针小于等于时 ,同时双指针移动
    swap(nums,less,greater);
    less++;
    greater--;
}
swap(nums,left,greater);
return greater;
}

public void swap(int[] nums, int index1, int index2) {
    int temp = nums[index1];
    nums[index1] = nums[index2];
    nums[index2] = temp;
}

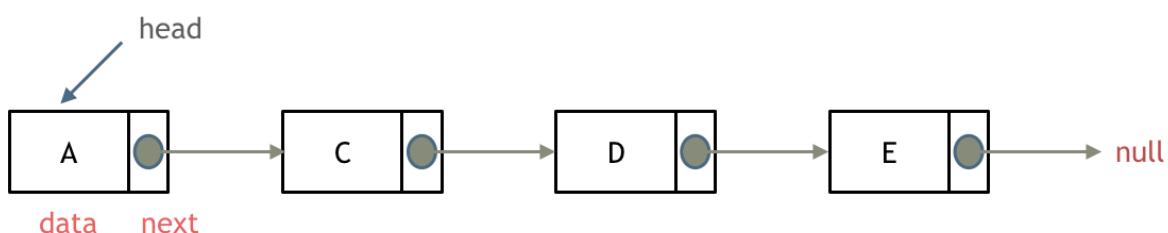
```

链表

链表是一种通过指针串联在一起的线性结构，每一个节点是由两部分组成，一个是数据域一个是指针域（存放指向下一个节点的指针）。

最后一个节点的指针域指向null（空指针的意思）。

链接的入口点称为列表的头结点也就是head。

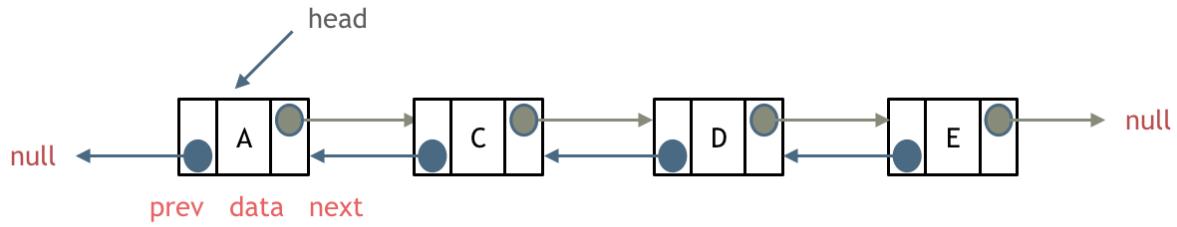


####

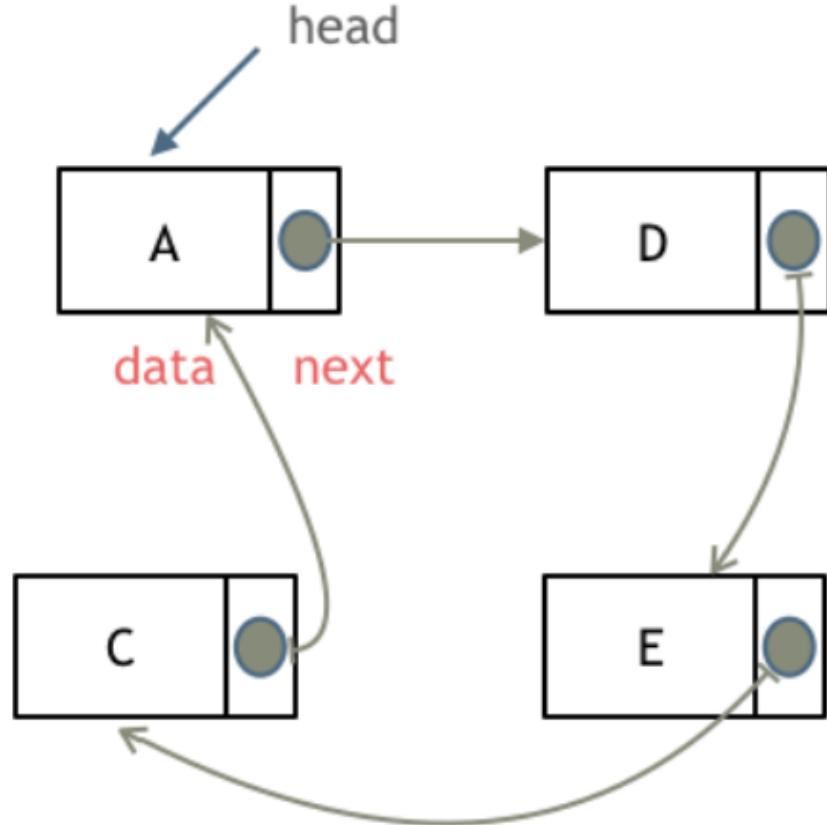
链表的类型

1. 单链表 同上图,单链表中的节点只能指向节点的下一个节点。

2. 双链表:每一个节点有两个指针域，一个指向下一个节点，一个指向上一个节点。所以双链表既可以向前查询也可以向后查询。



3.循环链表:顾名思义，就是链表首尾相连。循环链表可以用来解决约瑟夫环问题。



链表存储方式

数组是在内存中是连续分布的，但是链表在内存中可不是连续分布的。

链表是通过指针域的指针链接在内存中各个节点。

所以链表中的节点在内存中不是连续分布的，而是散乱分布在内存中的某地址上，分配机制取决于操作系统的内存管理。

删除节点

只要将节点的next指针 指向删除的节点后的下一个节点就可以了。

Java、Python，就有自己的内存回收机制，就不用自己手动内存释放了。但C++里最好是再手动释放这个删除的节点，释放这块内存。

删除最后一个节点时,需要从头节点查找到第四个节点通过next指针进行删除操作，查找的时间复杂度是O(n)。

性能分析

	插入/删除（时间复杂度）	查询（时间复杂度）	适用场景
数组	O(n)	O(1)	数据量固定，频繁查询，较少增删
链表	O(1)	O(n)	数据量不固定，频繁增删，较少查询

数组在定义的时候，长度就是固定的，如果想改动数组的长度，就需要重新定义一个新的数组。

链表的长度可以是不固定的，并且可以动态增删，适合数据量不固定，频繁增删，较少查询的场景。

移除链表元素问题

例题：删除链表中等于给定值 val 的所有节点。

示例 1：

输入：head = [1,2,6,3,4,5,6], val = 6

输出：[1,2,3,4,5]

示例 2：

输入：head = [], val = 1

输出：[]

示例 3：

输入：head = [7,7,7,7], val = 7

输出：[]

移除操作，就是让节点next指针直接指向下一个节点就可以了。

但如果删除的是头结点的话：

- 直接使用原来的链表来进行删除操作。

移除头结点和移除其他节点的操作是不一样的，因为链表的其他节点都是通过前一个节点来移除当前节点，而头结点没有前一个节点。

所以头结点移除，只要将头结点向后移动一位就可以，这样就从链表中移除了一个头结点。

```
public void removeVal(String val) {  
    if (head != null && val.equals(head.getData())) {  
        head = head.getNext();  
    }  
    if (head == null) {  
        return;  
    }  
  
    Node pre = head;
```

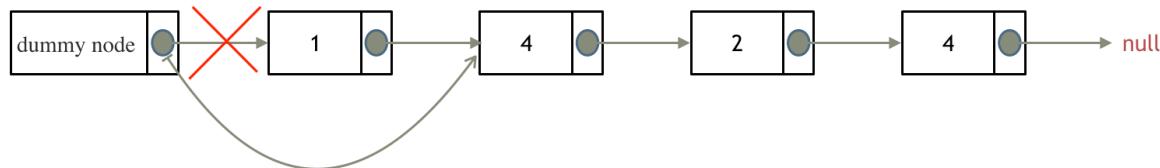
```

        Node cur = head.getNext();
        while (cur != null) {
            if (val.equals(cur.getData())) {
                pre.setNext(cur.getNext());
                size--;
            } else {
                pre = cur;
            }
            cur = cur.getNext();
        }
    }
}

```

- 设置一个虚拟头结点在进行删除操作。

链表: 1->4->2->4 移除元素1



```

public ListNode removeElements(ListNode head, int val) {
    if (head == null) {
        return head;
    }
    // 因为删除可能涉及到头节点，所以设置dummy节点，统一操作
    ListNode dummy = new ListNode(-1, head);
    ListNode pre = dummy;
    ListNode cur = head;
    while (cur != null) {
        if (cur.val == val) {
            pre.next = cur.next;
        } else {
            pre = cur;
        }
        cur = cur.next;
    }
    return dummy.next;
}

```

设计链表

例题:

在链表类中实现这些功能:

- get(index): 获取链表中第 index 个节点的值。如果索引无效，则返回-1。

- `addAtHead(val)`: 在链表的第一个元素之前添加一个值为 `val` 的节点。插入后，新节点将成为链表的第一个节点。
- `addAtTail(val)`: 将值为 `val` 的节点追加到链表的最后一个元素。
- `addAtIndex(index,val)`: 在链表中的第 `index` 个节点之前添加值为 `val` 的节点。如果 `index` 等于链表的长度，则该节点将附加到链表的末尾。如果 `index` 大于链表长度，则不会插入节点。如果 `index` 小于0，则在头部插入节点。
- `deleteAtIndex(index)`: 如果索引 `index` 有效，则删除链表中的第 `index` 个节点。

示例：

```
MyLinkedList linkedList = new MyLinkedList();
linkedList.addAtHead(1);
linkedList.addAtTail(3);
linkedList.addAtIndex(1,2);    //链表变为1-> 2-> 3
linkedList.get(1);           //返回2
linkedList.deleteAtIndex(1);  //现在链表是1-> 3
linkedList.get(1);           //返回3
```

分析:这道题目设计链表的五个接口：

- 获取链表第`index`个节点的数值
- 在链表的最前面插入一个节点
- 在链表的最后面插入一个节点
- 在链表第`index`个节点前面插入一个节点
- 删除链表的第`index`个节点

链表操作的两种方式：

1. 直接使用原来的链表来进行操作。
2. 设置一个虚拟头结点在进行操作。

为了方便,采用的设置一个虚拟头结点更好.

实现

单链表:

```
/**
 * @description: 单链表
 * @author: zhaotianyi
 * @time: 2021/6/16 15:47
 */
public class Node {
    private String data;
    private Node next;

    public Node(String data) {
```

```
    this.data = data;
}

public Node getNext() {
    return next;
}

public void setNext(Node next) {
    this.next = next;
}

public String getData() {
    return data;
}

}

/***
 * @description: TODO
 * @author: zhaotianyi
 * @time: 2021/6/16 16:10
 */
public class LinkList {
    int size; //链表长度
    Node head; //虚拟头结点

    /**
     * 初始化链表
     */
    public LinkList(){
        size=0;
        head = new Node("null data");
    }

    /**
     * 获取指定位置节点数值
     * @param index
     * @return
     */
    public String get(int index){
        // Index非法返回
        if (index < 0 || index >= size){
            return "Failed Index";
        }
        Node currentNode =head; //查询从头部节点开始
        //包含一个虚拟头节点，所以查找第 index+1 个节点
        for (int i = 0; i <= index; i++) {
            currentNode = currentNode.getNext();
        }
        return currentNode.getData();
    }

    /**
     * 向指定位置添加相关数据节点
     * @param index
     * @param val
     */
    public void addAtIndex(int index, String val){
        if (index > size){


```

```

        return;
    }
    if (index < 0){
        index = 0;
    }
    size++; //添加后链表数量自增
    //找到要添加节点的前一个节点
    Node pred = head; //查询从头部节点开始
    for (int i = 0; i < index; i++){
        pred = pred.getNext();
    }
    Node addNode = new Node(val);
    addNode.setNext(pred.getNext()); //新增节点的指针指向前节点的指针
    pred.setNext(addNode); //前节点指针改为新增指针 实现插入节点
}

/**
 * 向链表头部插入数据节点
 * @param val
 */
public void addAtHead(String val){
    addAtIndex(0, val);
}

/**
 * 向链表尾部插入数据节点
 * @param val
 */
public void addAtTail(String val){
    addAtIndex(size, val);
}

/**
 * 删除指定位置节点
 * @param index
 */
public void deleteAtIndex(int index){
    if (index >= size || index < 0){
        return;
    }
    size--; //删除后链表数量自减
    Node pred = head;
    for (int i = 0; i < index; i++) {
        pred = pred.getNext();
    }
    pred.setNext(pred.getNext().getNext());
}
}

```

双链表:

```

/**
 * 多链表
 * @description: TODO
 * @author: zhaotianyi
 * @time: 2021/6/16 16:43
 */

```

```

public class MyLinkedList {
    class ListNode {
        int val;
        ListNode next,prev;
        ListNode(int x) {val = x;}
    }

    int size; //链表长度
    ListNode head,tail; //哨兵节点 [头部节点,尾部节点]

    /** Initialize your data structure here. */
    public MyLinkedList() {
        size = 0;
        head = new ListNode(0);
        tail = new ListNode(0);
        head.next = tail;
        tail.prev = head;
    }

    /** Get the value of the index-th node in the linked list. If the index is
     * invalid, return -1. */
    public int get(int index) {
        if(index < 0 || index >= size){return -1;}
        ListNode cur = head;

        // 通过判断 index < (size - 1) / 2 来决定是从头结点还是尾节点遍历，提高效率
        if(index < (size - 1) / 2){
            for(int i = 0; i <= index; i++){
                cur = cur.next;
            }
        }else{
            cur = tail;
            for(int i = 0; i <= size - index - 1; i++){
                cur = cur.prev;
            }
        }
        return cur.val;
    }

    /** Add a node of value val before the first element of the linked list.
     * After the insertion, the new node will be the first node of the linked list. */
    public void addAtHead(int val) {
        ListNode cur = head;
        ListNode newNode = new ListNode(val);
        newNode.next = cur.next;
        cur.next.prev = newNode;
        cur.next = newNode;
        newNode.prev = cur;
        size++;
    }

    /** Append a node of value val to the last element of the linked list. */
    public void addAtTail(int val) {
        ListNode cur = tail;
        ListNode newNode = new ListNode(val);
        newNode.next = tail;
        newNode.prev = cur.prev;
        cur.prev.next = newNode;
    }
}

```

```

        cur.prev = newNode;
        size++;
    }

    /**
     * Add a node of value val before the index-th node in the linked list. If
     * index equals to the length of linked list, the node will be appended to the end
     * of linked list. If index is greater than the length, the node will not be
     * inserted. */
    public void addAtIndex(int index, int val) {
        if(index > size){return;}
        if(index < 0){index = 0;}
        ListNode cur = head;
        for(int i = 0; i < index; i++){
            cur = cur.next;
        }
        ListNode newNode = new ListNode(val);
        newNode.next = cur.next;
        cur.next.prev = newNode;
        newNode.prev = cur;
        cur.next = newNode;
        size++;
    }

    /**
     * Delete the index-th node in the linked list, if the index is valid. */
    public void deleteAtIndex(int index) {
        if(index >= size || index < 0){return;}
        ListNode cur = head;
        for(int i = 0; i < index; i++){
            cur = cur.next;
        }
        cur.next.next.prev = cur;
        cur.next = cur.next.next;
        size--;
    }
}

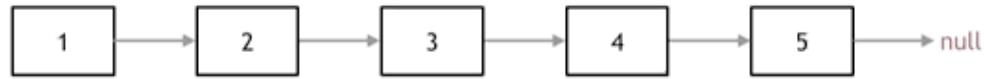
```

反转链表

题意：反转一个单链表。

如果再定义一个新的链表，实现链表元素的反转，其实这是对内存空间的浪费。

其实只需要改变链表的next指针的指向，直接将链表反转，而不用重新定义一个新的链表。



使用双指针方法.

首先定义一个cur指针，指向头结点，再定义一个pre指针，初始化为null。

反转前将cur中节点原指向节点进行备份,然后进行替换操作,最后将备份节点作为cur进行指向下一个节点,循环操作. prev则为新的反转后的链表.

```

/**
 * 反转链表
 */
public void reverseList() {
    Node prev = null;
    Node cur = head;
    Node temp = null;
    while (cur != null) {
        temp = cur.getNext(); //临时保存下一个节点
        cur.setNext(prev);
        prev = cur;
        cur = temp;
    }
    head = prev;
}

```

哈希表

哈希表是根据关键码的值而直接进行访问的数据结构。

直白来讲其实数组就是一张哈希表。哈希表中关键码就是数组的索引下表，然后通过下表直接访问数组中的元素。

使用环境:一般哈希表都是用来快速判断一个元素是否出现集合里。

哈希表的检索时间复杂度为O(1).

常见的三种哈希结构

- 数组
- set (集合)
- map(映射)

其中JAVA中

HashMap:底层采用哈希表,无序,不可重复

HashSet:底层采用哈希表,无序,不可重复 存储时, 会采用链式结构进行存储。

TreeSet:底层采用红黑树,有序,不可重复 存储时, 会采用链式结构进行存储。

当我们遇到了要快速判断一个元素是否出现集合里的时候, 就要考虑哈希法。

但是哈希法也是**牺牲了空间换取了时间**, 因为我们要使用额外的数组, set或者是map来存放数据, 才能实现快速的查找。

两个数组的交集(SET)

利用set的不可重复存储的特性,适合计算交集;

遍历数组,将其值添加至set中,然后在下一个数组中遍历选择相同的属性

```
/**
 * 返回两数值交集
 *
 * @param nums1
 * @param nums2
 * @return
 */
public static int[] intersection(int[] nums1, int[] nums2) {
    if (nums1 == null || nums2 == null || nums1.length == 0 || nums2.length == 0) {
        return new int[0];
    }
    Set<Integer> set1 = new TreeSet<>();
    Set<Integer> set2 = new TreeSet<>();
    for (int n : nums1) {
        set1.add(n);
    }
    for (int n : nums2) {
        if (set1.contains(n)) {
            set2.add(n);
        }
    }
    int[] ints = new int[set2.size()];
    int index = 0;
    for (int n : set2) {
        ints[index++] = n;
    }
    return ints;
}
```

两数之和的下标(MAP)

给定一个整数数组 `nums` 和一个目标值 `target`, 请你在该数组中找出和为目标值的那 两个 整数, 并返回他们的数组下标。你可以假设每种输入只会对应一个答案。但是, 数组中同一个元素不能使用两遍。

示例:

给定 `nums = [2, 7, 11, 15]`, `target = 9`

因为 `nums[0] + nums[1] = 2 + 7 = 9`

所以返回 `[0, 1]`

暴力的解法是两层for循环查找, 时间复杂度是 $O(n^2)$ 。

map是一种key value的存储结构, 可以用key保存数值, 用value在保存数值所在的下标。

`nums = [2, 7, 11, 15], target = 9`

下表: 0 1 2 3

寻找`target - nums[i]`是否在map中

map<数值, 下表>

```
public static int[] twoSum(int[] nums, int val) {
    int[] ints = new int[2];
    if (nums == null || nums.length == 0) {
        return ints;
    }
    HashMap<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        int temp = val - nums[i];
        if (map.containsKey(temp)) {
            ints[0] = i;
            ints[1] = map.get(temp); // 获取满足值的下标位置
            break;
        }
        map.put(nums[i], i); // 将key设为值,value设为下标位置
    }
    return ints;
}
```

四数组相加

给定四个包含整数的数组列表 A , B , C , D ,计算有多少个元组 (i, j, k, l) , 使得 $A[i] + B[j] + C[k] + D[l] = 0$ 。

例如:

输入: A = [1, 2] B = [-2,-1] C = [-1, 2] D = [0, 2] 输出: 2 解释: 两个元组如下:

1. (0, 0, 0, 1) -> $A[0] + B[0] + C[0] + D[1] = 1 + (-2) + (-1) + 2 = 0$
2. (1, 1, 0, 0) -> $A[1] + B[1] + C[0] + D[0] = 2 + (-1) + (-1) + 0 = 0$

```
public static int fourSumConunt(int[] nums1, int[] nums2, int[] nums3, int[]  
nums4) {  
    HashMap<Integer, Integer> map = new HashMap<>();  
    int temp;  
    int res = 0;  
    //统计两个数组中的元素之和，同时统计出现的次数，放入map  
    for (int i : nums1) {  
        for (int j : nums2) {  
            temp = i + j; // 相加的值进行存储  
            if (map.containsKey(temp)) {  
                map.put(temp, map.get(temp) + 1); //相同的值的话，计数加一  
            } else {  
                map.put(temp, 1); //第一次出现的值记录  
            }  
        }  
    }  
    //统计剩余的两个元素的和，在map中找是否存在相加为0的情况，同时记录次数  
    for (int i : nums3) {  
        for (int j : nums4) {  
            temp = i + j;  
            if (map.containsKey(-temp)){ //判断是否存在有其相反的值，与其相加为0  
                res += map.get(-temp); //出现计数累加  
            }  
        }  
    }  
    return res;  
}
```

获取验证码 easy-captcha

Maven依赖

```
<!-- 验证码easy-captcha -->
<dependency>
    <groupId>com.github.whvcse</groupId>
    <artifactId>easy-captcha</artifactId>
    <version>1.6.2</version>
</dependency>
```

使用方法

```
// 算数类型验证码
ArithmeticCaptcha captcha = new ArithmeticCaptcha(130, 48);
// 中文类型
ChineseCaptcha captcha = new ChineseCaptcha(130, 48);
```

```
// 几位数运算，默认为两位
captcha.setLen(2);
// 获取运算的公式: 3+2=?
captcha.getArithmetixcString();
// 获取运算的结果: 5
String value = captcha.text();
```

```
String key = UuidUtil.createUuid();
// 存入redis并设置过期时间为5分钟
RedisUtil.set(key, value, 600);
HashMap<String, String> captchaMap = new HashMap<String, String>(2);
captchaMap.put("captchaKey", key);
captchaMap.put("image", captcha.toBase64());
// 将key和验证码base64返回给前端
return Result.success(captchaMap);
```

AJAX 刷新验证码

```
function refreshcode(obj)
{
    obj.src="/captcha?id="+Math.random();
};
```

```

```

Git版本管理

Git 是一个开源的分布式版本控制系统，用于敏捷高效地处理任何或小或大的项目，适合单人、多人合作开发。

Git 是 Linus Torvalds 为了帮助管理 Linux 内核开发而开发的一个开放源码的版本控制软件。

无论是国外的 GitHub、GitLab，还是国内的 Gitee 都是 Git 项目代码管理网站。

Git 与 SVN 区别

Git 和 SVN 都是项目版本控制软件，但一些区别：

Git 是分布式的，SVN 不是：这是 Git 和其它非分布式的版本控制系统，例如 SVN，CVS 等，最核心的区别。

Git 把内容按元数据方式存储，而 SVN 是按文件：所有的资源控制系统都是把文件的元信息隐藏在一个类似 .svn、.cvs 等的文件夹里。

Git 分支和 SVN 的分支不同：分支在 SVN 中一点都不特别，其实它就是版本库中的另外一个目录。

Git 没有一个全局的版本号，而 SVN 有：目前为止这是跟 SVN 相比 Git 缺少的最大的一个特征。

Git 的内容完整性要优于 SVN：Git 的内容存储使用的是 SHA-1 哈希算法。这能确保代码内容的完整性，确保在遇到磁盘故障和网络问题时降低对版本库的破坏。

在目前国内大部分公司优先使用 Git 为主，一部分对内公司采用 SVN。

Git 安装

Git 各平台安装包下载地址为：<http://git-scm.com/downloads>

Linux 平台上安装

Git 的工作需要调用 curl, zlib, openssl, expat, libiconv 等库的代码，所以需要先安装这些依赖工具。

在有 yum 的系统上（比如 Fedora）或者有 apt-get 的系统上（比如 Debian 体系），可以用下面的命令安装：

各 Linux 系统可以使用其安装包管理工具（apt-get、yum 等）进行安装：

```
$ yum install curl-devel expat-devel gettext-devel \
openssl-devel zlib-devel

$ yum -y install git-core

$ git --version
git version 1.7.1
```

Git 配置

Git 提供了一个叫做 git config 的工具，专门用来配置或读取相应的工作环境变量。

在 Windows 系统上，Git 会找寻用户主目录下的 .gitconfig 文件。主目录即 \$HOME 变量指定的目录，一般都是 C:\Documents and Settings\\$USER。

Git 还会尝试找寻 /etc/gitconfig 文件，只不过看当初 Git 装在什么目录，就以此作为根目录来定位。

配置个人的用户名称和电子邮件地址：

```
$ git config --global user.name "runoob"
$ git config --global user.email test@runoob.com
```

如果用了 **--global** 选项，那么更改的配置文件就是位于你用户主目录下的那个，以后你所有的项目都会默认使用这里配置的用户信息。

我们先来理解下 Git 工作区、暂存区和版本库概念：

- **工作区**：就是你在电脑里能看到的目录。
- **暂存区**：英文叫 stage 或 index。一般存放在 .git 目录下的 index 文件 (.git/index) 中，所以我们把暂存区有时也叫作索引 (index) 。
- **版本库**：工作区有一个隐藏目录 .git，这个不算工作区，而是 Git 的版本库。

Git使用入门

使用Git前，需要先建立一个仓库(repository)。您可以使用一个已经存在的目录作为Git仓库或创建一个空目录,或者去克隆其他项目

1.使用命令在其项目文件夹下来创建Git项目,或者从其他项目下克隆下来

```
git init      #新建项目到文件夹  
git clone [项目地址]  #克隆其他项目到当前文件夹
```

2.设置该项目的远程服务器位置(可以在Github,Gitee上进行新建项目以便获取地址),以便之后上传文件(克隆项目跳过此步骤)

```
git remote add origin [项目地址]
```

2.使用命令来添加需要提交的文件

```
git add [文件名][文件名]...  #添加指定文件到暂存区  
git add [dir]          #添加指定目录到暂存区，包括子目录  
git add .            #添加当前目录下的所有文件到暂存区
```

3.现在我们已经添加了这些文件，我们希望它们能够真正被保存在Git HEAD中。

```
git commit -m "提交信息"  
git commit -a      # -a 参数设置后下次修改文件后不需要执行 git add 命令，直接来提交
```

4.现在我们修改之后可以进行推送到服务器中。

```
git push -u origin master  #将其HEAD区推出至远程服务器中的Master分支中  
git push #将其推出至当前所在分支中
```

5.当远程服务器中代码更新后,你可以从远程服务器中获取最新代码并合并本地的版本。

```
git pull #默认将当前分支更新合并  
git pull origin master:feature_A          #将远程服务器中的origin 的 master 分支拉取  
过来，与本地的 feature_A 分支合并。
```

6.获取历史提交版本信息

```
git log
```

7.撤回版本

```
git reset [版本值]      #返回至指定版本(版本值只需要一部分头部即可)  
git reset HEAD^        #回退至上个版本
```

分支管理

几乎每一种版本控制系统都以某种形式支持分支。

使用分支意味着你可以从开发主线上分离开来，然后在不影响主线的同时继续工作。

用于多功能多人合作工作.

1.创建分支 切换分支:

```
git branch [分支名]      #创建分支  
git checkout [分支名]    #切换分支  
git checkout -b [分支名]  #创建并切换分支
```

2.删除分支

```
git branch -d [分支名]
```

3.合并分支:一旦某分支有了独立内容，你终究会希望将它合并回到你的主分支。你可以使用以下命令将任何分支合并到当前分支中去

```
git merge [分支名]        #将其分支内容合并到当前分支
```

但不是每次合并都会非常成功,如果其中同一个文件内容之间有区别,则会发生冲突.这时Git会将其两分支文件合并并标记冲突位置.

手动解决冲突后使用 `git add` 来告知Git文件冲突已解决.并提交保存

```
git add [文件名]  
git commit -m [信息]
```

保存点

在进行合并分支操作时,一般都会出现代码冲突,为了安全起见,一般可以使用设置保存点方式来进行合并测试.

方式步骤:

1. 确保你在正确的分支上并且你有一个干净的工作状态(暂存区)。

```
git status
```

```
# On branch master  
nothing to commit (working directory clean)
```

2. 创建一个新的分支作为保存点,但不要切换到它.

```
git branch savepoint
```

3. 进行合并操作等

```
git merge spiffy_new_feature
```

4. 切换到您的可视化工具并预测刷新时其视图将如何变化。

5. 对结果满意吗?

如果满意,删除保存点

```
git branch -d savepoint
```

如果不满意,将分支重置到保存点

```
git reset --hard savepoint
```

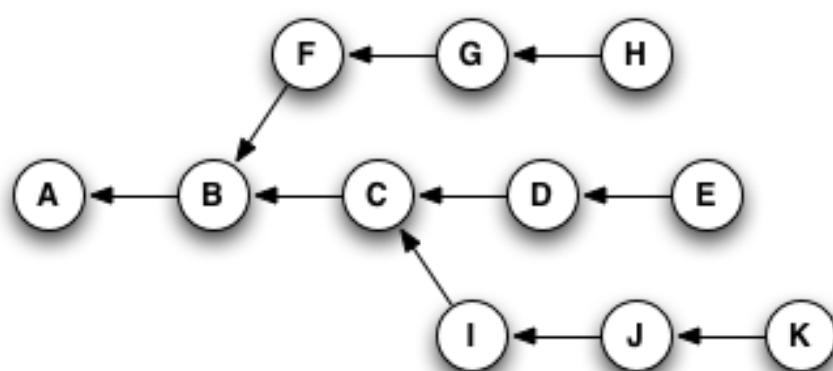
如果要清理, 现在可以使用 `git branch -d savepoint`

cherry-pick

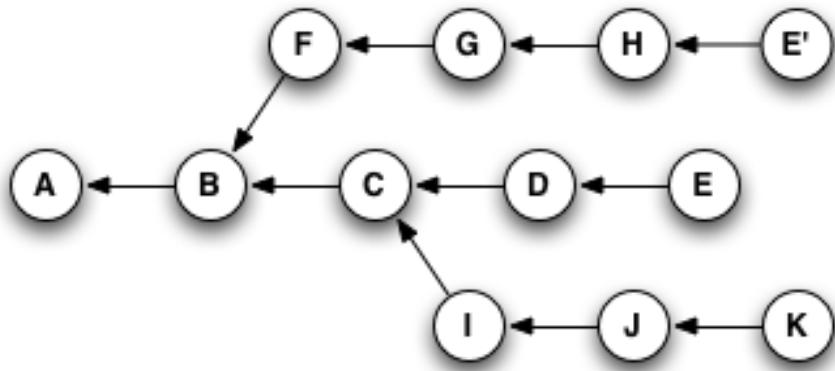
对于多分支的代码库, 将代码从一个分支转移到另一个分支是常见需求。

这时分两种情况。一种情况是, 你需要另一个分支的所有代码变动, 那么就采用合并 (`git merge`) 。

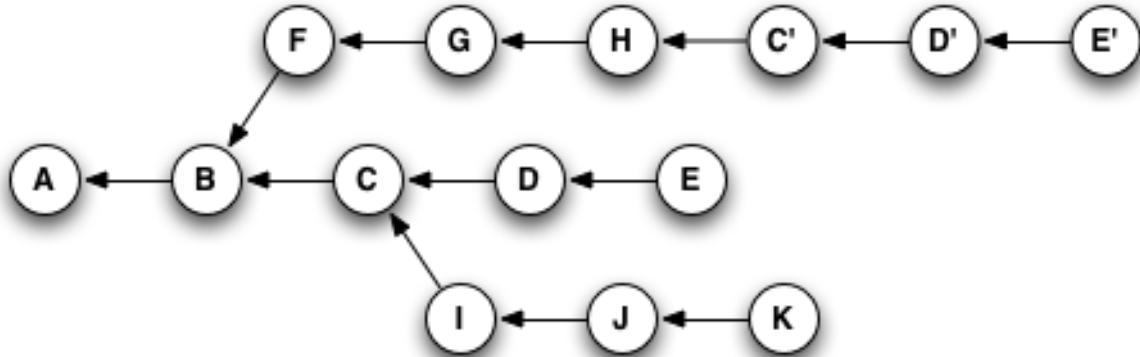
另一种情况是, 你只需要部分代码提交的变动 (某几个提交), 这时可以采用 `cherry-pick` 命令。



如果您目前在此图中的 H提交点,并输入了 `git cherry-pick E[SHA]`,你将获取E提交点到H后面



同理,你可以连续获取并合并多个提交点,你输入类似 `git cherry-pick C D E`, 你会在完成后得到这个:



但其中C D E 必须是按由前到后的先后顺序来的,否则会合并会报错.

MySQL详解

注意规范

注意:所有的创建和删除操作尽量添加 `IF EXISTS` 语句进行判断,以免报错.

- `` :反引号,字段名必须使用它包裹;
- -- info :单行注释,注意其--后必须空出一格才可以.
- /* info */ :多行注释.
- SQL关键语句大小写不敏感,但为了快速阅读以及排错,建议写小写.
- """:引号,Default 默认语句和Comment 备注使用.

修改表

修改表名

```
ALTER TABLE 表名 RENAME AS 新表名 ;
ALTER TABLE teacher RENAME AS student ;
```

增加表字段

```
ALTER TABLE 表名 ADD 字段名 列属性 ;
ALTER TABLE teacher ADD age int(10) ;
```

修改表字段

```
ALTER TABLE 表名 MODIFY 字段名 新的列属性[] ; -- 只能修改字段列的属性以及约束,不能修改字段名
ALTER TABLE teacher MODIFY age int(12) ;
```

```
ALTER TABLE 表名 CHANGE 字段名 新字段名 新的列属性[] ; -- 字段名及列属性都能修改
ALTER TABLE teacher CHANGE age age1 int(13) ;
```

删除表字段

```
ALTER TABLE 表名 DROP 字段名 ;
ALTER TABLE teacher DROP age ;
```

MySQL数据管理

外键(了解即可)

物理外键(不建议使用)

方法一:创建表时,增加约束.

不能单独删除被外键关系的表.

方法二:ALTER TABEL 表名 ADD CONSTRAINT '约束名' FOREIGN KEY('列名') REFERENCES '表名'('列名')

DML语言(全部记住)

数据库意义:数据存储,数据管理

DML语言:数据操作语言

添加

```
-- 插入单行数据
INSERT INTO 表名 (字段1,字段2,字段3,...) VALUES (值1,值2,值3,...);
INSERT INTO 表名 VALUES (值1,值2,值3,...); -- 必须输入表所有字段值,并且位置一一对应,否则报错.

-- 插入多行数据
INSERT INTO 表名 (字段1,字段2,字段3,...) VALUES (值1,值2,值3,...),(值1,值2,值3,...),...
```

修改

```
UPDATE `表名` SET `字段` = `值` WHERE 条件...
UPDATE `表名` SET `字段` = `值`; -- 无条件时默认修改所有列数据
UPDATE `表名` SET `字段` = `值` , `字段` = `值` WHERE 条件... -- 修改多个字段值
```

删除

```
DELETE FROM 表名 WHERE 条件...
TRUNCATE 表名 -- 清空表所有数据
```

TRUNCATE删除所有数据时会将自增字段计数归零,而DELETE则不会.

TRUNCATE删除所有数据不会影响事务.

DELETE删除问题: 重启数据库,在INNODB中,自增列从1开始(内存丢失).

在MyISAM中,自增列不会丢失计数.

DQL查询语言

DQL:Data query language -数据查询语言

Select完整的语法:

顺序很重要:

```
select 去重 要查询的字段 from 表 (注意: 表和字段可以取别名)
xxx join 要连接的表 on 等值判断
where (具体的值, 子查询语句)
Group By (通过哪个字段来分组)
Having (过滤分组后的信息, 条件和 where是一样的, 位置不同)
Order By .. (通过哪个字段排序) [升序/降序]
Limit startIndex, pagesize
```

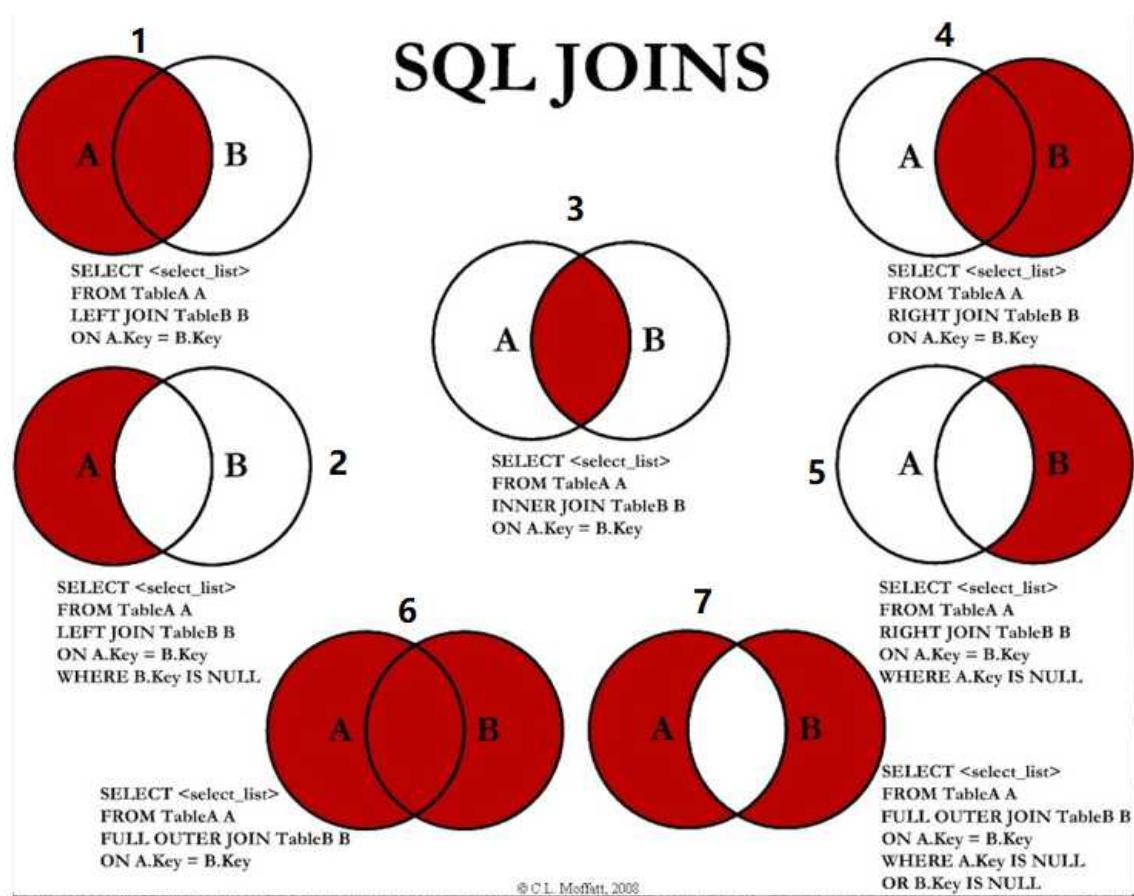
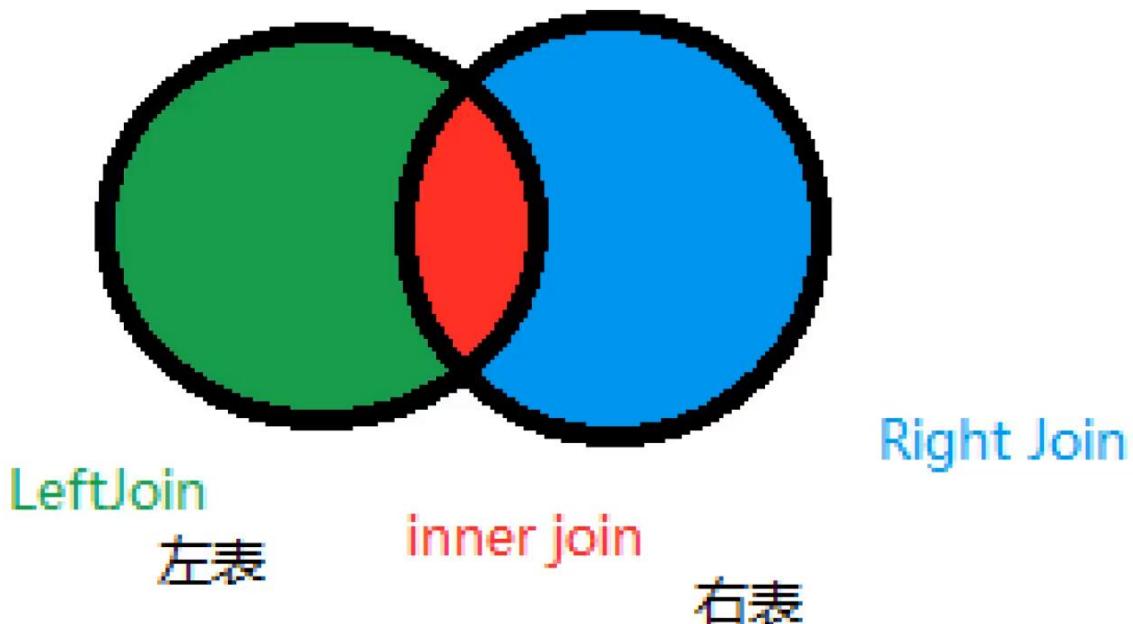
```
1 SELECT [ALL | DISTINCT]
2 { | table.* | [table.field1[as alias1][,table.field2[as alias2]][,...]]}
3 FROM table_name [as table_alias]
4 [left | right | inner join table_name2] -- 联合查询
5 [WHERE ...] -- 指定结果需满足的条件
6 [GROUP BY ...] -- 指定结果按照哪几个字段来分组
7 [HAVING] -- 过滤分组的记录必须满足的次要条件
8 [ORDER BY ...] -- 指定查询记录按一个或多个条件排序
9 [LIMIT {[offset,]row_count | row_countOFFSET offset}];
10 -- 指定查询的记录从哪条至哪条
```

注意:[]括号代表可选的, { }括号代表必选得

```
SELECT * FROM 表名; -- 查询表中所有字段数据
SELECT 字段1,字段2,字段3,... FROM 表名;
SELECT `字段1` AS 别名1,字段2 AS 别名2,字段3 AS 别名3,... FROM 表名; -- 以别名查询出来
字段数据

-- 函数Concat(a,b)将a与b两结果想拼接
SELECT CONCAT('姓名',StudentName) AS 新名字 FROM student;
-- Distinct去重
SELECT DISTINCT 字段 FROM 表名;
```

联表查询JoinON详解



由于mysql中没有 full outer join 操作，所以

mysql中6为1 union 4;

7为2 union 5;

思路:

1. 分析需求,分析查询的字段来自那些表,(连接查询)

2.确定使用那种连接查询? 7种

确定交叉点(这两表那些数据是相同的)

判断的条件: 表1 字段1=表2 字段2

```
-- join on 连接查询
-- where 等值查询
SELECT s.studentNo,studentName,subjectNo,studentResult
FROM student (AS)s  -- AS 可以省略
INNER JOIN/ LEFT JOIN/ RIGHT JOIN result (AS)r
WHERE/ ON s.studentNo=r.studentNo

SELECT s.studentNo,studentName,subjectName,studentResult
FROM student (AS)s  -- AS 可以省略
RIGHT JOIN result (AS)r
ON s.studentNo=r.studentNo
INNER JOIN subject sub
ON r.subjectNO=sub.subjectNO
-- 连接查询可以重叠查询
```

操作	描述
inner join	如果表中至少有一个匹配,返回所有值
left join	即使右表中没有匹配的数据,也会从左表中返回
right join	即使左表中没有匹配的数据,也会从右表中返回

自连接

自己的表和自己的表连接, 核心:一张表拆为两条一样的表

	categoryid	pid	categoryName
	7	5	ps技术
	6	3	web开发
	2	1	信息技术
	8	2	办公信息
	4	3	数据库
	5	1	美术设计
	3	1	软件开发

-- 把一张表看出两张一模一样的表

```
SELECT a.categoryName AS '父栏目',b.categoryName AS '子栏目'
FROM category AS a,category AS b
WHERE a.categoryid=b.pid
```

父栏目	子栏目
软件开发	数据库
软件开发	web开发
美术设计	ps技术
信息技术	办公信息

分页和排序

分页:limit 排序:order by

Order by:通过字段排序:升序 ASC ,降序 DESC

```
SELECT 字段1,字段2,....  
FROM 表名  
WHERE 条件  
ORDER BY 字段 (ASC/ DESC)
```

Limit 起始值,显示个数 (起始值首项为0)

```
SELECT 字段1,字段2,....  
FROM 表名  
WHERE 条件  
Limit 0,5 -- 从第一条数据开始,显示5条数据
```

-- 设定每页显示5条数据
--第一页 limit 0,5 (1-1)*5
--第二页 limit 5,5 (2-1)*5
--第三页 limit 10,5 (3-1)*5
--第N页 limit (N-1)*PageSize,PageSize
--[PageSize:页面大小,(N-1)*PageSize:起始值,N:当前页]

子查询

常用函数

ABS(-8) --绝对值 CEILING(9.4) --向上取整 FLOOR(9.4) --向下取整
RAND() --返回一个0-1之间的随机数 CHAR_LENGTH('scarf') --返回字符串的长度 CONCAT('2','3') --拼接字符串
REPLACE('2333','23','41') --替换指定字符串内容 SUBSTR('safer',1,3) --返回指定位置字符串(字符串,截取位置,截取长度)

聚合函数及分组过滤

GROUP BY 字段:通过字段来分组

COUNT():查询表中记录条数

COUNT(字段) -- 会忽略所有的NULL值

COUNT(*) -- 不会忽略NULL值,本质是计算行数

COUNT(1) -- 不会忽略NULL值,本质是计算行数

SUM(字段):计算所有行总和

AVG(字段):计算所有行平均分

MAX(字段):查询所有行中最高分

MIN(字段):查询所有行中最低分

WHERE 条件中不能包含聚合函数.聚合函数过滤 需要使用 HAVING

```
SELECT SubjectName, AVG(studentResult) AS '平均分'  
FROM result  
GROUP BY SubjectNo  
HAVING 平均分>80
```

事务

1.事务简介

(1)在 MySQL 中只有使用了 InnoDB 数据库引擎的数据库或表才支持事务。

(2)事务处理可以用来维护数据库的完整性，保证成批的 SQL 语句要么全部执行，要么全部不执行。

(3)事务用来管理 insert, update, delete 语句。

(4)MySQL 自动默认开启事务自动提交。

```
SET autocommit = 0 /* 关闭 */  
SET autocommit = 1 /* 开启(默认) */
```

2.事务四大特征/原则

一般来说，事务是必须满足4个条件 (ACID)：：原子性 (Atomicity, 或称不可分割性)、一致性 (Consistency)、隔离性 (Isolation, 又称独立性)、持久性 (Durability)。

原子性：一个事务 (transaction) 中的所有操作，**要么全部完成，要么全部失败**,不会只发生其中一个动作。

一致性：在事务开始结束前后**数据结果要保证一定一致**.

隔离性：数据库允许多个并发事务**同时**对其数据进行读写和修改的能力，多个并发事务不会互相影响。

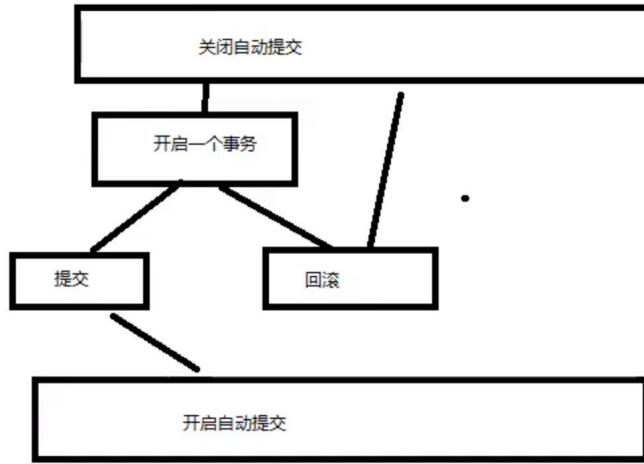
事务隔离分为不同级别：

- 脏读：指一个事务读取了另一个事务未提交的数据。
- 不可重复读：在一个事务读取数据时，多次读取结果不同。(不一定错误，只是场合不对)
- 幻读(虚读)：在一个事务读取了别的事务插入的数据，导致前后读取不一致。

持久性：事务处理结束后的数据不会随外界原因而导致数据丢失，**一旦事务提交不可逆**。

3.MYSQL 事务处理

手动处理事务



0.关闭事务自动提交

```
SET autocommit = 0
```

1.事务开启

```
START TRANSACTION -- 标记一个事务的开始,从这之后的SQL语句都在一个事务内.
```

2.提交事务 :持久化 (成功的话)

```
COMMIT
```

3.回滚事务 :回到之前的样子(失败的话)

```
ROLLBACK
```

3.事务结束

```
SET TRANSACTION = 1
```

4.保存点

```
SAVEPOINT xxx1(保存点名) -- 设置一个叫xxx1的事务保存点
ROLLBACK TO SAVEPOINT xxx1(保存点名) -- 回滚到xxx1保存点
```

```
RELEASE SAVEPOINT xxx1(保存点名) -- 撤销xxx1保存点
```

索引

索引是帮助Mysql高效获取数据的数据结构。

但索引的缺点：会降低更新表的速度，如对表进行INSERT、UPDATE和DELETE。因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件。

索引分类

- 主键索引 (PRIMARY KEY)
 - 唯一标识,主键不可重复,只能有一个列为主键
- 唯一索引 (UNIQUE KEY)
 - 避免重复的列出现,唯一索引可以重复,多个列可以标识为 唯一索引
- 常规索引 (KEY/INDEX)
 - 默认的index,key关键字来设置
- 全文索引(FULL TEXT)
 - 在特定的数据库才有,MyISAM
- 快速定位数据

索引使用

-- 在创建表时给字段添加索引

-- 创建完毕后,添加索引

```
SHOW INDEX FROM 表名 -- 显示表中的所有索引

ALTER TABLE 表名 ADD FULLTEXT/UNIQUE 索引名(字段名) -- 修改表结构,添加一个全文索引

CREATE INDEX/UNIQUE INDEX 索引名 ON 表名 (字段名) -- 在一个表中添加一个常规索引

/*在创建表时添加索引/
CREATE TABLE mytable(
    ID INT NOT NULL,
    username VARCHAR(16) NOT NULL,
    INDEX/UNIQUE [索引名] (username(length)) -- 如果是CHAR, VARCHAR类型, length可以小于字段实际长度; 如果是BLOB和TEXT类型, 必须指定 length。
);

DROP INDEX [索引名] ON 表名; -- 删除索引
```

```
-- 插入100万条数据
CREATE FUNCTION mock_date()
Returns INT
DELIMITER $$ -- 写函数之前必须写,标志
BEGIN
    DECLARE num INT DEFAULT 1000000;
    DECLARE i INT DEFAULT 0;
    WHILE i<num DO
        -- 插入语句
        INSERT INTO 表名(字段1,字段2,...) VALUES(CONCAT('值1',i),CONCAT('值2',i),...);
        SET i=i+1;
    END WHILE;
    RETURN i;
```

```
END;
```

索引原则

- 索引不是越多越好,表中数据非常的时才考虑.
- 不要对进程变动数据加索引
- 小数据量的表不需要加索引
- 索引一般加载常用来查询的字段上!

索引的数据结构:

Hash 类型的索引

Btree :InnoDB默认的默认索引类型

权限管理和备份

用户管理

用户表: mysql.user表

本质:对这张表进行增删改查

```
CREATE USER 用户名 IDENTIFIED BY '密码' -- 创建一个默认用户
SET PASSWORD=PASSWORD('密码') -- 修改当前用户密码
SET PASSWORD FROM 用户名=PASSWORD('密码') -- 修改指定用户密码

RENAME USER 用户名 TO 新用户名 -- 修改指定用户名
GRANT ALL PRIVILEGES ON *.* TO 用户名 -- 给指定用户授权所有权限(除了给别人授权)在所有数据库中所有表
SHOW GRANT FOR 用户名 -- 查看指定用户权限
REVOKE ALL PRIVILEGES ON *.* FROM 用户名 -- 撤销指定用户所有权限在所有数据库中所有表

DROP USER 用户名 -- 删除指定用户
```

Mysql备份

- 1.直接拷贝data文件夹下的物理文件
- 2.使用可视化工具手动导出
- 3.使用命令行导出 mysqldump 命令行

```
mysqldump -h`mysql地址` -u`用户名` -p`密码` 数据库名 表名 表2 表3 ... >物理磁盘位置:/文件名.sql(导出地址)
```

数据库三大范式

为什么需要数据库范式化?

- 信息重复
- 更新异常

- 插入异常
 - 无法正常显示信息
- 删除异常
 - 丢失有效信息

第一范式(1NF)

原子性: 保证每列不可再分

第二范式(2NF)

前提: 满足第一范式

每张表只描述一个事情

第三范式(3NF)

前提: 满足第一范式和第二范式

确保数据表中每列数据和主键直接相关, 而不能间接相关.

规范性和性能的问题:

关联查询的表不能成过三张表

- 考虑商业化需求和目标(成本, 用户体验)数据库的性能更加重要
- 规范性能的问题的时候, 需要适当考虑一下示范性
- 故意给某些表增加一些冗余的字段(从多表查询变单表查询)
- 故意增加一些计算列(大数据量降低为小数据量的查询: 索引)

Mybatis XML映射文件详解

基本SQL XML语句

Select语句 - 查询

```
<select
    id="selectPerson"
    parameterType="int"
    resultType="hashmap"
    resultMap="personResultMap"
    flushCache="false"
    useCache="true"
    timeout="10"
    fetchSize="256"
    statementType="PREPARED"
    resultSetType="FORWARD_ONLY">
    ...
</select>
```

属性	描述
id	在命名空间中唯一的标识符，可以被用来引用这条语句。
parameterMap	将会传入这条语句的参数的类全限定名或别名。这个属性是可选的。
resultType	期望从这条语句中返回结果的类全限定名或别名。注意，如果返回的是集合，那应该设置为集合包含的类型，而不是集合本身的类型。
resultMap	对外部 resultMap 的命名引用。 resultType 和 resultMap 之间只能同时使用一个。
flushCache	将其设置为 true 后，只要语句被调用，都会导致本地缓存和二级缓存被清空。
useCache	将其设置为 true 后，将会导致本条语句的结果被二级缓存缓存起来。
timeout	抛出异常之前，驱动程序等待数据库返回请求结果的秒数。

Insert、Update、Delete 语句

属性	描述
id	在命名空间中唯一的标识符，可以被用来引用这条语句。
parameterType	将会传入这条语句的参数的类全限定名或别名。这个属性是可选的。
flushCache	将其设置为 true 后，只要语句被调用，都会导致本地缓存和二级缓存被清空，默认值：（对 insert、update 和 delete 语句）true。
useGeneratedKeys	支持自动生成主键的字段，需要再填写keyProperty 设置（仅适用于 insert 和 update）
keyProperty	指定能够唯一识别对象的属性（仅适用于 insert 和 update）

Foreach语句

```
<foreach item="item" collection="list" separator=",">
    #{item.username}, #{item.password}, #{item.email}, #{item.bio}
</foreach>
```

- **item:** 集合中元素迭代时的别名，该参数为必选。
- **index:** 在list和数组中,index是元素的序号，在map中，index是元素的key，该参数可选
- **open:** foreach代码的开始符号，一般是(和close=")"合用。常用在in(),values()时。该参数可选
- **separator:** 元素之间的分隔符，例如在in()的时候，separator=","会自动在元素中间用","隔开，避免手动输入逗号导致sql错误，如in(1,2,)这样。该参数可选。
- **collection:** 要做foreach的对象，作为入参时，List对象默认用"list"代替作为键，数组对象有"array"代替作为键，Map对象没有默认的键。当然在作为入参时可以使用@Param("keyName")来设置键，设置keyName后，list,array将会失效。

Sql语句

```
<sql id="userColumns"> ${alias}.id,${alias}.username,${alias}.password </sql>
```

这个 SQL 片段可以在其它语句中使用。

```
<select id="selectUsers" resultType="map">
    select
        <include refid="userColumns"><property name="alias" value="t1"/></include>,
        <include refid="userColumns"><property name="alias" value="t2"/></include>
    from some_table t1
    cross join some_table t2
</select>
```

#{}/\${} -字符串替换 语句

使用 #{} 参数语法，会在 SQL 语句中直接插入一个转义的字符串。更安全，更迅速，通常也是首选做法。

使用 \${} 参数语法，直接会在 SQL 语句中直接插入一个不转义的字符串。但用这种方式接受用户的输入，并用作语句参数是不安全的，会导致潜在的 SQL 注入攻击。

```
#{{property, javaType=int, jdbcType=NUMERIC}}
```

和MyBatis 的其它部分一样，几乎总是可以根据参数对象的类型确定 javaType，除非该对象是一个 `HashMap`。这个时候，你需要显式指定 `javaType` 来确保正确的类型处理器（`TypeHandler`）被使用。

JDBC 要求，如果一个列允许使用 null 值，并且会使用值为 null 的参数，就必须要指定 JDBC 类型 (`jdbcType`) 。

对于数值类型，还可以设置 `numericScale` 指定小数点后保留的位数。

当 SQL 语句中的元数据（如表名或列名）是动态生成的时候，字符串替换将会非常有用。举个例子，如果你想 `select` 一个表任意一列的数据时，不需要这样写：

```
@Select("select * from user where id = #{id}")
User findById(@Param("id") long id);

@Select("select * from user where name = #{name}")
User findByName(@Param("name") string name);

@Select("select * from user where email = #{email}")
User findByEmail(@Param("email") string email);

// 其它的 "findByXXX" 方法
...
```

而是可以只写这样一个方法：

```
@select("select * from user where ${column} = #{value}")
User findByColumn(@Param("column") String column, @Param("value") String value);
```

其中 `#{column}` 会被直接替换，而 `#{value}` 会使用 `?` 预处理。

ResultMap -结果映射

ResultMap 元素是 MyBatis 中最重要最强大的元素。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。其设计思想是，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。

```
<!-- 非常复杂的结果映射 -->
<resultMap id="detailedBlogResultMap" type="Blog">
    <constructor>
        <idArg column="blog_id" javaType="int"/>
    </constructor>
    <result property="title" column="blog_title"/>
    <association property="author" javaType="Author">
        <id property="id" column="author_id"/>
        <result property="username" column="author_username"/>
        <result property="password" column="author_password"/>
        <result property="email" column="author_email"/>
        <result property="bio" column="author_bio"/>
        <result property="favouriteSection" column="author_favourite_section"/>
    </association>
    <collection property="posts" ofType="Post">
        <id property="id" column="post_id"/>
        <result property="subject" column="post_subject"/>
        <association property="author" javaType="Author"/>
        <collection property="comments" ofType="Comment">
            <id property="id" column="comment_id"/>
        </collection>
        <collection property="tags" ofType="Tag" >
            <id property="id" column="tag_id"/>
        </collection>
        <discriminator javaType="int" column="draft">
            <case value="1" resultType="DraftPost"/>
        </discriminator>
    </collection>
</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

id & result

```
<id property="id" column="post_id"/>
<result property="subject" column="post_subject"/>
```

这些元素是结果映射的基础。`id` 和 `result` 元素都将一个列的值映射到一个简单数据类型（String, int, double, Date 等）的属性或字段。

属性	描述
<code>property</code>	映射到列结果的字段或属性。如果 JavaBean 有这个名字的属性（property），会先使用该属性。否则 MyBatis 将会寻找给定名称的字段（field）。
<code>column</code>	数据库中的列名，或者是查询、修改、删除时的列的别名。
<code>javaType</code>	一个 Java 类的全限定名，或一个类型别名（关于内置的类型别名，可以参考上面的表格）。如果你映射到一个 JavaBean，MyBatis 通常可以推断类型。
<code>jdbcType</code>	JDBC 类型，所支持的 JDBC 类型参见这个表格之后的“支持的 JDBC 类型”。只需要在可能执行插入、更新和删除的且允许空值的列上指定 JDBC 类型。

Association - 关联

关联（association）元素处理“有一个”类型的关系。比如，在我们的示例中，一个博客有一个用户。关联结果映射和其它类型的映射工作方式差不多。你需要指定目标属性名以及属性的 `javaType`（很多时候 MyBatis 可以自己推断出来），在必要的情况下你还可以设置 JDBC 类型。

关联的不同之处是，你需要告诉 MyBatis 如何加载关联。MyBatis 有两种不同的方式加载关联：

1. 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。
 2. 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。
- 关联的嵌套 Select 查询

属性	描述
<code>column</code>	数据库中的列名，或者是查询、修改、删除时的列的别名。
<code>select</code>	用于加载复杂类型属性的映射语句的 ID。

```

<resultMap id="blogResult" type="Blog">
    <association property="author" column="author_id" javaType="Author"
    select="selectAuthor"/>
</resultMap>

<select id="selectAuthor" resultType="Author">
    SELECT * FROM AUTHOR WHERE ID = #{id}
</select>

<select id="selectBlog" resultMap="blogResult">
    SELECT * FROM BLOG WHERE ID = #{id}
</select>

```

两个 select 查询语句：一个用来加载博客（Blog），另外一个用来加载作者（Author），而且博客的结果映射描述了应该使用 `selectAuthor` 语句加载它的 author 属性。

方式虽然很简单，但这个方法会导致成百上千的 SQL 语句被执行。影响SQL性能。

- 关联的嵌套结果映射

```

<resultMap id="blogResult" type="Blog">
    <id property="id" column="blog_id" />
    <result property="title" column="blog_title"/>
    <association property="author" column="blog_author_id" javaType="Author"
    resultMap="authorResult"/>
</resultMap>

<resultMap id="authorResult" type="Author">
    <id property="id" column="author_id"/>
    <result property="username" column="author_username"/>
    <result property="password" column="author_password"/>
    <result property="email" column="author_email"/>
    <result property="bio" column="author_bio"/>
</resultMap>

```

上面的示例使用了外部的结果映射元素来映射关联。这使得 Author 的结果映射可以被重用。博客（Blog）作者（author）的关联元素委托名为 “authorResult”的结果映射来加载作者对象的实例。

id 元素在嵌套结果映射中扮演着非常重要的角色。你应该总是指定一个或多个可以唯一标识结果的属性。

也可以所有的结果映射放在一个具有描述性的结果映射元素中。直接将结果映射作为子元素嵌套在内。

```

<resultMap id="blogResult" type="Blog">
    <id property="id" column="blog_id" />
    <result property="title" column="blog_title"/>
    <association property="author" javaType="Author">
        <id property="id" column="author_id"/>
        <result property="username" column="author_username"/>
        <result property="password" column="author_password"/>
        <result property="email" column="author_email"/>
        <result property="bio" column="author_bio"/>
    </association>
</resultMap>

```

Collection -集合

```

<collection property="posts" ofType="domain.blog.Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <result property="body" column="post_body"/>
</collection>

```

一个博客 (Blog) 只有一个作者 (Author)。但一个博客有很多文章 (Post)。

```
private List<Post> posts;
```

映射嵌套结果集合到一个 List 中，可以使用集合元素。和关联元素一样，可以使用嵌套 Select 查询，或基于连接的嵌套结果映射集合。

- 集合的嵌套 Select 查询

```

<resultMap id="blogResult" type="Blog">
    <collection property="posts" javaType="ArrayList" column="id" ofType="Post"
    select="selectPostsForBlog"/>
</resultMap>

<select id="selectBlog" resultMap="blogResult">
    SELECT * FROM BLOG WHERE ID = #{id}
</select>

<select id="selectPostsForBlog" resultType="Post">
    SELECT * FROM POST WHERE BLOG_ID = #{id}
</select>

```

在一般情况下，MyBatis 可以推断 javaType 属性，因此并不需要填写。

- 集合的嵌套结果映射

```

<resultMap id="blogResult" type="Blog">
    <id property="id" column="blog_id" />
    <result property="title" column="blog_title"/>
    <collection property="posts" ofType="Post" resultMap="blogPostResult"
    columnPrefix="post_" />
</resultMap>

<resultMap id="blogPostResult" type="Post">
    <id property="id" column="id"/>
    <result property="subject" column="subject"/>
    <result property="body" column="body"/>
</resultMap>

```

其中columnPrefix属性其含义是将XXX自动添加到它下面的column中。

Discriminator - 鉴定器

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）。鉴定器（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。鉴定器的概念很好理解——它很像 Java 语言中的 switch 语句。

```

<resultMap id="vehicleResult" type="Vehicle">
    ...
    <discriminator javaType="int" column="vehicle_type">
        <case value="1" resultMap="carResult"/>
        <case value="2" resultMap="truckResult"/>
        <case value="3" resultMap="vanResult"/>
        <case value="4" resultMap="suvResult"/>
    </discriminator>
</resultMap>

<resultMap id="carResult" type="Car">
    <result property="doorCount" column="door_count" />
</resultMap>

```

动态 SQL

MyBatis 3 替换了之前的大部分元素，大大精简了元素种类，现在要学习的元素种类比原来的一半还要少。

if语句

```

<if test="title != null">
    XXX
</if>

```

如果传入了“title”参数，那么就会额外输出XXX内容。

choose、when、otherwise语句

```
<choose>
    <when test="title != null">
        XXX
    </when>
    <when test="author != null and author.name != null">
        XXX
    </when>
    <otherwise>
        XXX
    </otherwise>
</choose>
```

MyBatis 提供了 choose 元素，它有点像 Java 中的 switch 语句。

trim、where、set语句

```
<where>
    <if test="state != null">
        XXX
    </if>
    <if test="title != null">
        XXX
    </if>
    <if test="author != null and author.name != null">
        XXX
    </if>
</where>
```

TK.mybatis框架使用

```
<dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper-spring-boot-starter</artifactId>
    <version>2.0.3-beta</version>
</dependency>

<dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper</artifactId>
    <version>4.0.0</version>
</dependency>
```

创建BaseMapper接口继承Mapper, ConditionMapper

```
public interface BaseMapper<T> extends Mapper<T>, ConditionMapper<T>
    IdsMapper<T>,
    InsertListMapper<T>{
}
```

创建对应Dao层接口,例如类的Dao

```
public interface SysUserMapper extends BaseMapper<SysUser> {
    ...
}
```

BaseMapper方法:

```
/**
 * 保存一个实体, null属性也会保存
 *
 * @param record
 * @return
 */
int insert(T record);

/**
 * 保存一个实体, null属性不会保存
 *
 * @param record
 * @return
 */
int insertSelective(T record);

/**
 * 根据实体属性作为条件进行删除, 查询条件使用等号
 */
int delete(T record);

/**
 * 根据主键更新属性不为null的值
 */
int updateByPrimaryKeySelective(T record);

/**
 * 根据实体中的属性值进行查询, 查询条件使用等号
 */
List<T> select(T record);

/**
 * 查询全部结果, select(null)方法能达到同样的效果
 */
List<T> selectAll();

/**
 * 根据实体中的属性进行查询, 只能有一个返回值, 有多个结果是抛出异常, 查询条件使用等号
 */
T selectOne(T record);

/**
```

```
* 根据实体中的属性查询总数，查询条件使用等号  
*/  
int selectCount(T record);
```

IdsMapper方法：

```
/**  
 * 根据主键@Id进行查询，多个Id以逗号，分割  
 * @param id  
 * @return  
 */  
List<T> selectByIds(String ids);  
  
/**  
 * 根据主键@Id进行删除，多个Id以逗号，分割  
 * @param id  
 * @return  
 */  
int deleteByIds(String ids);
```

ConditionMapper方法：

```
/**  
 * 根据Condition条件进行查询  
 */  
public List selectByCondition(Object condition);  
  
/**  
 * 根据Condition条件进行查询  
 */  
public int selectCountByCondition(Object condition);  
  
/**  
 * 根据Condition条件删除数据，返回删除的条数  
 */  
public int deleteByCondition(Object condition);  
  
/**  
 * 根据Condition条件更新实体`record`包含的全部属性，null值会被更新，返回更新的条数  
 */  
public int updateByCondition(T record, Object condition);  
  
/**  
 * 根据Condition条件更新实体`record`包含的全部属性，null值会被更新，返回更新的条数  
 */  
public int updateByConditionSelective(T record, Object condition);
```

其中传入的Object condition应为tk.mybatis.mapper.entity.Condition

JWT (JSON Web Token)跨域身份验证

结构: *heard.payload.signature*

标头.有效负载.签名

结构解析

1. 标头 (heard) : 包含令牌的类型以及使用的签名算法。
2. 有效负载 (payload) : 包含实体和其他数据的声明。尽管它可以防止被篡改，但任何人依然可以读取这些签名的信息。除非加密，否则不要在 JWT 的有效载荷或头部元素中放入秘密信息。
3. 签名 (singnature) : 签名用于验证消息是否在整个过程中被更改。对于使用私钥签名的令牌，它还可以验证 JWT 的发送方是否是它所说的那个发送方。

使用JWT

1. Maven依赖

```
<dependency>
    <groupId>com.auth0</groupId>
    <artifactId>java-jwt</artifactId>
    <version>3.11.0</version>
</dependency>
```

2. 利用数据，生成对应Token

```
Calendar instance = calendar.getInstance();
instance.add(calendar.SECOND, 20);

String token = JWT.create()
    .withHeader(map) //设置header
    .withClaim("userId", 21) //设置payload
    .withClaim("userName", "xxx")
    .withExpiresAt(instance.getTime()) //指定令牌过期时间
    .sign(Algorithm.HMAC256("@QWER@")); //设置singnature 其中使用HMAC256算法
```

3. 使用对应Token，获取数据

```
JWTVerifier jwtVerifier = JWT.require(Algorithm.HMAC256("@QWER@")).build(); //获取对应Jwt算法解析器

//解析对应Token，获得解码后的Jwt
DecodedJWT decodedJWT =
jwtVerifier.verify("eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9" +
    ".eyJ1c2VyTmFtZSI6Inh4ecISImV4CCI6MTYwMzc2NTU4MCwidXNlcklkIjoyMX0" +
    ".CHJX4xoGPFB18Qgk6me_44F1y_lunIDS9v0DjfLudw8");

//获取对应Token中数据
System.out.println(decodedJWT.getClaim("userId").asInt());
System.out.println(decodedJWT.getClaim("userName").asString());
```

SpringBoot+Mybatis整合

Maven依赖

```
<!--mysql数据库驱动-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>

<!--mybatis-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>2.1.0</version>
</dependency>

<!-- https://mvnrepository.com/artifact/com.alibaba/druid -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>1.2.5</version>
</dependency>
```

操作方法

application.yml文件配置

```
spring:
  datasource:
    username: root
    password: 123456
    #?serverTimezone=UTC解决时区的报错
    url: jdbc:mysql://localhost:3306/springboot?
    serverTimezone=UTC&useUnicode=true&characterEncoding=utf-8
    driver-class-name: com.mysql.cj.jdbc.Driver
    type: com.alibaba.druid.pool.DruidDataSource

    #Spring Boot 默认是不注入这些属性值的，需要自己绑定
    #druid 数据源专有配置
    initialSize: 5
    minIdle: 5
    maxActive: 20
    maxWait: 60000
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
    poolPreparedStatements: true

    #配置监控统计拦截的filters, stat:监控统计、log4j: 日志记录、wall: 防御sql注入
    #如果允许时报错 java.lang.ClassNotFoundException: org.apache.log4j.Priority
    #则导入 log4j 依赖即可，Maven 地址:
    https://mvnrepository.com/artifact/log4j/log4j
```

```
filters: stat,wall,log4j
maxPoolPreparedStatementPerConnectionSize: 20
useGlobalDataSourceStat: true
connectionProperties: druid.stat.mergeSql=true;druid.stat.showSqlMillis=500
```

1.编写实体类

```
@Table(name = "SYS_ORDER") //声明此对象映射到数据库的数据表
public class SysOrder implements Serializable {
    private static final long serialVersionUID = 2688088497753868718L;

    @Id //主键字段
    private String merchantOrderNo;
    private String merchantNo;
    private Integer amount;
    ...

    public String getMerchantOrderNo() {
        return merchantOrderNo;
    }

    public void setMerchantOrderNo(String merchantOrderNo) {
        this.merchantOrderNo = merchantOrderNo;
    }

    public String getMerchantNo() {
        return merchantNo;
    }

    public void setMerchantNo(String merchantNo) {
        this.merchantNo = merchantNo;
    }

    ...
}
```

2.编写mapper接口

```
@Mapper //指定这是一个操作数据库的mapper
public interface SysorderMapper extends BaseMapper<Sysorder> {

    BizDataDTO getBizDataByOrderNo(@Param("merchantOrderNo") String
merchantOrderNo);

    List<SysorderDTO> queryOrderData(@Param("orderQueryRequest")
OrderQueryRequest orderQueryRequest);

    SysOrderDTO getOrderByOrderNo(@Param("merchantOrderNo") String
merchantOrderNo);

    ...
}
```

3. 编写mapper xml映射文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="cn.com.bossoftcq.ipp.mapper.SysOrderMapper">
    <resultMap id="BaseResultMap" type="cn.com.bossoftcq.ipp.model.doo.SysOrder">
        <id column="MERCHANT_ORDER_NO" jdbcType="VARCHAR" property="merchantorderNo"/>
    <result column="MERCHANT_NO" jdbcType="VARCHAR" property="merchantNo" />
    <result column="AMOUNT" jdbcType="DECIMAL" property="amount" />
    </resultMap>
    ...
    <select id="getOrderByOrderNo"
            parameterType="java.lang.String" resultMap="orderwithBizList">
        SELECT
            so.* , bi.biz_ID, bi.sys_type
        FROM
            sys_order so,sys_order_biz bi
        where
            so.merchant_order_no = bi.merchant_order_no
            and so.MERCHANT_ORDER_NO = #{merchantOrderNo,jdbcType=VARCHAR}
    </select>
    ...
</mapper>
```

4. 编写service接口

```
public interface Sysorderservice extends BaseService<SysOrder>{
    PageInfo<O> queryorderData(OrderQueryRequest orderQueryRequest);
    Result cancelOrderByMerchantorderNo(String merchantorderNo);
    ...
}
```

5. 编写service实现类

```
@Service //需要在接口实现类中使用@Service注解，才能被SpringBoot扫描
public class SysorderServiceImpl extends BaseServiceImpl<SysOrder> implements SysOrderService {
    private static final Logger log =
    LoggerFactory.getLogger(SysorderServiceImpl.class);

    @Autowired
    private SysOrderMapper orderMapper;
    @Autowired
    private EduBizSplitDataMapper eduBizSplitDataMapper;
    ...
    @Override
    public PageInfo<O> queryOrderData(OrderQueryRequest orderQueryRequest) {
        //如果是super则查询所有，否则上查下
        UserDTO user = UserUtil.getUser();
        if (user.isSuperAdmin()) {
```

```

        orderQueryRequest.setDeptId(null);
    }else{
        orderQueryRequest.setDeptId(user.getDeptId());
    }

PageHelper.startPage(orderQueryRequest.getPage(),orderQueryRequest.getLimit());
List<SysOrderDTO> orders =
orderMapper.queryOrderData(orderQueryRequest);
if (!orders.isEmpty()) {
    PageInfo<SysOrderDTO> pageInfo = new PageInfo<>(orders);
    return new PageInfoBO(pageInfo.getTotal(), pageInfo.getList());
}
return null;
}

...
}

```

6.编写controller文件

```

@RestController
public class OrderController {

    private static final Logger LOGGER =
LoggerFactory.getLogger(OrderController.class);

    @Autowired
    private SysOrderService sysOrderService;

    @GetMapping("/order/queryData")
    public Result queryOrderData(OrderQueryRequest orderQueryRequest) {
        return
Result.success(sysOrderService.queryOrderData(orderQueryRequest));
    }

}

```

7.配置property文件(或者yml文件)

```

mybatis:
# Mybatis配置Mapper路径
mapper-locations: classpath:mapping/*.xml
# Mybatis配置Model类对应
type-aliases-package: cn.com.bossoftcq.ipp.model

```

Mybatis逆向工程

操作方法

一.Maven依赖

```
<dependency>
    <groupId>org.mybatis.generator</groupId>
    <artifactId>mybatis-generator-core</artifactId>
    <version>1.3.7</version>
</dependency>
```

二.配置generatorConfig.xml文件

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE generatorConfiguration
PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
"http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">

<generatorConfiguration>
    <context id="sakila" targetRuntime="MyBatis3">

        <!-- 生成的Java文件的编码 -->
        <property name="javaFileEncoding" value="UTF-8" />
        <!-- 格式化java代码 -->
        <property name="javaFormatter"
            value="org.mybatis.generator.api.dom.DefaultJavaFormatter" />
        <!-- 格式化XML代码 -->
        <property name="xmlFormatter"
            value="org.mybatis.generator.api.dom.DefaultXmlFormatter" />
        <!-- 使用tkMapper插件 -->
        <plugin type="tk.mybatis.mapper.generator.MapperPlugin">
            <!-- <property name="mappers"
value="tk.mybatis.mapper.common.Mapper,tk.mybatis.mapper.hsqldb.HsqldbMapper"/> --
            <!-- value可以为默认的,也可以为用户定义的tkMapper实体类 -->
            <property name="mappers" value="tk.mybatis.mapper.common.Mapper"/>
            <!-- caseSensitive默认false, 当数据库表名区分大小写时, 可以将该属性设置为true
-->
            <property name="caseSensitive" value="true"/>
        </plugin>

        <commentGenerator>
            <!-- 是否去除自动生成的注释 true: 是 : false:否 -->
            <property name="suppressAllComments" value="true" />
            <!-- 是否生成注释代时间戳 -->
            <property name="suppressDate" value="true" />
        </commentGenerator>

        <!--数据库连接的信息: 驱动类、连接地址、用户名、密码 -->
        <jdbcConnection driverClass="com.mysql.cj.jdbc.Driver"
            connectionURL="jdbc:mysql://localhost:3306/23?
useSSL=false&useUnicode=false&characterEncoding=UTF8"
            userId="root"
            password="200428">
        </jdbcConnection>

        <javaTypeResolver>
            <!-- 把jdbc对应的日期类型转成java8中的LocateDateTime类型 -->
            <property name="useJSR310Types" value="true"/>
        </javaTypeResolver>
    
```

```

<!-- targetProject:生成PO类的位置 -->
<javaModelGenerator
targetPackage="com.example.seconddemowebrestfulcrud.entity"
    targetProject=".\\src\\main\\java">
    <!-- 从数据库返回的值被清理前后的空格 -->
    <property name="trimStrings" value="true"/>
</javaModelGenerator>

<!-- targetProject:mapper映射文件生成的位置 -->
<sqlMapGenerator targetPackage="mapper"
    targetProject=".\\src\\main\\resources"/>

<!-- targetPackage: mapper接口生成的位置 -->
<javaClientGenerator type="XMLMAPPER"

targetPackage="com.example.seconddemowebrestfulcrud.mapper"
    targetProject=".\\src\\main\\java"/>

<!--生成service,serviceImpl-->
<javaServiceGenerator targetPackage="com.shsoft.platform.service"
targetProject="src/main/java"

implementationPackage="com.shsoft.platform.service">
    </javaServiceGenerator>

<!-- 指定数据库表 -->
<table schema="23" tableName="user_info"
    domainObjectName="User"
    enableCountByExample="false"
    enableDeleteByExample="false" enableSelectByExample="false"
    enableUpdateByExample="false">
    <!-- 如果设置为true, 生成的entity类会直接使用column本身的名字, 而不会再使用驼峰命名法 -->
    <property name="useActualColumnNames" value="false" />
    <!-- 生成的SQL中的表名将不会包含schema和catalog前缀 -->
    <property name="ignoreQualifiersAtRuntime" value="true" />
</table>
</context>
</generatorConfiguration>

```

三.编写生成器

```

public class Generator {

    public void generator() throws Exception {
        List<String> warnings = new ArrayList<String>();
        boolean overwrite = true;
        /**指向逆向工程配置文件*/
        File configFile = new File("D:\\first-app-demo\\second-demo-web-
restfulcrud\\src\\main\\resources\\generator" +
            "\\generatorConfig.xml");
        ConfigurationParser parser = new ConfigurationParser(warnings);
        Configuration config = parser.parseConfiguration(configFile);
        DefaultShellCallback callback = new DefaultShellCallback(overwrite);
        MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config,
            callback, warnings);
    }
}

```

```

        myBatisGenerator.generate(null);
    }

    public static void main(String[] args) throws Exception {
        try {
            Generator generatorSqlmap = new Generator();
            generatorSqlmap.generator();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

最后，运行其中main方法即可生成POJO实体类、Mapper接口、Xml映射文件。

PageHelper 分页

Maven依赖

```

<dependency>
    <groupId>com.github.pagehelper</groupId>
    <artifactId>pagehelper-spring-boot-starter</artifactId>
    <version>1.3.0</version>
</dependency>

```

操作方法

1.配置文件

```

pagehelper:
    # 指定分页插件使用哪种方言
    helperDialect: mysql
    # 分页合理化，默认值为false。当该参数设置为 true 时，pageNum<=0 时会查询第一页,
    pageNum>pages, 会查询最后一页。
    reasonable: true
    ...

```

2.编写对应需要分页controller层方法

```

/**
 * 会员管理
 */
@RequestMapping("/manageMember")
public String manageMember(@RequestParam(defaultValue = "1") int pageNum,
                           @RequestParam(defaultValue = "10") int pageSize,
                           Model model){
    // 设置查询分页大小
    PageHelper.startPage(pageNum,pageSize);
    // 装入PageInfo容器中
    PageInfo pageInfo=new PageInfo(adminMemberService.selectAllUser());
    model.addAttribute("pageInfo",pageInfo);
    return "adminMemberManage";
}

```

```
public PageInfoBO queryOrderData(OrderQueryRequest orderQueryRequest) {  
    //如果是super则查询所有，否则上查下  
    UserDTO user = UserUtil.getUser();  
    if (user.isSuperAdmin()) {  
        orderQueryRequest.setDeptId(null);  
    } else {  
        orderQueryRequest.setDeptId(user.getDeptId());  
    }  
  
    PageHelper.startPage(orderQueryRequest.getPage(), orderQueryRequest.getLimit());  
    //查询数据  
    List<SysorderDTO> orders =  
    orderMapper.queryOrderData(orderQueryRequest);  
    if (!orders.isEmpty()) {  
        PageInfo<SysOrderDTO> pageInfo = new PageInfo<>(orders);  
        // pageInfo.getTotal : 获取总条数  
        //pageInfo.getList : 获取数据  
        return new PageInfoBO(pageInfo.getTotal(), pageInfo.getList());  
    }  
    return null;  
}  
}
```

Redis基础学习

什么是NoSql

NoSql=not only Sql

泛指非关系型数据库,web2.0互联网诞生,传统的关系型数据库难以对付Web2.0时代.尤其超大规模的高并发社区.

关系型数据库:表格,行列记录.(POI)

NoSQL的特点:

- 1.方便扩展(数据之间没有关系)
- 2.大数据量高性能(Redis一秒写8万次,读取11万)

Redis是开源的一个内存数据库, 数据保存在内存中, 但是我们都知道内存的数据变化是很快的, 也容易发生丢失。

幸好Redis还为我们提供了持久化的机制, 分别是RDB(Redis DataBase)和AOF(Append Only File)。

Linu下安装

一.将安装文件包复制到 opt 文件夹下解压

二.安装C++环境包

```
yum install gcc-c++
```

三.进入redis文件夹 运行 make 命令 ,然后在运行 make install 命令.

Redis默认安装路径 'usr/local/bin/'

四.将Redis配置文件复制到Redis默认安装路径usr/local/bin/下. 之后就使用这个作为配置文件启动.

五.Redis默认不是后台启动的.需要修改配置文件

```
daemonize yes
```

六.启动Redis服务.

七.查看redis服务打开端口状态

```
ps -ef|grep redis
```

服务器

打开服务器

```
redis-server.exe (/.../redis.conf)
```

访问服务器

```
redis-cli.exe -h 服务器IP -p 端口号(默认6379)
```

基础知识

Redis默认有16个数据库

默认使用第0个数据库. 可以使用select进行切换数据库

```
select (2) # 切换数据库  
dbsize # 查看数据库大小  
flushdb # 清空当前数据库  
flushall # 清空全部数据库  
  
exists (key) #判断是否存在key,返回0,1  
exists (key) (数据库名) #移动key到指定数据库中,返回0,1  
  
expire (key) (秒数) #设置key的过期时间,返回0,1 key过期后会自动为null
```

Redis是单线程(Redis6.0支持多线程).Redis是基于 内存操作,CPU不是Redis的性能瓶颈,Redis的瓶颈是根据机器的内存和网络带宽.

Redis将所有的数据放在内存中的,所有说使用单线程性能最高.

数据类型

Redis支持5类数据类型:

String (字符串)

hash (散列)

list (列表)

set (集合)

zset (有序集合)

1.String

string 是 redis 最基本的类型，一个 key 对应一个 value。

string 类型是二进制安全的。意思是 redis 的 string 可以包含任何数据。比如jpg图片或者序列化的对象。

string 类型的单个值最大能存储 512MB。

```
set user:1:xx {name=aa,age=32,email=231432@qq.com}
```

SET命令

设置指定 key 的值

```
SET -key- -value-
```

GET命令

获取指定 key 的值

```
GET -key-
```

GETSET命令

将给定 key 的值设为 value，并返回 key 的旧值

```
GETSET -key- -value-
```

KEYS命令

查询当前数据库所有key

```
KEYS *
```

STRLEN命令

返回 key 所储存的字符串值的长度

```
STRLEN -key-
```

MSET命令

同时设置一个或多个 key-value 对

```
MSET -key1- -value1- -key2- -value2- ...
```

2.Hash

hash 哈希 是一个键值(key=>value)对集合， string 类型的 field 和 value 的映射表。

hash 特别适合用于存储对象。

每个 hash 可以存储 $2^{32} - 1$ 键值对 (40多亿) 。

HMSET命令

为指定key设置hash表

```
HMSET -key- -field1- -value1- -field2- -value2- ...
```

HGETALL命令

遍历整个key的hash表内容

```
HGETALL -key-
```

HGET命令

获取某个key中指定hash表内容

```
HGET -key- -field-
```

HDEL命令

删除整个key中的某个field以及对应value

```
HDEL -key- -field-
```

HLEN命令

获取hash表中字段的数量

```
HLEN -key-
```

3.List

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

LPUSH命令

将一个或多个值插入到列表头部。

```
LPUSH -key- -value1- -value2- ...
```

LRANGE命令

根据指定长度获取对应list中值

```
LRANGE key -start index- -stop index-
```

LSET命令

通过索引来设置元素的值。

```
LSET -key- -index- -value-
```

LLEN命令

获取列表长度。

```
LLEN -key-
```

LIINDEX命令

通过索引获取列表中的元素。

```
LIINDEX -key- -index-
```

LPOP/RPOP命令

删除最后一个元素/删除第一个元素

```
LPOP -key-
RPOP -key-
```

RPOPLPUSH命令

删除第一个元素并将其添加到另一个List列表中

```
RPOPLPUSH -key- -other key-
```

4.Set

Set 是 string 类型的无序集合。

Set不允许内容重复。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是 0或者1

SADD命令

添加一个 string 元素到 key 对应的 set 集合中，成功返回 1，如果元素已经在集合中返回 0。

```
SADD -key- -value1- -value2- ...
```

SMEMBER命令

返回集合中的所有的成员。

```
SMEMBER -key-
```

SCARD命令

命令返回集合中元素的数量。

```
SCARD -key-
```

SISMEMBER命令

判断成员元素是否是集合的成员。返回 0或者1

```
SISMEMBER -key- -value-
```

SREM命令

移除set集合中的指定元素

```
SREM -key- -value-
```

SRANDMEMBER命令

在指定集合中随机获取指定个数的成员

```
SRANDMEMBER -key- -size(默认为1)-
```

对比命令

SDIFF命令

展示出两集合中的差集

```
SDIFF -key1- -key2-
```

SINTER命令

展示出两集合中的交集

```
SINTER -key1- -key2-
```

SUNION命令

展示出两集合中的并集

```
SUNION -key1- -key2-
```

5.Zset

有序集合和集合一样也是 string 类型元素的集合,且不允许重复的成员。

不同的是每个元素都会关联一个 double 类型的分数。

redis 正是通过分数来为集合中的成员进行从小到大的排序。

有序集合的成员是唯一的,但分数(score)却可以重复。

分数值可以是整数值或双精度浮点数。

ZADD命令

向有序集合添加一个或多个成员,或者更新已存在成员的分数

```
ZADD -key- -score1- -value1- -score2- -value2- ...
```

ZSCARD命令

计算集合中元素的数量。

```
ZSCARD -key-
```

ZRANGE/ZREVRANGE命令

通过索引区间返回有序集合指定区间内的成员

```
ZRANGE/ZREVRANGE -key- -minScore- -maxScore- (-withscores-)
```

三种特殊数据类型

Geospatial - 地理位置

Geospatial 可以推算地理位置信息,两地之间距离,方圆几里的人.

GEOSPATIAL的底层原理是ZSET. 所以也可以使用ZSET的命令

Geoadd - 添加地理位置

```
# 两极无法添加,通常我们使用java程序一次性导入
Geoadd -表名- -东经- -北纬- -列名- [-东经- -北纬- -列名-]...
Geoadd china:city 106.73 31.86 bazhong
```

Geopos - 查询地理位置

```
# 获取指定的城市的经纬度
geopos -表名- -列名-
geopos china:city shanghai
```

Geodist - 计算地理之间距离

单位:

- **M** 表示 单位为米(默认)
- **KM** 表示 单位为千米
- **MI**表示 单位为英里
- **FT** 表示 单位为英寸

```
Geodist -表名- -列名1- -列名2- [单位]
Geodist china:city chengdu bazhong km
```

Georadius - 寻找附近(通过经纬度)

```
Georadius -表名- -经度- -纬度- -半径距离- [单位] [withcoord/withdist] [count 数量]
Georadius china:city 110 30 500 km count 3
```

Georadiusbymember - 寻找附近(通过指定元素)

```
Georadiusbymember -表名- -列名- -半径距离- [单位]
Georadiusbymember china:city bazhong 1000 km
```

Hyperloglog数据结构 - 基数统计

Redis Hyperloglog - 基数统计.

基数:不可重复的元素,可以接受误差.

网页的UV(一个人访问网站多次,依旧算一个人的数据量)

传统的方式,set保存用户的id,然后就可以统计set中的元素数量作为访问标准,会消耗大量的资源,比较麻烦.

Hyperloglog的优点:

- 占用的内存固定,2^64大小元素,只需要12KB内存存储.

```
PFADD mykey a b c d e f g h i j k # 创建第一组元素
(integer) 1
PFCOUNT mykey # 统计第一组元素基数量
(integer) 11
```

```

PFADD mykey2 a b c d p q l # 创建第二组元素
(integer) 1
PFCOUNT mykey2 # 统计第二组元素基数数量
(integer) 7

# PFMERGE(合并log) -newkey- -key1- -key2-
PFMERGE mykey3 mykey mykey2 # 合并第一组元素和第二组元素
OK
PFCOUNT mykey3 # 统计第三组元素基数数量
(integer) 15

```

如果允许容错,那么一定可以使用Hyperloglog. 但如果不允许容错,那使用SET或者自己的数据类型即可 .

Bitmaps - 位图(位存储)

统计用户信息,活跃,不活跃,登录,未登录,打卡,未打卡...两种状态所有Bitmaps!

Bitmaps数据结构,二进制记录,只有0,1 两种状态.

365天的数据状态=365 bit 1字节=8bit 46个字节即可.

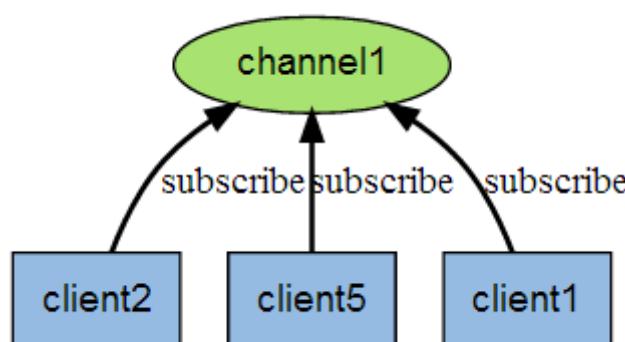
```

# Setbit -表名- -时间- -状态(0\1)-
127.0.0.1:6379> setbit sign 0 1 # 周一 打卡
(integer) 0
127.0.0.1:6379> setbit sign 1 0 # 周二 未打卡
(integer) 0
127.0.0.1:6379> setbit sign 2 1 # 周三 打卡
(integer) 0
127.0.0.1:6379> setbit sign 3 1 # 周四 打卡
(integer) 0
127.0.0.1:6379> setbit sign 4 0 # 周五 未打卡
(integer) 0
127.0.0.1:6379> getbit sign 4 # 查询周五打卡状态
(integer) 0
127.0.0.1:6379> bitcount sign # 查询所有打卡次数(只统计1)
(integer) 3

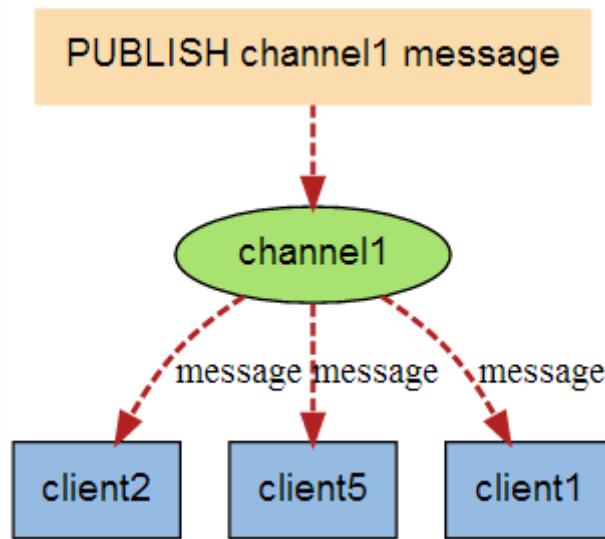
```

发布订阅

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



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



SUBSCRIBE命令

订阅给定的一个或多个频道的信息。

```
Subscribe -channel1- -channel2- ....
```

PUBLISH命令

将信息推送到指定的频道。

```
Publish -channel- -message-
```

UNSUBSCRIBE命令

退订指定频道。

```
Unsubscribe -channel1- -channel2-
```

事务

Redis的事务本质:一组命令的集合.一个事务中所有的命令都会被序列化,在事务执行过程中,会按照顺序执行!

Redis单条命令保证原子性,但Redis的事务不保证原子性!

-Redis事务没有隔离级别的概念-

事务执行步骤

- 开启事务(multi)
- 命令入队(...)
- 执行事务(exec) / 撤销事务(discard)

事务监控

悲观锁:无论什么操作都认为会存在问题,随时都会上锁

乐观锁:无论什么操作都不会认为会存在问题,随时都不上锁.更新数据时会判断一下,在此期间是否修改过数据,

获取Version.更新时对比Version,没变则更新成功,否则失败.

测试多线程修改值,使用watch可以当作redis的乐观锁的操作.unwatch取消乐观锁.

Redis的持久化

面试和工作,持久化都是重点!

RDB(Redis DataBase)

指定的时间间隔内将内存中的数据集快照 写入磁盘中,恢复时就将快照读取到内存中.

Redis会单独创建一个(fork)子进程来进行持久化,先将数据写入一个临时文件中,再用这个临时文件替换上场持久化好的文件.

整个过程主进程不会进行任何IO操作的,这就确保了极高的性能.

RDB缺点:最后一次持久化后的数据宕机可能丢失.

Redis默认持久化就是RDB,一般情况下不需要修改这个配置!

RDB保存的文件是dump.rdb 在工作环境我们会将该文件进行备份.

```
# The filename where to dump the DB
dbfilename dump.rdb
```

RDB触发机制:

1.save的规则满足情况下触发RDB持久化规则.

2.执行了flushall命令,也会触发RDB持久化规则.

3.退出Redis,也会触发RDB持久化规则

如何恢复RDB文件:

1.将需要恢复的RDB文件放置到redis启动的文件目录下就可以,redis启动时就会自动检查dump.rdb文件并恢复其中的数据.

2.查看需要存放的位置

```
127.0.0.1:6379> config get dir
1) "dir"
2) "D:\\redis" # 如果在这个目录下查找dump.rdb,启动就会自动恢复其中的数据.
```

RDB优点:

1.适合大规模的数据恢复!

2.对数据完整性要求不高!

RDB缺点:

1.需要一定时间间隔进行操作.最后一次持久化后的数据宕机可能丢失.

2.fork进程的时候,会占用一定的内容空间.

AOF(Append Only File)

AOF:将我们的所有命令都记录下来(history),恢复时就把history文件全部执行一次!

原理:以日志的形式记录每个操作,将Redis执行过的每个指令记录下来(读取操作不记录),只需增加文件,不许改写文件.Redis重启后根据日志文件进行全部执行以达到恢复内容.

AOF默认不开启.需要手动在config配置文件配置.

```
appendonly yes # 开启AOF
```

AOF保存的是appendonly.aof文件(在config配置文件配置)

```
appendfilename "appendonly.aof" # 保存为appendonly.aof文件
```

AOF持久化策略(在config配置文件配置)

```
appendfsync everysec # AOF每一秒写一次  
no-appendfsync-on-rewrite no # AOF不重写,保证数据安全性
```

AOF默认文件无限追加,文件会越来越大.

如果AOF文件存在错误,Redis则不能正常启动服务.

Redis给我们通过了redis-check-aof工具.可以对错误的AOF文件进行恢复.但被破坏的数据会被丢失.

```
redis-check-aof --fix appendonly.aof # 修复AOF文件
```

AOF优点:

1.每一步修改都会被同步,文件完整性更好!

2.默认每秒同步一次

3.从不同步,效率最高.

AOF缺点:

1.相对于数据文件来说,AOF远远大于RDB,修复速度比RDB慢.

2.AOF运行效率比RDB慢.所以Redis默认配置RDB持久化,而非AOF持久化.

Redis主从复制

概率:

指一台Redis服务器的数据,复制到其他Redis服务器. 前者被称为主节点(master/leader),后者称为从节点(slave/follower).

主从复制,读写分离,数据的复制是单向的,只能从主节点到从节点. Master以写为主,Slave以读为主.

主从复制作用主要包括:

- 1.数据冗余:主从复制实现数据的热备份,是持久化以外的数据冗余方式.
- 2.故障恢复:当主节点出现的问题时.可以由从节点通过服务,实现快速的故障恢复.
- 3.负载均衡:配合读写分离,分担服务器负担.
- 4.高可用(集群)基石:主从复制还是哨兵模式和集群能够实施的基础.

在工程项目中一般要配置三个Redis服务器(一主二从),来保证基本使用.

== 单台Redis最大使用内存不应超过20GB ==

主要是公司中,主从复制就必须要使用,因为真实项目不可能单机使用Redis!

配置环境

```
127.0.0.1:6379> info replication #查看该服务器信息
# Replication
role:master # 主节点
connected_slaves:0 #没有从机,单机
master_replid:b5bc2229c79b87ab1925372736a6684ff1a0332c
master_replid2:0000000000000000000000000000000000000000000000000000000000000000
master_repl_offset:0
second_repl_offset:-1
replog_size:1048576
replog_file_size:1048576
replog_histlen:0
```

- 1.复制多个config配置文件,以启动多个服务器
- 2.修改各个配置文件的port端口号(为了本机测试不冲突,不易相同)
- 3.修改配置文件中log文件名(为了本机测试不冲突,名称不易相同)
- 4.修改配置文件中pid名称(为了本机测试不冲突,名称不易相同)
- 5.修改配置文件中RDB的dump.rdb文件名(为了本机测试不冲突,名称不易相同)
- 6.利用配置文件启动多个Redis服务器.

默认情况下,每台Redis服务器都是主节点;一般情况下只配置从节点,不用配置主节点.

```
# slaveof 服务器IP 端口号
127.0.0.1:6380> slaveof 127.0.0.1 6379 # 配置127.0.0.1 6379接口下的服务器为主节点
OK
```

注:真实的主从配置是在Config配置文件中配置,使用slaveof命令只是暂时的.

```
# slaveof <masterip> <masterport> # 使用时去掉前'#'即可
```

==主从复制细节==

主节点可以进行写入操作,从节点只能读取操作.

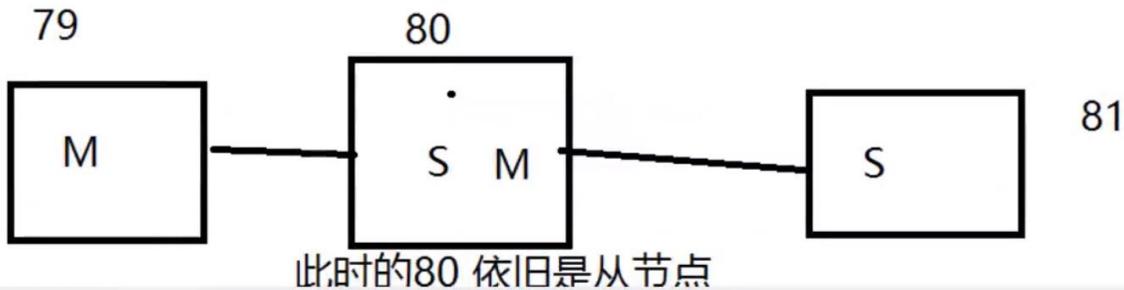
主机断开,从机依然连接到主机,依旧可以进行读取操作.

如果使用命令行配置主从关系,从机打开服务器将会断开主从关系,恢复为主节点.此时服务器的数据将为断开前主节点的数据.

恢复主从关系其主节点将会发送sync同步命令,其主节点数据就可以继续读写.

层层链路

当一个从节点 其 主节点 为 另一个主节点的从节点 时,后者依旧为从节点.



删除主从关系:

```
127.0.0.1:6380> slaveof no one # 删除之前设置的主从关系,恢复为主节点
OK
```

哨兵模式

自动切换主从关系的模式.

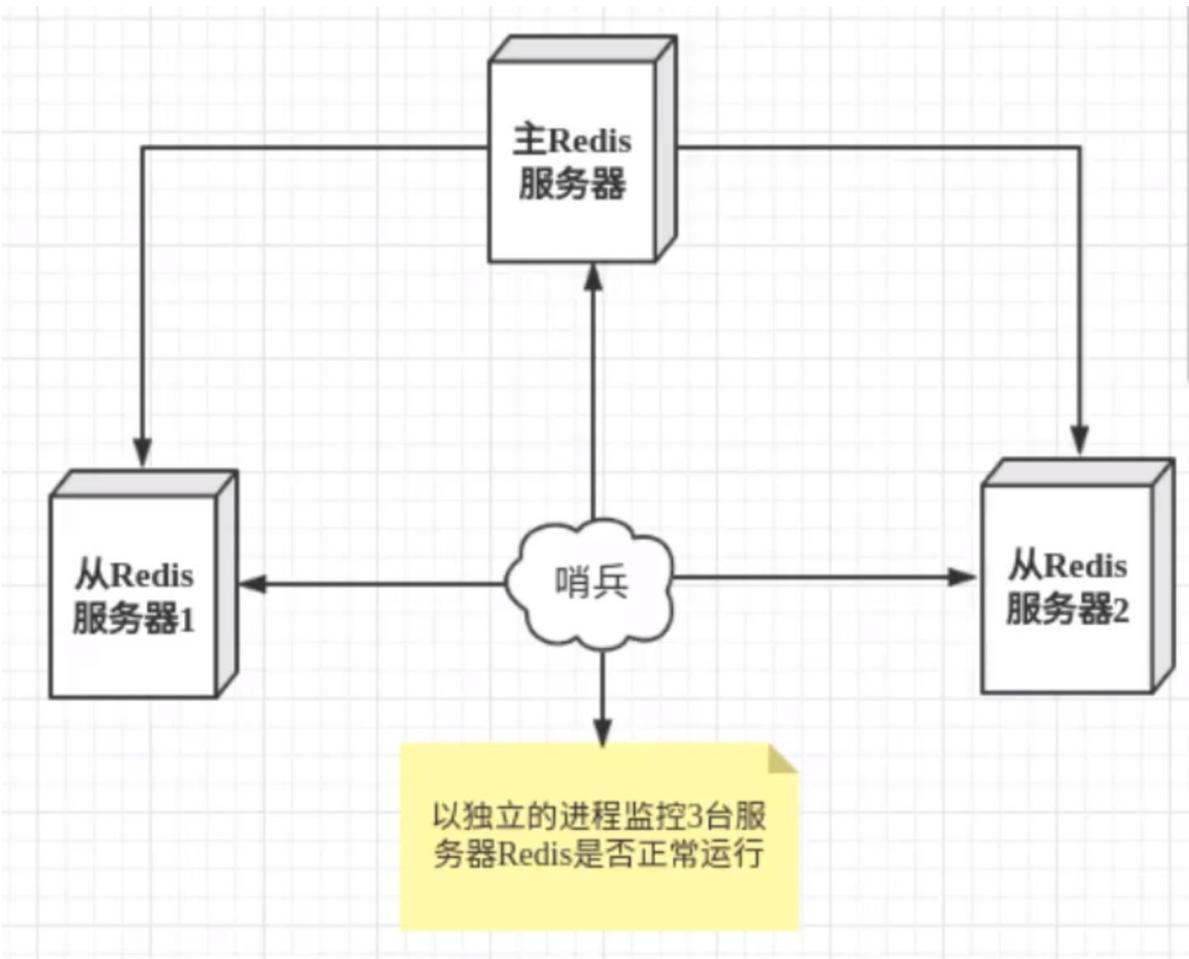
主从切换的模式方法是:当主服务器宕机后,需要手动把一台服务器切换为主服务器 ,费时.

而Redis2.8后支持Sentinel(哨兵)框架解决该问题.

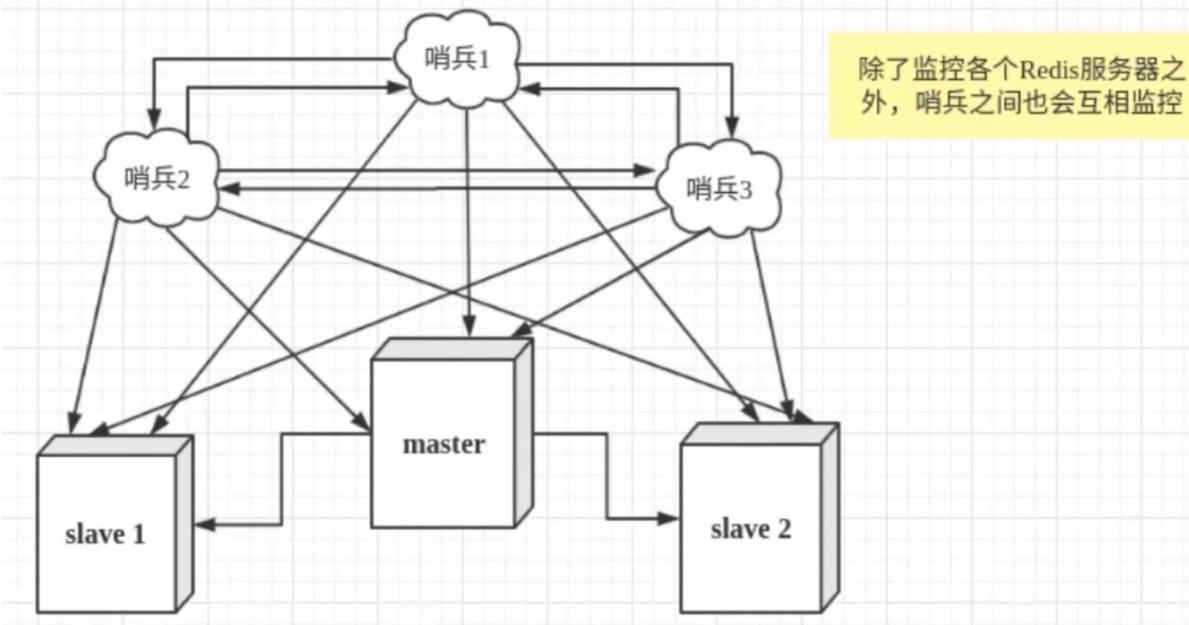
哨兵模式能够后台监控主机是否故障,如果故障了会根据投票数**自动将从库转换为主库**.

哨兵模式是一个特殊的模式,是一个独立的进程,需要独立运行.

其原理是哨兵通过发送命令,等待Redis服务器响应,从而监控多个Redis实例.



然而一个哨兵进程监控Redis服务器,也有可能出现问题,为此,可以使用多个哨兵进行监控. 各个哨兵之间还会进行监控,从而形成多哨兵模式.



配置哨兵模式:

1. 配置哨兵配置文件 sentinel.conf

----- 简易配置 -----

```
# sentinel monitor 被监控的名称 主机地址 端口 1(投票制)
sentinel monitor myredis 127.0.0.1 6379 1
```

```

----- 全部默认 -----
# 这个是Redis6379配置内容，其他文件同理新增然后改一下端口即可，26380
#当前Sentinel服务运行的端口
protected-mode no
port 26381
# 哨兵监听的主服务器 后面的1表示主机挂掉以后进行投票，只需要1票就可以从机变主机
sentinel monitor mymaster 127.0.0.1 6379 2
# 3s内mymaster无响应，则认为mymaster宕机了 默认为30s
sentinel down-after-milliseconds mymaster 3000
#如果10秒后,mysater仍没启动过来，则启动failover
sentinel failover-timeout mymaster 10000
# 执行故障转移时，最多有1个从服务器同时对新的主服务器进行同步
sentinel parallel-syncs mymaster 1
# 设置哨兵sentinel 连接主从的密码 注意必须为主从设置一样的验证密码，没有的话不用设置
sentinel auth-pass mymaster 123456

```

2.启动哨兵.

```

# windows下启动哨兵模式
redis-server sentinel.conf --sentinel
# Linux下启动哨兵模式
redis-sentinel sentinel.conf

```

如果此时Master节点断开了,这是就会在其中从机中投票选出新主机(Master节点).

可以从哨兵日志查看出新主节点服务器.

```

[16196] 03 Apr 13:15:00.092 * +failover-state-wait-promotion slave
127.0.0.1:6381 127.0.0.1 6381 @ myredis 127.0.0.1 6379 # 6379端口主节点断开
[16196] 03 Apr 13:15:01.079 # +promoted-slave slave 127.0.0.1:6381 127.0.0.1
6381 @ myredis 127.0.0.1 6379
[16196] 03 Apr 13:15:01.079 # +failover-state-reconf-slaves master myredis
127.0.0.1 6379
[16196] 03 Apr 13:15:01.150 * +slave-reconf-sent slave 127.0.0.1:6380 127.0.0.1
6380 @ myredis 127.0.0.1 6379
[16196] 03 Apr 13:15:02.140 * +slave-reconf-inprog slave 127.0.0.1:6380
127.0.0.1 6380 @ myredis 127.0.0.1 6379
[16196] 03 Apr 13:15:02.140 * +slave-reconf-done slave 127.0.0.1:6380 127.0.0.1
6380 @ myredis 127.0.0.1 6379
[16196] 03 Apr 13:15:02.211 # +failover-end master myredis 127.0.0.1 6379
[16196] 03 Apr 13:15:02.211 # +switch-master myredis 127.0.0.1 6379 127.0.0.1
6381 # 主节点切换为6381端口服务器
[16196] 03 Apr 13:15:02.212 * +slave slave 127.0.0.1:6380 127.0.0.1 6380 @
myredis 127.0.0.1 6381
[16196] 03 Apr 13:15:02.212 * +slave slave 127.0.0.1:6379 127.0.0.1 6379 @
myredis 127.0.0.1 6381

```

如果原主节点服务器重启后,其原主节点服务器只能当做新主节点服务器的从机.

哨兵模式优点:

- 1.哨兵集群,基于主从复制的模式,所有的主从配置优点它全都有.
- 2.主从可以切换,故障可以转移,系统的可用性更好.
- 3.哨兵模式就是主从模式的升级,手动到自动.

哨兵模式缺点:

- 1.Redis不好在线扩容,集群容量一旦到达上限,在线扩容就十分麻烦.
- 2.实现哨兵模式的配置很麻烦.

Jedis

Jedis是Redis官方推荐的Java连接开发工具.使用Java操作Redis的中间键.

Maven依赖

```
<!--导入jedis包-->
<!-- https://mvnrepository.com/artifact/redis.clients/jedis -->
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>3.3.0</version>
</dependency>
<!--导入fastjson包-->
<!-- https://mvnrepository.com/artifact/com.alibaba/fastjson -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.2.75</version>
</dependency>
```

编码测试

- 连接数据库

```
//1.创建一个Jedis对象
Jedis jedis = new Jedis("127.0.0.1", 6379);
System.out.println(jedis.ping());
```

- 操作命令

```
JSONObject jsonObject=new JSONObject();
jsonObject.put("hello","world");
jsonObject.put("name","zssaer");
//开启redis事务
Transaction multi=jedis.multi();
String jsonString = jsonObject.toJSONString();

try {
    multi.set("user1",jsonString);
    multi.exec();
} catch (Exception e) {
    multi.discard();
    e.printStackTrace();
```

```
    }finally {
        System.out.println(jedis.get("user1"));
        jedis.close();
    }
}
```

- 断开连接

```
jedis.close(); //关闭连接
```

SpringBoot+Redis

在SpringBoot2.x后,原来使用的Jedis被替换成lettuce.

Jedis:底层采用的是直连方式,多个线程操作是不安全的. 多线程连接操作需要使用jedis-pool连接池!更像BIO模式

lettuce:采用netty,实例可以在多个线程中共享,不存在线程不安全的情况.可以减少线程数量.更像NIO模式.

Maven依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
    <version>2.3.0.RELEASE</version>
</dependency>

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-annotations</artifactId>
    </dependency>
</dependency>
```

RedisAutoConfiguration源码分析:

```
@Bean
@ConditionalOnMissingBean(name = {"redisTemplate"}) //我们可以自己定义个
redisTemplate来替换这个默认的!
@ConditionalOnSingleCandidate(RedisConnectionFactory.class)
public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory
redisConnectionFactory) {
    //默认的RedisTemplate 没有过多设置,redis对象都是需要序列化!
    //两个泛型都是Object, Object类型,我们后使用需要强制转换<String, Object>
    RedisTemplate<Object, Object> template = new RedisTemplate();
    template.setConnectionFactory(redisConnectionFactory);
    return template;
}

@Bean
@ConditionalOnMissingBean
@ConditionalOnSingleCandidate(RedisConnectionFactory.class)
```

```
public StringRedisTemplate stringRedisTemplate(RedisConnectionFactory  
redisConnectionFactory) {  
    StringRedisTemplate template = new StringRedisTemplate();  
    template.setConnectionFactory(redisConnectionFactory);  
    return template;  
}
```

RedisTemplate使用:

```
//opsForValue 操作String  
//opsForList 操作List  
//opsForSet 操作Set  
//opsForHash 操作Hash  
//opsForZset 操作Zset  
redisTemplate.opsForValue().set("mykey", "myvalue");  
System.out.println(redisTemplate.opsForValue().get("mykey"));
```

配置方法

1. 配置properties文件(yaml文件)中的redis环境参数

```
# Redis数据库索引（默认为0）  
spring.redis.database=0  
# Redis服务器地址  
spring.redis.host=localhost  
# Redis服务器连接端口  
spring.redis.port=6379  
# Redis服务器连接密码（默认为空）  
spring.redis.password=123456  
#连接池最大连接数（使用负值表示没有限制）  
spring.redis.jedis.pool.max-active=8  
# 连接池最大阻塞等待时间（使用负值表示没有限制）  
spring.redis.jedis.pool.max-wait=-1  
# 连接池中的最大空闲连接  
spring.redis.jedis.pool.max-idle=8  
# 连接池中的最小空闲连接  
spring.redis.jedis.pool.min-idle=0  
# 连接超时时间（毫秒）  
spring.redis.timeout=300
```

2. 配置RedisConfig类

以下配置类直接copy使用即可

```
/**  
 * Redis配置类  
 */  
@Configuration  
@EnableCaching //开启注解  
public class RedisConfig extends CachingConfigurerSupport {  
    /**  
     * retemplate相关配置  
     * @param factory  
     * @return  
     */
```

```
@Bean
public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory factory) {

    RedisTemplate<String, Object> template = new RedisTemplate<>();
    // 配置连接工厂
    template.setConnectionFactory(factory);

    // 使用Jackson2JsonRedisSerializer来序列化和反序列化redis的value值（默认使用JDK的序列化方式）
    Jackson2JsonRedisSerializer jacksonSeial = new Jackson2JsonRedisSerializer(Object.class);

    ObjectMapper om = new ObjectMapper();
    // 指定要序列化的域，field,get和set,以及修饰符范围，ANY是都有包括private和public
    om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
    // 指定序列化输入的类型，类必须是非final修饰的，final修饰的类，比如String, Integer等会跑出异常
    om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
    jacksonSeial.setObjectMapper(om);

    // 值采用json序列化
    template.setValueSerializer(jacksonSeial);
    // 使用StringRedisSerializer来序列化和反序列化redis的key值
    template.setKeySerializer(new StringRedisSerializer());

    // 设置hash key 和value序列化模式
    template.setHashKeySerializer(new StringRedisSerializer());
    template.setHashValueSerializer(jacksonSeial);
    template.afterPropertiesSet();

    return template;
}

/**
 * 对hash类型的数据操作
 *
 * @param redisTemplate
 * @return
 */
@Bean
public HashOperations<String, String, Object>
hashOperations(RedisTemplate<String, Object> redisTemplate) {
    return redisTemplate.opsForHash();
}

/**
 * 对redis字符串类型数据操作
 *
 * @param redisTemplate
 * @return
 */
@Bean
public ValueOperations<String, Object> valueOperations(RedisTemplate<String,
Object> redisTemplate) {
    return redisTemplate.opsForValue();
}
```

```

    /**
     * 对链表类型的数据操作
     *
     * @param redisTemplate
     * @return
     */
    @Bean
    public ListOperations<String, Object> listOperations(RedisTemplate<String,
Object> redisTemplate) {
        return redisTemplate.opsForList();
    }

    /**
     * 对无序集合类型的数据操作
     *
     * @param redisTemplate
     * @return
     */
    @Bean
    public SetOperations<String, Object> setOperations(RedisTemplate<String,
Object> redisTemplate) {
        return redisTemplate.opsForSet();
    }

    /**
     * 对有序集合类型的数据操作
     *
     * @param redisTemplate
     * @return
     */
    @Bean
    public ZSetOperations<String, Object> zSetOperations(RedisTemplate<String,
Object> redisTemplate) {
        return redisTemplate.opsForZSet();
    }
}

```

3.配置RedisUtil类

以下工具类直接copy使用即可

```

    /**
     * TODO(控制RedisTemplate封装)
     */
    @Component
    public class Redisutil {

        @Autowired
        private RedisTemplate<String, Object> redisTemplate;

        public Redisutil(RedisTemplate<String, Object> redisTemplate) {
            this.redisTemplate = redisTemplate;
        }

        /**
         * 指定缓存失效时间
         * @param key 键
         */

```

```

        * @param time 时间(秒)
        * @return
        */
    public boolean expire(String key, long time){
        try {
            if(time>0){
                redisTemplate.expire(key, time, TimeUnit.SECONDS);
            }
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 根据key 获取过期时间
     * @param key 键 不能为null
     * @return 时间(秒) 返回0代表为永久有效
     */
    public long getExpire(String key){
        return redisTemplate.getExpire(key,TimeUnit.SECONDS);
    }

    /**
     * 判断key是否存在
     * @param key 键
     * @return true 存在 false不存在
     */
    public boolean hasKey(String key){
        try {
            return redisTemplate.hasKey(key);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 删除缓存
     * @param key 可以传一个值 或多个
     */
    @SuppressWarnings("unchecked")
    public void del(String ... key){
        if(key!=null&&key.length>0){
            if(key.length==1){
                redisTemplate.delete(key[0]);
            }else{
                redisTemplate.delete(CollectionUtils.arrayToList(key));
            }
        }
    }

    //=====
    /**
     * 普通缓存获取
     * @param key 键
     * @return 值

```

```
/*
public Object get(String key){
    return key==null?null:redisTemplate.opsForValue().get(key);
}

/***
 * 普通缓存放入
 * @param key 键
 * @param value 值
 * @return true成功 false失败
 */
public boolean set(String key, Object value) {
    try {
        redisTemplate.opsForValue().set(key, value);
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/***
 * 普通缓存放入并设置时间
 * @param key 键
 * @param value 值
 * @param time 时间(秒) time要大于0 如果time小于等于0 将设置无限期
 * @return true成功 false 失败
 */
public boolean set(String key, Object value, long time){
    try {
        if(time>0){
            redisTemplate.opsForValue().set(key, value, time,
TimeUnit.SECONDS);
        }else{
            set(key, value);
        }
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/***
 * 递增
 * @param key 键
 * @param delta 要增加几(大于0)
 * @return
 */
public Long incr(String key, long delta){
    if(delta<0){
        throw new RuntimeException("递增因子必须大于0");
    }
    return redisTemplate.opsForValue().increment(key, delta);
}

/***
 * 递减
 */
```

```
* @param key 键
* @param delta 要减少几(小于0)
* @return
*/
public long decr(String key, long delta){
    if(delta<0){
        throw new RuntimeException("递减因子必须大于0");
    }
    return redisTemplate.opsForValue().increment(key, -delta);
}

//=====Map=====
/***
 * HashGet
 * @param key 键 不能为null
 * @param item 项 不能为null
 * @return 值
*/
public Object hget(String key,String item){
    return redisTemplate.opsForHash().get(key, item);
}

/***
 * 获取hashKey对应的所有键值
 * @param key 键
 * @return 对应的多个键值
*/
public Map<Object, Object> hmget(String key){
    return redisTemplate.opsForHash().entries(key);
}

/***
 * HashSet
 * @param key 键
 * @param map 对应多个键值
 * @return true 成功 false 失败
*/
public boolean hmset(String key, Map<String, Object> map){
    try {
        redisTemplate.opsForHash().putAll(key, map);
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/***
 * HashSet 并设置时间
 * @param key 键
 * @param map 对应多个键值
 * @param time 时间(秒)
 * @return true成功 false失败
*/
public boolean hmset(String key, Map<String, Object> map, long time){
    try {
        redisTemplate.opsForHash().putAll(key, map);
        if(time>0){
```

```
        expire(key, time);
    }
    return true;
} catch (Exception e) {
    e.printStackTrace();
    return false;
}
}

/***
 * 向一张hash表中放入数据,如果不存在将创建
 * @param key 键
 * @param item 项
 * @param value 值
 * @return true 成功 false失败
 */
public boolean hset(String key,String item, Object value) {
    try {
        redisTemplate.opsForHash().put(key, item, value);
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/***
 * 向一张hash表中放入数据,如果不存在将创建
 * @param key 键
 * @param item 项
 * @param value 值
 * @param time 时间(秒) 注意:如果已存在的hash表有时间,这里将会替换原有的时间
 * @return true 成功 false失败
 */
public boolean hset(String key, String item, Object value, long time) {
    try {
        redisTemplate.opsForHash().put(key, item, value);
        if(time>0){
            expire(key, time);
        }
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/***
 * 删除hash表中的值
 * @param key 键 不能为null
 * @param item 项 可以使多个 不能为null
 */
public void hdel(String key, Object... item){
    redisTemplate.opsForHash().delete(key, item);
}

/***
 * 判断hash表中是否有该项的值
 */
```

```

    * @param key 键 不能为null
    * @param item 项 不能为null
    * @return true 存在 false不存在
    */
    public boolean hHasKey(String key, String item){
        return redisTemplate.opsForHash().hasKey(key, item);
    }

    /**
     * hash递增 如果不存在,就会创建一个 并把新增后的值返回
     * @param key 键
     * @param item 项
     * @param by 要增加几(大于0)
     * @return
     */
    public double hincr(String key, String item,double by){
        return redisTemplate.opsForHash().increment(key, item, by);
    }

    /**
     * hash递减
     * @param key 键
     * @param item 项
     * @param by 要减少记(小于0)
     * @return
     */
    public double hdecr(String key, String item,double by){
        return redisTemplate.opsForHash().increment(key, item,-by);
    }

//=====set=====
    /**
     * 根据key获取Set中的所有值
     * @param key 键
     * @return
     */
    public Set<Object> sGet(String key){
        try {
            return redisTemplate.opsForSet().members(key);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }

    /**
     * 根据value从一个set中查询,是否存在
     * @param key 键
     * @param value 值
     * @return true 存在 false不存在
     */
    public boolean sHasKey(String key,object value){
        try {
            return redisTemplate.opsForSet().isMember(key, value);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

```

```
}

/**
 * 将数据放入set缓存
 * @param key 键
 * @param values 值 可以是多个
 * @return 成功个数
 */
public long sSet(String key, Object...values) {
    try {
        return redisTemplate.opsForSet().add(key, values);
    } catch (Exception e) {
        e.printStackTrace();
        return 0;
    }
}

/**
 * 将set数据放入缓存
 * @param key 键
 * @param time 时间(秒)
 * @param values 值 可以是多个
 * @return 成功个数
 */
public long sSetAndTime(String key,long time,object...values) {
    try {
        Long count = redisTemplate.opsForSet().add(key, values);
        if(time>0) {
            expire(key, time);
        }
        return count;
    } catch (Exception e) {
        e.printStackTrace();
        return 0;
    }
}

/**
 * 获取set缓存的长度
 * @param key 键
 * @return
 */
public long sGetsetSize(String key){
    try {
        return redisTemplate.opsForSet().size(key);
    } catch (Exception e) {
        e.printStackTrace();
        return 0;
    }
}

/**
 * 移除值为value的
 * @param key 键
 * @param values 值 可以是多个
 * @return 移除的个数
 */
public long setRemove(String key, Object ...values) {
```

```
    try {
        Long count = redisTemplate.opsForSet().remove(key, values);
        return count;
    } catch (Exception e) {
        e.printStackTrace();
        return 0;
    }
}

//=====list=====

/***
 * 获取list缓存的内容
 * @param key 键
 * @param start 开始
 * @param end 结束 0 到 -1代表所有值
 * @return
 */
public List<Object> lGet(String key, Long start, Long end){
    try {
        return redisTemplate.opsForList().range(key, start, end);
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

/***
 * 获取list缓存的长度
 * @param key 键
 * @return
 */
public long lGetListsize(String key){
    try {
        return redisTemplate.opsForList().size(key);
    } catch (Exception e) {
        e.printStackTrace();
        return 0;
    }
}

/***
 * 通过索引 获取list中的值
 * @param key 键
 * @param index 索引 index>=0时, 0 表头, 1 第二个元素, 依次类推; index<0时, -1, 表尾, -2倒数第二个元素, 依次类推
 * @return
 */
public Object lGetIndex(String key, long index){
    try {
        return redisTemplate.opsForList().index(key, index);
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

/***
 * 将list放入缓存
 */
```

```
* @param key 键
* @param value 值
* @return
*/
public boolean lSet(String key, Object value) {
    try {
        redisTemplate.opsForList().rightPush(key, value);
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * 将list放入缓存
 * @param key 键
 * @param value 值
 * @param time 时间(秒)
 * @return
*/
public boolean lSet(String key, Object value, Long time) {
    try {
        redisTemplate.opsForList().rightPush(key, value);
        if (time > 0) {
            expire(key, time);
        }
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * 将list放入缓存
 * @param key 键
 * @param value 值
 * @return
*/
public boolean lSet(String key, List<Object> value) {
    try {
        redisTemplate.opsForList().rightPushAll(key, value);
        return true;
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
}

/**
 * 将list放入缓存
 * @param key 键
 * @param value 值
 * @param time 时间(秒)
 * @return
*/
public boolean lSet(String key, List<Object> value, Long time) {
```

```

        try {
            redisTemplate.opsForList().rightPushAll(key, value);
            if (time > 0) {
                expire(key, time);
            }
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 根据索引修改list中的某条数据
     * @param key 键
     * @param index 索引
     * @param value 值
     * @return
     */
    public boolean lUpdateIndex(String key, Long index, Object value) {
        try {
            redisTemplate.opsForList().set(key, index, value);
            return true;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    /**
     * 移除N个值为value
     * @param key 键
     * @param count 移除多少个
     * @param value 值
     * @return 移除的个数
     */
    public long lRemove(String key, long count, Object value) {
        try {
            Long remove = redisTemplate.opsForList().remove(key, count, value);
            return remove;
        } catch (Exception e) {
            e.printStackTrace();
            return 0;
        }
    }
}

```

4.创建实体类

```
@Component
public class User implements Serializable {
    private Integer id;

    private String username;

    private String password;

    public Integer getId() {
        return id;
    }
    ...
}
```

其中实体类必须继承其Serializable类实现其序列化

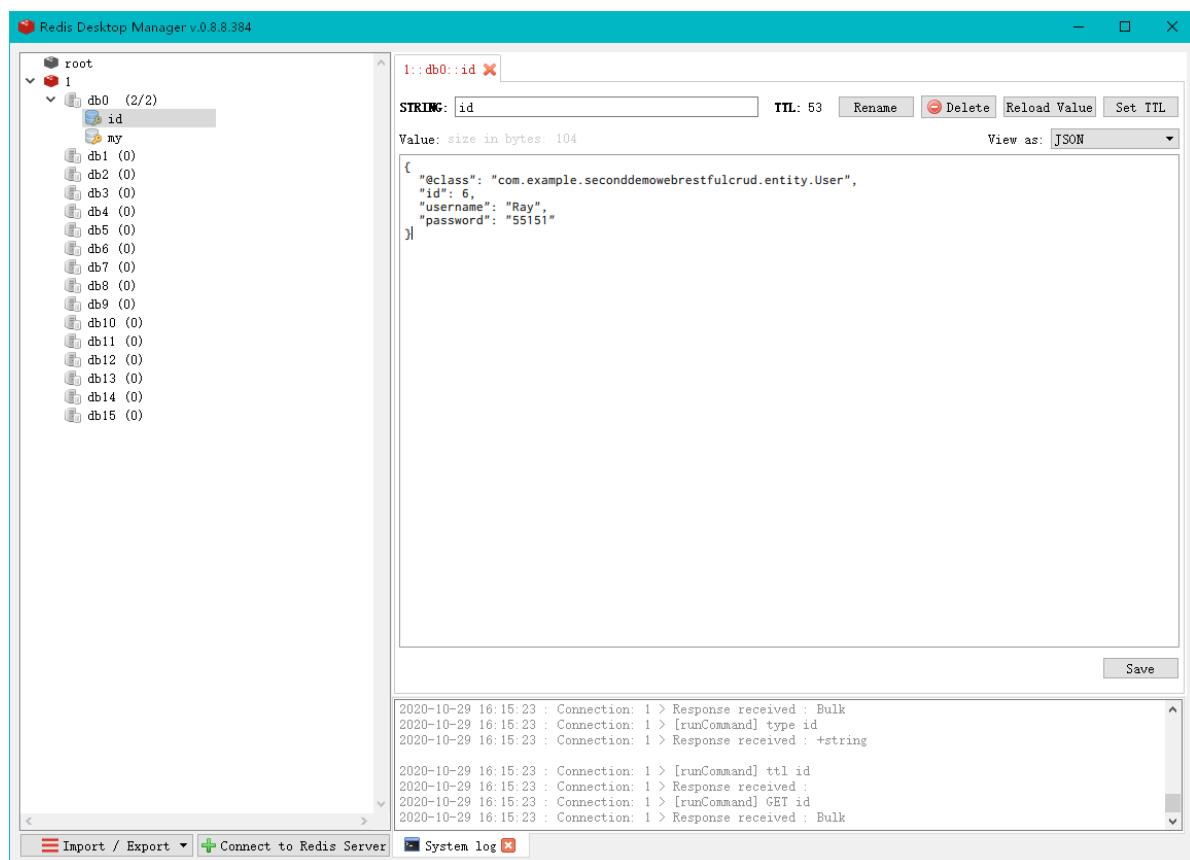
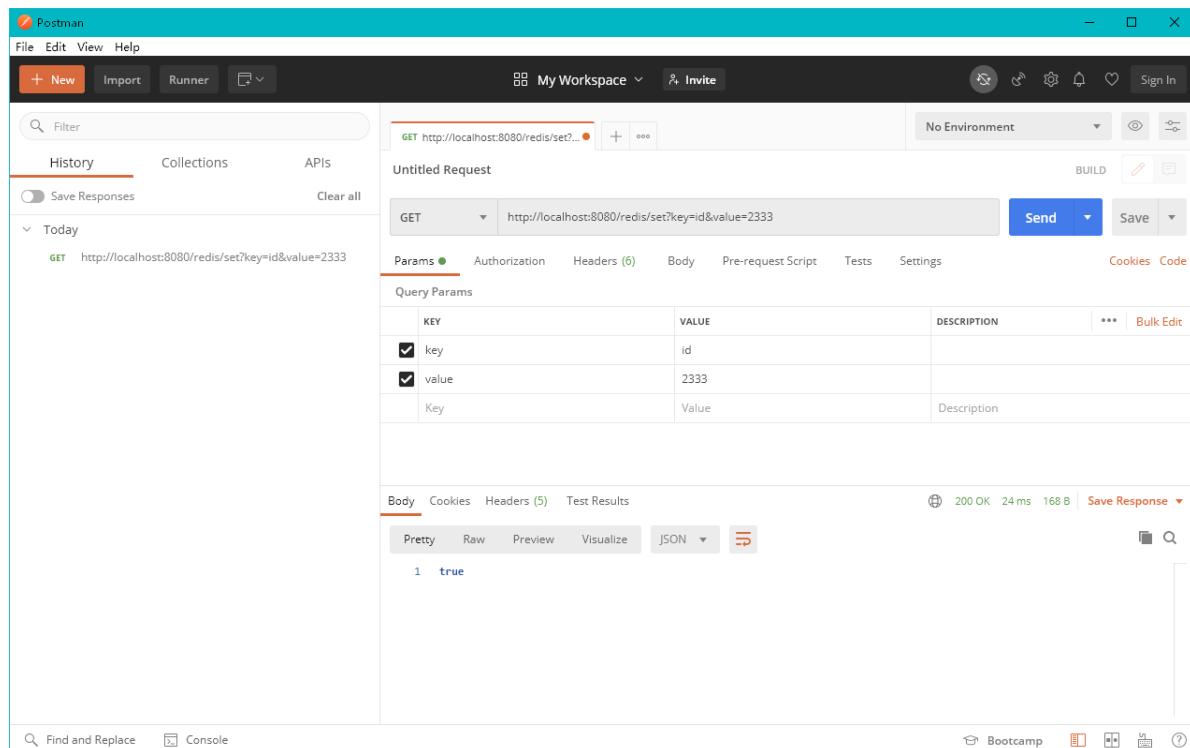
5.在Controller中使用（读写）

```
private static int ExpireTime = 60; //设置redis存储的过期时间为60s

@Resource
private RedisUtil redisUtil;

/**
 * 写入数据至Redis
 * @param key Key值
 * @param value
 * @return
 */
@RequestMapping("set")
public boolean redisSet(String key, String value) {
    User user=new User();
    user.setId(6);
    user.setUsername("Ray");
    user.setPassword("55151");
    return redisUtil.set(key, user, ExpireTime);
}

/**
 * 读取Redis中的键值
 * @param key Key值
 * @return
 */
@RequestMapping("get")
public Object redisGet(String key) {
    return redisUtil.get(key);
}
```



MD5+Salt+Hash散列进行数据加密

对于存储重要信息内容（如密码、支付码）时，为了用户信息安全，必须使用数据加密。

其中MD5+Salt+Hash散列 加密方式比较流行且简便。

主要方法

注册用户时

```
public class User {  
    private Integer id;  
    private String username;  
    private String password;  
    //保存注册时随机盐值，以确保登录时解密  
    private String salt;  
    ...  
}
```

```
public class Saltutils {  
    /**  
     * 生成随机salt的静态方法，以确保salt不固定  
     * @param n  
     * @return  
     */  
    public static String getsalt(int n){  
        char[] chars =  
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*()  
".toCharArray();  
        StringBuffer sb=new StringBuffer();  
        for (int i = 0; i < n; i++) {  
            char c=chars[new Random().nextInt(chars.length)];  
            sb.append(c);  
        }  
        return sb.toString();  
    }  
}
```

```
/**  
 * 以下步骤多在Service层实现  
 */  
//1.生成随机盐  
//2.将随机盐保存到数据  
String salt = Saltutils.getSalt(8);  
//明文密码进行MD5 + salt + hash散列次数  
Md5Hash md5Hash = new Md5Hash(password,salt,1024);  
//将用户输入的密码进行16进制化  
password=md5Hash.toHex();
```

Shiro安全框架

Apache Shiro是一个强大且易用的Java安全框架,执行身份验证、授权、密码和会话管理。使用Shiro的易于理解的API,您可以快速、轻松地获得任何应用程序,从最小的移动应用程序到最大的网络和企业应用程序。

主要功能

shiro主要有三大功能模块：

1. **Subject**: 主体，一般指用户。
2. **SecurityManager**: 安全管理器，管理所有Subject，可以配合内部安全组件。(类似于SpringMVC中的DispatcherServlet)
3. **Realms**: 用于进行权限信息的验证，一般需要自己实现。

细分功能

1. Authentication: 身份认证/登录(账号密码验证)。
2. Authorization: 授权，即角色或者权限验证。
3. Session Manager: 会话管理，用户登录后的session相关管理。
4. Cryptography: 加密，密码加密等。
5. Web Support: Web支持，集成Web环境。
6. Caching: 缓存，用户信息、角色、权限等缓存到如redis等缓存中。
7. Run As: 允许一个用户假装为另一个用户（如果他们允许）的身份进行访问。
8. Remember Me: 记住我，登录后，下次再来的话不用登录了。

Maven依赖

```
<!--shiro-->
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-spring</artifactId>
    <version>1.7.1</version>
</dependency>
```

快速入门语句

```
// 得到DefaultSecurityManager对象
DefaultSecurityManager defaultSecurityManager=new DefaultSecurityManager();
// 读取ini配置文件
IniRealm iniRealm=new IniRealm("classpath:shiro.ini");
// 配置DefaultSecurityManager对象
defaultSecurityManager.setRealm(iniRealm);
// 获取SecurityUtils对象
SecurityUtils.setSecurityManager(defaultSecurityManager);

// 获取当前用户对象 Subject
Subject currentUser = SecurityUtils.getSubject();

// 通过当前用户获取Session
Session session = currentUser.getSession();

// 判断用户是否被认证
currentUser.isAuthenticated()

// 通过Token进行登录操作
currentUser.login(token)
```

```

//根据输入账户名和密码获取Token
UsernamePasswordToken token = new UsernamePasswordToken("lonestarr",
"vespa");

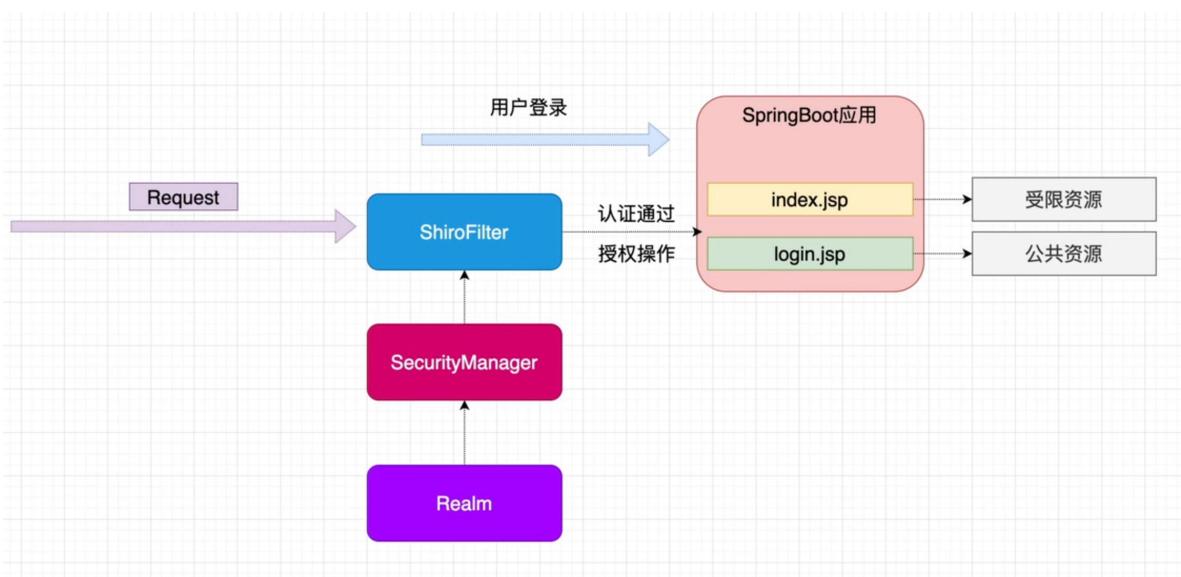
//判断用户的身份
currentUser.hasRole("schwartz")

//判断用户拥有的权限
currentUser.isPermitted("lightsaber:wield")

//注销当前用户
currentUser.logout();

```

SpringBoot继承Shiro



Maven依赖

```

<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-spring-boot-starter</artifactId>
    <version>1.6.0</version>
</dependency>

```

创建Realm类

```

public class CustomRealm extends AuthorizingRealm {

    @Autowired
    private LoginService loginService;

    /**
     * @MethodName doGetAuthorizationInfo
     * @Description 权限配置类
     * @Param [principalCollection]
     * @Return AuthorizationInfo
     * @Author WangShiLin
    */
}

```

```

/*
@Override
protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principalCollection) {
    //获取登录用户名
    String name = (String) principalCollection.getPrimaryPrincipal();
    //查询用户名称
    User user = loginService.getUserByName(name);
    //添加角色和权限
    SimpleAuthorizationInfo simpleAuthorizationInfo = new
SimpleAuthorizationInfo();
    for (Role role : user.getRoles()) {
        //添加角色
        simpleAuthorizationInfo.addRole(role.getRoleName());
        //添加权限
        for (Permissions permissions : role.getPermissions()) {
            //将用户拥有的权限加载到权限中

            simpleAuthorizationInfo.addStringPermission(permissions.getPermissionsName());
        }
    }
    return simpleAuthorizationInfo;
}

/**
 * @MethodName doGetAuthenticationInfo
 * @Description 认证配置类
 * @Param [authenticationToken]
 * @Return AuthenticationInfo
 * @Author WangShiLin
 */
@Override
protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken
authenticationToken) throws AuthenticationException {
    if (StringUtils.isEmpty(authenticationToken.getPrincipal())) {
        return null;
    }
    //获取用户信息
    String name = authenticationToken.getPrincipal().toString();
    User user = loginService.getUserByName(name);
    if (user == null) {
        //这里返回后会报出对应异常
        return null;
    } else {
        //这里验证authenticationToken和simpleAuthenticationInfo的信息
        SimpleAuthenticationInfo simpleAuthenticationInfo = new
SimpleAuthenticationInfo(name,
        user.getPassword().toString(),ByteSource.Util.bytes("x23*2d"),getName());
        return simpleAuthenticationInfo;
    }
}

```

创建Realm类继承AuthorizingRealm，重写doGetAuthorizationInfo（授权配置）、
doGetAuthenticationInfo（认证配置）方法。

其中AuthenticationToken 用于收集用户提交的身份（如用户名）及凭据（如密码）。

其中ByteSource.Util.bytes方法为用户设置时的随机盐值。

创建ShiroConfig配置类

```
@Configuration
public class ShiroConfig {
    //将自己的验证方式加入容器
    @Bean
    public CustomRealm myShiroRealm() {
        CustomRealm myShiroRealm = new CustomRealm();
        //设置realm hash验证
        HashedCredentialsMatcher credentialsMatcher= new
        HashedCredentialsMatcher();
        //使用加密方法
        credentialsMatcher.setHashAlgorithmName("md5");
        //散列次数
        credentialsMatcher.setHashIterations(1024);
        myShiroRealm.setCredentialsMatcher(credentialsMatcher);
        return myShiroRealm;
    }

    //权限管理，配置主要是Realm的管理认证
    @Bean
    public DefaultWebSecurityManager securityManager() {
        DefaultWebSecurityManager securityManager = new
        DefaultWebSecurityManager();
        securityManager.setSessionManager(sessionManager());
        //绑定Realm
        securityManager.setRealm(myShiroRealm);
        return securityManager;
    }

    //Filter工厂，设置对应的过滤条件和跳转条件
    @Bean
    public ShiroFilterFactoryBean shiroFilterFactoryBean(SecurityManager
    securityManager) {
        ShiroFilterFactoryBean shiroFilterFactoryBean = new
        ShiroFilterFactoryBean();
        shiroFilterFactoryBean.setSecurityManager(securityManager);
        Map<String, String> map = new HashMap<>();
        //登出
        map.put("/logout", "logout");
        //对所有用户认证
        map.put("/**", "authc");
        //登录
        shiroFilterFactoryBean.setLoginUrl("/login");
        //首页
        shiroFilterFactoryBean.setSuccessUrl("/index");
        //错误页面，认证不通过跳转
        shiroFilterFactoryBean.setUnauthorizedUrl("/error");
        shiroFilterFactoryBean.setFilterChainDefinitionMap(map);
        return shiroFilterFactoryBean;
    }

    //设置SessionManager，防止Shiro在第一次重定向时发送Jsessionid
    @Bean
    public DefaultWebSessionManager SessionManager(){}
```

```

DefaultwebSessionManager defaultSessionManager = new
DefaultwebSessionManager();
    //将sessionIdUrlRewritingEnabled属性设置成false
    defaultSessionManager.setSessionIdUrlRewritingEnabled(false);
    return defaultSessionManager;
}

//注入权限管理
@Bean
public AuthorizationAttributeSourceAdvisor
authorizationAttributeSourceAdvisor(SecurityManager securityManager) {
    AuthorizationAttributeSourceAdvisor authorizationAttributeSourceAdvisor
= new AuthorizationAttributeSourceAdvisor();
    authorizationAttributeSourceAdvisor.setSecurityManager(securityManager);
    return authorizationAttributeSourceAdvisor;
}
}

```

其中shiro内置过滤器：

```

anno: 无需认证即可访问
authc: 必须认证才可以访问
user : 不许拥有记住我功能才能访问
perms: 拥有对某个资源访问权限才能使用 ((perms认证必须放在authc认证前, 否则无效))
role: 拥有某个角色权限才能访问

```

权限限定访问：

```
map.put("/set", "perms[user:set]"); //只限定拥有‘user:set’权限的用户访问
```

Controller类

```

@RestController
@Slf4j
public class LoginController {

    @GetMapping("/login")
    public String login(User user) {
        if (StringUtils.isEmpty(user.getUserName()) ||
Stringutils.isEmpty(user.getPassword())) {
            return "请输入用户名和密码!";
        }
        //用户认证信息
        Subject subject = SecurityUtils.getSubject();
        UsernamePasswordToken usernamePasswordToken = new UsernamePasswordToken(
            user.getUserName(),
            user.getPassword()
        );
        try {
            //进行验证, 这里可以捕获异常, 然后返回对应信息
            subject.login(usernamePasswordToken);
            subject.checkRole("admin");
            subject.checkPermissions("query", "add");
        } catch (UnknownAccountException e) {
            log.error("用户名不存在!", e);
            return "用户名不存在!";
        }
    }
}

```

```

        } catch (AuthenticationException e) {
            log.error("账号或密码错误!", e);
            return "账号或密码错误!";
        } catch (AuthorizationException e) {
            log.error("没有权限!", e);
            return "没有权限";
        }
        return "login success";
    }

    ...
}

```

- 1.用SecurityUtils.getSubject()获取Subject类。
- 2.将用户输入进去的账户密码信息封装入UsernamePasswordToken类。
- 3.使用Subject类的login方法判断登录结果，并捕捉相关错误异常。

登录错误异常

- UnknownAccountException: 用户名不存在
- AuthenticationException: 账户或者密码错误
- AuthorizationException: 没有权限
- Account Exception : 账号异常
 - ConcurrentAccessException: 并发访问异常 (多个用户同时登录时抛出)
 - UnknownAccountException: 未知的账号
 - ExcessiveAttemptsException: 认证次数超过限制
 - DisabledAccountException: 禁用的账号
 - LockedAccountException: 账号被锁定
 - UnsupportedTokenException: 使用了不支持的Token

Shiro+Thymeleaf页面整合

Maven依赖:

```

<!-- https://mvnrepository.com/artifact/com.github.theborakompanioni/thymeleaf-
extras-shiro -->
<dependency>
    <groupId>com.github.theborakompanioni</groupId>
    <artifactId>thymeleaf-extras-shiro</artifactId>
    <version>2.0.0</version>
</dependency>

```

Thymeleaf页面头部加入 xmlns:shiro="<http://www.pollix.at/thymeleaf/shiro>" 开启代码提示。

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"
      xmlns:shiro="http://www.pollix.at/thymeleaf/shiro">
    ...
</html>

```

常用标签：

The has Permission tag

shiro:hasPermission="xxx" 判断当前用户是否拥有xxx权限

```
<div shiro:hasPermission="user:set"></div>
```

The authenticated tag

authenticated="" 已经用户得到认证

```
<a shiro:authenticated="" href="updateAccount.html">Update your contact information</a>
```

The hasRole tag

shiro:hasRole="xxx" 判断当前用户为xxx权限

```
<a shiro:hasRole="administrator" href="admin.html">Administer the system</a>
```

权限、角色访问控制

方法一：直接在页面控制（以Thymeleaf为例）

```
<!--拥有user:add权限的任何人才能看见-->
<div shiro:hasPermission="user:add:*>
    <a th:href="@{/user/add}">Add</a>
</div>
<!--拥有admin角色才能看见-->
<div shiro:hasRole="admin">
    <a th:href="@{/user/update}">Update</a>
</div>
```

方法二：Controller代码层中控制

```
//获取当前用户
Subject subject = SecurityUtils.getSubject();
if (subject.hasRole("admin")) {
    System.out.println("添加成功！");
} else {
    System.out.println("添加失败！");
}
```

方法三：代码注释控制

```
@RequestMapping("/user/add")
@RequiresRoles("admin") //判断角色
@RequiresPermissions("user:add:*) //判断权限
public String add() {
    return "user/add";
}
```

JAVA设计模式

OOP七大原则

开：开闭原则

口：接口隔离原则

里：里氏替换原则

合：合成复用原则

最：最少知道原则（迪米特原则）

单：单一职责原则

依：依赖倒置原则

单例模式

单例模式（Singleton Pattern）是Java中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

注意：

- 1、单例类只能有一个实例。
- 2、单例类必须自己创建自己的唯一实例。
- 3、单例类必须给所有其他对象提供这一实例。
-

单例模式类的构造器必须为private私有化，并且需要内建一个实例，必为其提供一个Get实例的公共方法。

获取单例模式的类时只有通过使用该Get方法获取当前实例。

```
public class SingleObject {  
  
    //创建 SingleObject 的一个对象  
    private static SingleObject instance = new SingleObject();  
  
    //让构造函数为 private，这样该类就不会被实例化  
    private SingleObject(){}  
  
    //获取唯一可用的对象  
    public static SingleObject getInstance(){  
        return instance;  
    }  
  
    public void showMessage(){  
        System.out.println("Hello world!");  
    }  
}
```

工厂模式

工厂模式实现了创建者和调用者的分离

工厂模式分为: 简单工厂模式

 工厂方法模式

 抽象工厂模式

核心本质: 实例化对象不能使用new,而用工厂方法代替

 将选择实现类,建立统一管理和控制.

工厂模式需要创建一个工厂类,用来返回对应需求的实体类.

```
public class Rectangle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Inside Rectangle::draw() method.");  
    }  
}
```

```
public class ShapeFactory {  
  
    //使用 getShape 方法获取形状类型的对象  
    //简单工厂模式  
    public Shape getShape(String shapeType){  
        if(shapeType == null){  
            return null;  
        }  
        if(shapeType.equalsIgnoreCase("CIRCLE")){  
            return new Circle();  
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){  
            return new Rectangle();  
        } else if(shapeType.equalsIgnoreCase("SQUARE")){  
            return new Square();  
        }  
        return null;  
    }  
}
```

Swagger

简介

Swagger 是一个规范和完整的框架，用于生成、描述、调用和可视化 RESTful 风格的 Web 服务。

前后端分离 Vue+SpringBoot

1可以通过Swagger 给一些难以理解 的属性或者接口,增加注释信息.

2接口文档实时更新

3可以在线测试

SpringBoot集成Swagger

1.新建SpringBoot-Web项目

2.导入相关Maven依赖

```
<!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger-ui -->
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
<!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger2 -->
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
</dependency>
```

3.编写一个Hello工程

4.配置Swagger Config

```
@Configuration
@EnableSwagger2 //开启Swagger2
public class SwaggerConfig {

}
```

打开<http://localhost:8080/swagger-ui.html>

The screenshot shows the Swagger UI homepage. At the top, there's a navigation bar with links for 'Api Documentation', 'Terms of service', and 'Apache 2.0'. Below the navigation, there are two main sections: 'basic-error-controller' and 'hello-controller'. Under 'basic-error-controller', there's a 'Models' section with three items: 'ModelAndView', 'View', and '用户实体类'. The 'hello-controller' section is collapsed. At the bottom right of the page, there's a search bar labeled 'Select a spec' with the value 'zty2'.

配置Swagger

Swagger的bean实例Docket;

```
@Configuration
@EnableSwagger2 //开启Swagger2
public class SwaggerConfig {

    @Bean
    public Docket docket(){
        return new Docket(DocumentationType.SWAGGER_2)
            .apiInfo(apiInfo());
    }

    //配置Swagger信息 apiInfo类
    private ApiInfo apiInfo(){
        Contact contact = new Contact("Zssaer", "https://baidu.com",
"641348448@qq.com");
        return new ApiInfo("Zssaer的Swagger API文档",
"Api Documentation",
"1.1",
"urn:tos",
contact,
"Apache 2.0",
"http://www.apache.org/licenses/LICENSE-2.0",
new ArrayList());
    }
}
```

Swagger配置扫描接口

```
@Bean
public Docket docket(){
    return new Docket(DocumentationType.SWAGGER_2)
        .apiInfo(apiInfo())
        //是否开启Swagger，默认为true
        .enable(true/false)
        .select()
        //RequestHandlerSelectors，配置要扫描接口的方式
        //basePackage：指定地址扫描
        //any：扫描全部
        //none：不扫描
        //withClassAnnotation：扫描类上的注解，参数是一个注解的反射对象
        //withMethodAnnotation：扫描方法上的注解

        .apis(RequestHandlerSelectors.basePackage("com.example.swaggercontroller")/
        .any/.none/
            .withClassAnnotation/.withMethodAnnotation)
        //过滤路径
        .paths(PathSelectors.ant("/swaggertest/**"))
        .build();
}

//配置Swagger扫描指定地址的Controller
Docket.select(RequestHandlerSelectors.basePackage(包名地址)).build()
```

Swagger只在生产环境中使用,不在发布时使用:

- 判断是否为生产环境 flag=false
- 注入enabel(false)

1.在application.properties表明使用的配置文件

```
spring.profiles.active=dev
```

使用对应的application-* .properties的配置文件 配置其他内容; *:spring.profiles.active的内容

2.在SwaggerConfig配置中进行判断

```
@Bean
public Docket docket(Environment environment){
    //获取项目的开发环境:
    //设置要显示的Swagger环境,可以规定多个环境
    Profiles profiles=Profiles.of("dev", "xx", ...);
    //通过environment.acceptsProfiles判断是否当前是否处在设定当中
    boolean flag = environment.acceptsProfiles(profiles);

    return new Docket(DocumentationType.SWAGGER_2)
        .apiInfo(apiInfo())
        //判断是否开启Swagger
        .enable(flag)
        .select()
        //RequestHandlerSelectors,配置要扫描接口的方式

    .apis(RequestHandlerSelectors.basePackage("com.example.swaggertest.controller"))
        .build();
}
```

配置API的分组

```
Docket(DocumentationType.SWAGGER_2)
    .groupName("zty")
```

如需要配置多个分组,就需要多个Docket.

```
@Bean
public Docket docket1() {
    return new Docket(DocumentationType.SWAGGER_2).groupName("zty1")
}
@Bean
public Docket docket2() {
    return new Docket(DocumentationType.SWAGGER_2).groupName("zty2")
}
@Bean
public Docket docket3() {
    return new Docket(DocumentationType.SWAGGER_2).groupName("zty3 ")
}
```

扫描实体类

1. 创建一个实体类

```
//ApiModelProperty:在Swagger中对其类进行注释
@ApiModel("用户实体类")
public class User {
    //ApiModelProperty:在Swagger中对其属性进行注释
    @ApiModelProperty("用户名")
    public String username;
    @ApiModelProperty("密码")
    public String password;
}
```

2. 在Swagger扫描的包中返回实体类, Swagger会自动将其导入Model中

```
@Api(tags = "用户管理")
@RestController
public class HelloController {

    //ApiOperation:在Swagger中对其方法进行注释
    @ApiOperation(value = "登录", notes = "用户登录")
    @PostMapping("/user")
    //@ApiParam:在Swagger中对其参数进行注释
    public User user(@ApiParam("用户名") String username){
        return new User();
    }
}
```

The screenshot shows the Swagger UI interface. At the top, there's a green header bar with the title 'swagger' and a dropdown menu 'Select a spec' set to 'zty1'. Below the header, the main content area displays the 'Zssaer的Swagger API文档' (version 1.1). It includes a 'Base URL: localhost:8081/' link and links to 'Terms of service', 'Zssaer - Website', 'Send email to Zssaer', and 'Apache 2.0'. The interface is divided into sections: 'hello-controller' (Hello Controller) and 'Models'. The 'hello-controller' section contains two operations: a 'GET /hello' operation and a 'POST /user' operation. The 'Models' section shows the 'User' entity definition: '用户名' (username) and '密码' (password).

Thymeleaf

Thymeleaf 是一个现代服务器端 Java 模板引擎，用于 web 和独立环境。

Maven依赖

```
<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-thymeleaf -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
    <version>2.3.3.RELEASE</version>
</dependency>
```

首次使用

只要我们把HTML页面放在classpath:/templates/, thymeleaf就能自动渲染;

1.导入thymeleaf的名称空间

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
```

2、使用thymeleaf语法;

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <h1>成功! </h1>
    <!--th:text 将div里面的文本内容设置为 -->
    <div th:text="${hello}">这是显示欢迎信息</div>
</body>
</html>
```

语法规则

Order	Feature		Attributes
1	Fragment inclusion	片段包含: jsp:include	th:insert th:replace
2	Fragment iteration	遍历: c:forEach	th:each
3	Conditional evaluation	条件判断: c:if	th:if th:unless th:switch th:case
4	Local variable definition	声明变量: c:set	th:object th:with
5	General attribute modification	任意属性修改 支持prepend , append	th:attr th:attrprepend th:attrappend
6	Specific attribute modification	修改指定属性默认值	th:value th:href th:src ... th:text th:utext
7	Text (tag body modification)	修改标签体内容	转义特殊字符
8	Fragment specification	声明片段	不转义特殊字符 th:fragment
9	Fragment removal		th:remove

th: 任意html属性; 来替换原生属性的值

1. 表达式

`${...}` : 获取变量值;

1) 、获取对象的属性、调用方法

2) 、使用内置的基本对象:

`#ctx : the context object.`

`#vars: the context variables.`

`#locale : the context locale.`

`#request : (only in Web Contexts) the HttpServletRequest object.`

`#response : (only in Web Contexts) the HttpServletResponse object.`

`#session : (only in Web Contexts) the HttpSession object.`

`#servletContext : (only in Web Contexts) the ServletContext object.`

3) 、内置的一些工具对象:

`#uris : methods for escaping parts of URLs/URIs`

`#dates : methods for java.util.Date objects: formatting, component extraction, etc.`

`#calendars : analogous to #dates , but for java.util.Calendar objects.`

`#numbers : methods for formatting numeric objects.`

`#strings : methods for String objects: contains, startsWith, prepending/appending, etc.`

`#objects : methods for objects in general.`

`#bools : methods for boolean evaluation.`

`#arrays : methods for arrays.`

`#lists : methods for lists.`

`#sets : methods for sets.`

`#maps : methods for maps.`

`*{...}`: 选择表达式: 和 `${...}` 在功能上是一样的;

补充: 配合 `th:object="${session.user}"`:

```
<div th:object="${session.user}">
    <p>Name: <span th:text="*{firstName}">Sebastian</span>.</p>
    <p>Surname: <span th:text="*{lastName}">Pepper</span>.</p>
    <p>Nationality: <span th:text="*{nationality}">Saturn</span>.</p>
</div>
```

#...: 获取国际化内容

@...: 定义URL;

```
<link href="asserts/css/dashboard.css" th:href="@{/asserts/css/dashboard.css}"
      rel="stylesheet">
```

~...: 片段引用表达式

```
<div th:insert="~{commons :: main}">...</div>
```

thymeleaf公共页面元素抽取

1、抽取公共片段

```
<div th:fragment="copy">
    &copy; 2011 The Good Thymes Virtual Grocery
</div>
```

2、引入公共片段

```
<div th:insert="~{footer :: copy}"></div>
~{templatename::selector}: 模板名::选择器
~{templatename::fragmentname}:模板名::片段名
```

3、默认效果:

insert的公共片段在div标签中

如果使用th:insert等属性进行引入，可以不用写~{}:

行内写法可以加上: [[~{}]]; [(~{})];

thymeleaf有三种引入公共片段的th属性：

th:insert: 将公共片段整个插入到声明引入的元素中

th:replace: 将声明引入的元素替换为公共片段

th:include: 将被引入的片段的内容包含进这个标签中

三种方式的区别

```
footer.html下设置th:fragment
<footer th:fragment="copy">
    &copy; 2011 The Good Thymes Virtual Grocery
</footer>
```

引入方式

```
<div th:insert="footer :: copy"></div>
<div th:replace="footer :: copy"></div>
<div th:include="footer :: copy"></div>
```

th:insert的效果

```
<div>
```

```
<footer>
    &copy; 2011 The Good Thymes Virtual Grocery
</footer>
</div>
```

th:replace的效果

```
<footer>
    &copy; 2011 The Good Thymes Virtual Grocery
</footer>
```

th:include的效果

```
<div>
    &copy; 2011 The Good Thymes Virtual Grocery
</div>
```

对于跨路径下引用:

在commos下bar.html中:

```
<!--topbar-->
<nav class="navbar navbar-dark sticky-top bg-dark flex-md-nowrap p-0"
th:fragment="topbar">
    <a class="navbar-brand col-sm-3 col-md-2 mr-0"
    href="http://getbootstrap.com/docs/4.0/examples/dashboard/#">
        [[${session.loginUser}]]</a>
        <input class="form-control form-control-dark w-100" type="text"
    placeholder="Search" aria-label="Search">
    <ul class="navbar-nav px-3">
        <li class="nav-item text-nowrap">
            <a class="nav-link"
    href="http://getbootstrap.com/docs/4.0/examples/dashboard/#">sign out</a>
        </li>
    </ul>
</nav>

<nav class="col-md-2 d-none d-md-block bg-light sidebar" id="sidebar">
    <a class="nav-link active" th:class="${activeUri=='emps'?'nav-link
active':'nav-link'}">
        </a>
</nav>
```

在其他页面下引用

```
...
<div th:replace="commons/bar::topbar"></div>
<!-- 使用#id选择器，()可以传递参数--&gt;
&lt;div th:replace="commons/bar::#sidebar(activeUri='emps')"&gt;&lt;/div&gt;
...&lt;br/&gt;</pre>
```

Vue

环境配置

一.下载Node.js

官方地址: <https://nodejs.org/en/download/>

二.设置nodejs prefix (全局) 和cache (缓存) 路径

在Node.js目录下新建**node_global**和**node_cache**两个文件夹

名称	修改日期	类型	大小
node_cache	2018/2/28 11:22	文件夹	
node_global	2018/2/28 11:27	文件夹	
node_modules	2018/2/28 11:17	文件夹	
node.exe	2018/2/23 2:56	应用程序	21,872 KB
node_etw_provider.man	2018/2/11 1:08	MAN 文件	9 KB
node_perfctr_provider.man	2018/2/11 0:08	MAN 文件	5 KB
nodevars.bat	2018/2/11 1:08	Windows 批处理...	1 KB
npm	2018/2/11 1:08	文件	1 KB
npm.cmd	2018/2/11 0:08	Windows 命令脚本	1 KB
npx	2018/2/15 18:01	文件	1 KB
npx.cmd	2018/2/15 18:01	Windows 命令脚本	1 KB

三.设置缓存文件夹

```
npm config set cache "D:\vueProject\nodejs\node_cache"
```

四.设置全局模块存放路径

```
npm config set prefix "D:\vueProject\nodejs\node_global"
```

五.设置环境变量

将其Node.js目录下**node_global**文件夹添加至Path中。

新增系统变量**NODE_PATH**, 路径为Node.js目录下**node_modules**文件夹。

基于 Node.js 安装cnpm (淘宝镜像)

```
npm install -g cnpm --registry=https://registry.npm.taobao.org
```

六.安装Vue

```
cnpm install vue -g
```

七.安装vue命令行工具, 即vue-cli 脚手架

```
cnpm install vue-cli -g
```

利用vue-cli脚手架搭建新项目

在CMD定位至项目目录下输入

```
vue init webpack-simple 项目名称（使用英文）
```

其中项目名称中不能使用大写.

CMD定位到工程目录下，安装该工程依赖的模块，这些模块将被安装在：项目名称\node_module目录下，node_module文件夹会被新建，而且根据package.json的配置下载该项目的modules

```
cnpm install
```

八.运行Vue项目

```
cnpm run dev / npm run dev
```

九.Vue打包上线

```
npm run build
```

打包完成后，会生成 dist 文件夹，如果已经修改了文件路径，可以直接打开本地文件查看。
项目上线时，只需要将 dist 文件夹放到服务器就行了。

vue.js基础

属性绑定：

```
v-bind:xxx="aaa" / v-bind:title="message3"
```

将aaa内容绑定到XXX属性上

事件监听器：

```
v-on:xxx="aaa" / v-on:click="sayHi"
```

XXX属性进行监听，返回aaa内容

双向绑定：

```
v-model="XXX" / v-model="message"
```

将XXX进行view，model双向绑定到上，修改一处会同时变化

```
<div id="main">
  {{message}}
  {{message2}}
  <!-- 将message3内容绑定到title属性中 -->
  <span v-bind:title="message3">
    鼠标悬停几秒钟查看此处动态绑定的提示信息！
  </span>
  <!--foreach循环-->
  <li v-for="(item,index) in items">
    {{item.ms}}--{{index}}
  </li>

  <button v-on:click="sayHi">click Me</button>
```

```

<!--model双向绑定-->
<input type="text" v-model="message">
<p></p>
<input type="radio" name="sex" value="男" v-model="sex" checked>男
<input type="radio" name="sex" value="女" v-model="sex">女
<p>选择了: {{sex}}</p>
</div>

<!--导入vue.js-->
<script src="https://cdn.jsdelivr.net/npm/vue@2.5.21/dist/vue.min.js"></script>
<script>
var vm=new Vue({
    el:"#main",
    data:{
        message : "Hello,Vue!",
        message2 : "Hello,world!",
        message3 : "Hello,World!!!",
        items : [
            {ms : "ms1"},
            {ms : "ms2"},
            {ms : "ms3"}
        ],
        sex : ""
    },
    // 方法必须定义在vue的methods中
    methods:{
        sayHi:function(){
            alert(this.message);
        }
    }
});
</script>

```

Vue组件

组件是可复用的Vue实例,就是一组可以自定义使用的模板,跟JSTL的自定义标签、Thymeleaf的th:fragment相似。

```

// 定义一个Vue组件 名为 'demo'
Vue.component("demo",{
    template : '<li>Hello,world</li>'
});

var vm= new Vue({
    el : "#main"
});

```

```

<div id="main">
    <demo></demo>
</div>

```

Vue组件不能直接访问data层数据,需要绑定到template中的props属性中。组件中可以带属性。

```

<!--view层-->
<div id="main">
    <!--template不能直接访问data层数据,需要绑定到template中的props属性中-->

```

```

<demo v-for="item in items" v-bind:item1="item"></demo>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue@2.5.21/dist/vue.min.js"></script>
<script>
    // 定义一个Vue组件 名为 'demo' 内含一个'item1'的属性
    Vue.component("demo", {
        props : ['item1'],
        template : '<li>{{item1}}</li>'
    });

    var vm= new Vue({
        el : "#main",
        data : {
            items : ["Vue.js!","JavaScript","JQuery"]
        }
    });
</script>

```

Axios异步通讯

安装Axios:

```
npm install --save axios vue-axios
```

Axios API: 使用 `axios` 可以传递其他链接下的数据,以及传递数据.

main.js中进行配置Axios:

```

import axios from 'axios'
import VueAxios from 'vue-axios'

Vue.use(VueAxios, axios)

```

在前后端分离项目进行前端链接通讯.

```

<div id="vue">
    <div>
        {{info.name}}
        {{info.address.city}}
        <a v-bind:href="info.url">Baidu</a>
    </div>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue@2.5.21/dist/vue.min.js"></script>
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
<script>
    var vm = new Vue({
        el: "#vue",
        data:{ 
            info:null
        },
        mounted(){ //钩子函数
            axios.get('../data.json').then(response=>(this.info=response.data));
        }
    });
</script>

```

```
        }
    });
</script>
```

data.json:

```
{
  "name": "test",
  "url": "http://baidu.com",
  "page": 1,
  "isNonProfit": true,
  "address": {
    "street": "含光门",
    "city": "陕西西安",
    "country": "中国"
  },
  "links": [
    {
      "name": "bilibili",
      "url": "https://www.bilibili.com/"
    },
    {
      "name": "4399",
      "url": "https://www.4399.com/"
    },
    {
      "name": "百度",
      "url": "https://www.baidu.com/"
    }
  ]
}
```

```
mounted() { //钩子函数
  axios.get('../data.json').then(response=>(this.info=response.data));
}
```

Slot插槽

slot:可重复定义的组件

```
<div id="main">
  <todo>
    <utitle slot="title-slot" :title="mytitle"></utitle>
    <items slot="items-slot" v-for="item in myitems" :item="item"></items>
  </todo>
</div>


<script src="https://cdn.jsdelivr.net/npm/vue@2.5.21/dist/vue.min.js"></script>
<script>
  Vue.component("todo",{
    template: '<div>\n      <slot name="title-slot"></slot>\n      <ul>\n        <slot name="items-slot"></slot>\n      </ul>\n    </div>',
    data: function() {
      return {
        mytitle: "我的待办事项",
        myitems: [
          {id: 1, title: "学习Vue.js基础", status: "未完成"},
          {id: 2, title: "完成项目", status: "未完成"},
          {id: 3, title: "优化代码", status: "未完成"}]
      }
    }
  })
</script>
```

```

        </div>'  

});  

Vue.component("utitle", {  

    props: ['title'],  

    template: '<div>{{title}}</div>'  

});  

Vue.component("items", {  

    props: ['item'],  

    template: '<li>{{item}}</li>'  

});  
  

var vm=new Vue({  

    el: "#main",  

    data:{  

        mytitle: 'zssaer',  

        myitems: ["zty","sdf","gfr","rgg","sxc","oju","zty"]  

    }  

});  

</script>

```

zssaer

- zty
- sdf
- gfr
- rgg
- sxc
- oju
- zty

在component中的template下定义插槽.

name为该插槽的名称,当一个组件存在多个slot时,使用name的值来区分.

在view中使用complete充当插槽时,定义slot="slot-name".

注意:在 2.6.0 中, 我们为具名插槽和作用域插槽引入了一个新的统一的语法 (即 `v-slot` 指令)。它取代了 `slot` 和 `slot-scope` 这两个目前已被废弃但未被移除且仍在文档中的 attribute。

自定义事件

\$emit(自定义事件名, 参数):

\$emit 绑定一个自定义事件event，当这个语句被执行到的时候，就会将参数arg传递给父组件，父组件通过@事件(v-on:事件)进行监听并接收参数。

```
<div id="main">
  <todo>
    <utitle slot="title-slot" :title="mytitle"></utitle>
    <items slot="items-slot" v-for="(item,index) in myitems" :item="item"
:index="index" @remove="removeItems(index)"></items>
  </todo>
</div>

<!--导入vue.js-->
<script src="https://cdn.jsdelivr.net/npm/vue@2.5.21/dist/vue.min.js"></script>
<script>
  Vue.component("todo",{
    template: '<div>\n      <slot name="title-slot"></slot>\n      <ul>\n        <slot name="items-slot"></slot>\n      </ul>\n    </div>'
  });
  Vue.component("utitle",{
    props:['title'],
    template: '<div>{{title}}</div>'
  });
  Vue.component("items",{
    props:['item','index'],
    template: '<li>{{index}}---{{item}} <button
@click="remove">Delete</button></li>',
    methods:{
      remove: function () {
        this.$emit('remove',this.index)
      }
    }
  });

var vm=new Vue({
  el: "#main",
  data:{
    mytitle: 'zssae',
    myitems: ["zty","sdf","gfr","rgg","sxc","oju","zty"]
  },
  methods:{
    removeItems:function (index) {
      console.log("删除了"+this.myitems[index]+"元素");
      this.myitems.splice(index,1); //一次删除一个元素
    }
  }
});
```

WebPack打包

WebPack可以将其项目modules中的js文件全部打包为一个js文件.减少项目导入的资源量.

1.使用npm,安装webpack以及webpack-cli

```
npm install webpack -g  
npm install webpack-cli -g
```

2.在modules文件夹定义一个hello.js文件

```
//暴露其方法,使其可以被外部调用. 相当于JAVA中的public修饰符  
exports.sayHi = function () {  
    document.write("<h1>Hi !</h1>");  
}
```

3.在modules文件夹定义入口文件main.js文件

```
var hello = require("./hello"); //使用require方法调用其它module文件,并将其赋值给  
hello.相当于JAVA中的new  
hello.sayHi(); //使用其中的方法
```

2.在项目主目录下创建webpack.config.js文件

```
module.exports = {  
    entry: './modules/main.js', //定义其js入口文件  
    // mode: 'development',  
    output: {  
        filename: "./js/bundle.js" //输出位置,其外自动为dist文件夹  
    }  
};
```

3.使用webpack在项目主目录下打包

```
webpack
```

Vue-router路由

1.在项目中安装vue-router(如果使用脚手架构建vue项目时安装了router的话可以跳过)

```
npm install vue-router --save-dev  
//cnpm install vue-router --save-dev
```

2.在src下新建router文件夹,并创建index.js文件进行配置Router. ('index.js'为Vue默认下的router文件)

```
import Vue from 'vue'  
import Router from 'vue-router'  
/*导入相关组件*/  
import Content from '../components/Content'  
import IndexPage from '../components/Index'
```

```

//安装路由
Vue.use(VueRouter);

//配置导出路由
export default new VueRouter({
  routes:[
    {
      //路由路径 相当于Spring中的@RequestMapping
      path: '/content',
      name: 'content',
      //跳转的组件
      component: Content
    },
    {
      path: '/main',
      name: 'main',
      component: IndexPage
    }
    ...
  ]
});

```

注意:使用import引入component的话,会导致其项目打包时路由里的所有component都会被打包在一个js中,造成进入首页时,需要加载的内容过多,时间相对比较长。

当你用require这种方式引入的时候,会将你的component分别打包成不同的js,加载的时候也是按需加载,只用访问这个路由网址时才会加载这个js。

```

resolve => require(['../components/Index'], resolve),


{
  // 进行路由配置,规定'/'引入到home组件
  path: '/',
  component: resolve => require(['../components/Index'], resolve),
  meta: {
    title: 'home'
}

```

3.在入口main.js文件下导入,配置router

```

import Router from './router' //Vue默认配置下会自动扫描router下的index文件

new Vue({
  el: '#app',
  //配置路由
  router: Router,
  ...
})

```

4.在显示Vue下进行使用Router

```
<template>
```

```

<div id="app">
  <h1>ONEREPUBLIC OF THE POP ARTISTS FROM UNITED STATES OF AMERICA</h1>
  <!-- 设置Router超链接,相当于a -->
  <router-link to="/content">CONTENT</router-link>
  <router-link to="/main">INDEX</router-link>
  <!-- 展示Router内容 -->
  <router-view></router-view>
</div>
</template>

<script>
export default {
  name: 'App',
}
</script>

```

:进行设置路由链接 :进行展示路由内容

在方法中进行路由转发跳转导航:

```

// 使用vue-router路由到指定页面,该方式叫做编程式导航
this.$router.push("/main");

```

路由嵌套

在一个页面中部分切换显示内容,而不跳转新页面. 做到局部刷新,则可以使用Vue-router的路由嵌套方法. 如导航栏点击切换主内容,页面局部刷新,导航栏部分不变,主内容改变.

```

...
<el-submenu index="1">
  <template slot="title"><i class="el-icon-message"></i>用户管理</template>
  <el-menu-item index="1-1">
    <!-- 设置路由跳转链接 -->
    <router-link to="/user/profile">个人信息</router-link>
  </el-menu-item>
  <el-menu-item index="1-2">
    <!-- 设置路由跳转链接 -->
    <router-link to="/user/list">用户列表</router-link>
  </el-menu-item>
</el-submenu>
...

<el-main>
  <!-- 在该部分进行路由显示 -->
  <router-view />
</el-main>

```

在路由配置文件下:

```

routes:[
  ...
  {
    path: '/main',
    component: resolve => require(['../views/Main.vue'], resolve),
  }
]

```

```
// 嵌套路由，做到局部刷新
children:[
  {
    path: '/user/profile',
    component: resolve => require(['../views/user/Profile.vue'], resolve)
  },
  {
    path: '/user/list',
    component: resolve => require(['../views/user/List.vue'], resolve)
  },
]
},
...
]
```

路由参数传递

路由跳转时，可以传递参数。

```
<router-link :to="{name: 'UserProfile', params: {id:23}}">个人信息</router-link>
```

在路由配置中：

```
{
  path: '/main',
  component: resolve => require(['../views/Main.vue'], resolve),
  // 嵌套路由
  children:[
    {
      // 后续添加：参数名，以实现参数传递
      path: '/user/profile:id',
      // 定义其路由名
      name: 'UserProfile',
      component: resolve => require(['../views/user/Profile.vue'], resolve)
    },
    {
      path: '/user/list',
      component: resolve => require(['../views/user/List.vue'], resolve)
    }
}
```

v-bind:to="{name:'路由名(在配置中定义)',params:{参数名:参数值}}"

```
<font color="red">--但会在地址中 会暴露参数值 --</font>
```

1. 通过路由直接获取参数值

在传递后的页面中显示：

```
<template>
<div>
  <h1>个人信息</h1>
  {{$route.params.id}}
</div>
</template>
```

使用{{\$route.params.参数名}}进行提取出参数值

注意:在Vue中声明其template下只能存在一个根元素,不能存在一个以上的元素.

2.通过props进行获取参数

在路由配置中:

```
{
  path: '/user/profile:id',
  name: 'UserProfile',
  component: resolve => require(['../views/user/Profile.vue'], resolve),
  // 设置其允许接收属性值
  props: true
}
```

在传递后的页面中显示:

```
<div>
  <h1>个人信息</h1>
  {{$id}}
</div>

<script>
export default {
  // 获取其传递的值
  props: ['id'],
  name: "UserProfile"
}
</script>
```

路由重定向

使用Vue-Router可以将转发页面进行重定向.

在路由配置中:

```
{
  path: '/goHome',
  // 跳转页面进行重定向
  redirect: '/main'
}
```

参数检验

```
validate({参数值}){
    return true/false
}
```

ElementUI

ElementUi是一个国内基于Vue2.0的桌面端组件库,类似于Bootstrap.

```
npm i element-ui -s
# 安装Sass加载器
npm i node-sass sass-loader --save-dev
```

在main.js文件中:

```
import ElementUI from 'element-ui';
import 'element-ui/lib/theme-chalk/index.css'; // 默认主题

Vue.use(ElementUI);

new Vue({
  el: '#app',
  ...
  render: h => h(App) //Element UI
})
```

在单个页面设置Body背景颜色:

```
export default {
  ...
  // 更改单个页面颜色
  mounted() {
    //设置背景颜色
    document.querySelector('body').setAttribute('style', 'background-color:#268785')
  },
  beforeDestroy() {
    document.querySelector('body').removeAttribute('style')
  }
}
```

单个页面设置显示标题

在Vue项目中的页面不包含head元素,所以无法通过head修改其title 为其修改页面标题.

1.修改项目title

通过修改项目中的index.html文件中的title来实现更改首页项目的显示标题

2.单个页面设置不同标题

在路由配置中:

```
routes: [
  {
    path: '/index',
    name: 'index',
    component: Index,
    meta: {
      // 页面标题title
      title: '首页'
    }
    ...
  ]
]
```

在main.js中:

```
import router from './router'

router.beforeEach((to, from, next) => {
  /* 路由发生变化修改页面title */
  if (to.meta.title) {
    document.title = to.meta.title
  }
  next()
})
```

设置404页面

在路由配置中:

```
routes:[
  ...
  {
    // * 代表错误页面
    path:'*',
    component: resolve => require(['../views/404.vue'], resolve)
  }
]
```

路由钩子

beforeRouteEnter:在路由前执行 其效果相当于拦截器 chain

beforeRouteLeave:离开路由前执行

```
beforeRouteEnter:(to,from,next)=>{
    ...
    next();
}
beforeRouteLeave:(to,from,next)=>{
    ...
    next();
}
```

参数说明:

to:路由将要跳转的路径信息

from:路径跳转前的路径信息

next:路由的控制参数

next():跳入下一个页面

next('/path'):改变路由的跳转方向,使其跳到另一个路由

next(false):取消路由跳转,返回原来的页面

next((vm)=>{}):仅在beforeRouteEnter中使用,vm是组件实例

在钩子函数中使用异步请求

1.安装Axios

```
npm install axios -s
# cnpm install axios -s
npm install --save axios vue-axios
# cnpm install --save axios vue-axios
```

2.在入口文件main.js配置Axios

```
import Vue from 'vue'
import axios from 'axios'
import VueAxios from 'vue-axios'

Vue.use(VueAxios, axios)
```

按照这个顺序分别引入这三个文件: `vue`, `axios` 和 `vue-axios`

3.在应用的component中使用Axios:

```

methods: {
    getData:function(){
        this.axios({
            method: 'get',
            url: 'http://localhost:8080/static/mock/data.json'
        }).then((response)=>{
            console.log(response.data)
            this.posts=response.data;
        });
    }
}

```

4.在路由钩子中应用:

```

beforeRouteEnter:(to,from,next)=>{
    console.log("进入路由之前");
    next(vm => {
        vm.getData(); //进入路由前执行getData方法
    });
}

```

Vue前端传输表单

```

<el-form
    :model="ruleForm"
    :rules="rules"
    ref="ruleForm"
    label-width="70px"
    label-position="left"
    class="login-form"
    >
    <el-form-item prop="userName" label="用户名">
        <el-input v-model="ruleForm.userName" prefix-icon="fa fa-user"
        placeholder="请输入登录名称"></el-input>
    </el-form-item>
    <el-form-item prop="password" label="密码">
        <el-input
            type="password"
            placeholder="请输入登录密码"
            prefix-icon="fa fa-lock"
            v-model="ruleForm.password"
            @keyup.enter.native="submitForm('ruleForm')"
        ></el-input>
    </el-form-item>
    <el-form-item prop label="验证码">
        <el-col :span="10">
            <el-input
                v-model="ruleForm.verifyCode"
                prefix-icon="fa fa-get-pocket"
                placeholder="输入验证码"
                @keyup.enter.native="submitForm('ruleForm')"
            ></el-input>
        </el-col>
        <el-col :span="12" :offset="2">
            <div class="imgDiv">

```

```
        
    </div>
</el-col>
<div class="login-btn">
    <el-button type="warning" @click="submitForm('ruleForm')"
:loading="loging">登 录</el-button>
    </div>
</el-form-item>
</el-form>
...
<script>
    import { userLogin, getCationCode } from "../../api/api";
    const code = "200";
    export default {
        data() {
            ...
            return {
                doCaptcha: "",
                loging: false,
                ruleForm: {
                    userName: "",
                    password: "",
                    verifyCode: "",
                    validKey: ""
                },
                rules: {
                    userName: [{ required: true, message: "请输入用户名", trigger: "blur" }],
                    password: [{ required: true, message: "请输入密码", trigger: "blur" }],
                    verifyCode: [{ required: true, message: "请输入验证码", trigger: "blur" }]
                },
                ...
            }
        },
        methods: {
            editCaptcha() {
                this.getImg();
            },
            getImg() {
                getCaptcha({}).then((res) => {
                    if (res.data.code == code) {
                        this.doCaptcha = res.data.data.baseImg;
                        this.ruleForm.validKey = res.data.data.validKey;
                    }
                });
            },
            submitForm(formName) {
                const self = this;
                self.$refs[formName].validate(valid => {
                    if (valid) {
                        this.loging = true;
                        userLogin(this.ruleForm).then(res => {
                            this.loging = false;
                            if (res.data.code == code) {
                                localStorage.setItem("user", JSON.stringify(res.data.data.userInfo));
                            }
                        });
                    }
                });
            }
        }
    }
</script>
```

```
localStorage.setItem("voucher",
res.data.data.voucher);
if(res.data.data.isInitialPassword == 1) { //初始
密码
    this.$alert('检测到您的密码为初始密码，请点击确定后
进行修改', '提示', {
        confirmButtonText: '确定',
        callback: action => {this.dialogVisible
= true;}
    });
} else {
    this.$message({
        message: "登录成功！",
        type: "success"
    });
    window.name = res.data.data.voucher;
    self.$router.push("/readme");
}
} else {
    this.$message.error(res.data.message);
    return false;
}
}).catch(function(error) {
    console.log(error);
    self.logging = false;
    self.$message.error("网络故障，请稍后再试");
});
} else {
    console.log("error submit!!!");
    return false;
}
});
},
},
mounted() {
    this.getImg();
},
}

}
</script>
```

设置VUE页面切换动画

在进入/离开的过渡中，会有 6 个 class 切换。

1. `v-enter` : 定义进入过渡的开始状态。在元素被插入之前生效，在元素被插入之后的下一帧移除。
2. `v-enter-active` : 定义进入过渡生效时的状态。在整个进入过渡的阶段中应用，在元素被插入之前生效，在过渡/动画完成之后移除。这个类可以被用来定义进入过渡的过程时间，延迟和曲线函数。
3. `v-enter-to` : 2.1.8 版及以上 定义进入过渡的结束状态。在元素被插入之后下一帧生效 (与此同时 `v-enter` 被移除)，在过渡/动画完成之后移除。
4. `v-leave` : 定义离开过渡的开始状态。在离开过渡被触发时立刻生效，下一帧被移除。
5. `v-leave-active` : 定义离开过渡生效时的状态。在整个离开过渡的阶段中应用，在离开过渡被触发时立刻生效，在过渡/动画完成之后移除。这个类可以被用来定义离开过渡的过程时间，延迟和曲线函数。
6. `v-leave-to` : 2.1.8 版及以上 定义离开过渡的结束状态。在离开过渡被触发之后下一帧生效 (与此同时 `v-leave` 被删除)，在过渡/动画完成之后移除。

1. 在 app.vue 里使用 transition 标签

```
<div id="app">
  <transition :name="transitionName">
    <router-view/>
  </transition>
</div>
...
<script>
export default {
  ...
  data(){
    ...
    return{
      ...
      transitionName: "slide-right" // 初始化过度动画
    }
  },
  watch: {
    $route(to,from){
      // 切换动画
      let isBack = this.$router.isBack; // 监听路由变化时的状态为前进还是后退
      if(isBack){
        this.transitionName = "slide-right";
      }else{
        this.transitionName = "slide-left";
      }
      this.$router.isBack = false;
    }
  }
}
```

```
</script>
```

2.在app.vue里监听用户是跳转还是后退

```
...
window.addEventListener('popstate',function(e){
    router.isBack = true
},false)
```

3.编写动画效果

```
.slide-left-enter, .slide-right-leave-to {
    opacity: 0;
    transform: translateX(100%)
}

.slide-left-leave-to, .slide-right-enter {
    opacity: 0;
    transform: translateX(-100%)
}

.slide-left-enter-active, .slide-left-leave-active, .slide-right-enter-active,
.slide-right-leave-active {
    transition: 1.5s;
    position: absolute;
    top:0;
}
...
```

4.配置路由跳转

Vue常见问题

VUE使用axios数据请求时将其数据赋值报错 TypeError: Cannot set property 'xxxx' of undefined

在函数里面进行赋值 `this.list = response.data.result` 报错TypeError: Cannot set property 'listgroup' of undefined

主要原因：

在 then的内部不能使用Vue的实例化的this, 因为在内部 this 没有被绑定。

解决办法：

1、用ES6箭头函数，箭头方法可以和父方法共享变量

```
# ...then(function(response)){});
...then((response)=>{
    this.posts=response.data;
});
```

2、在请求axios外面定义一下 var that=this

```
getData:function(){
    var that=this;
    this.axios({
        method: 'get',
        url: 'http://localhost:8080/static/mock/data.json'
    }).then(function(response){
        console.log(response.data)
        that.posts=response.data;
    });
}
```

Vue3

创建项目

1. 使用命令行创建

```
vue create `项目名`
```

2. 使用图形UI创建

```
vue ui
```

Setup()

使用 `setup` 函数时，它将接受两个参数：

1. `props`
2. `context`

其中

Props: `setup` 函数中的第一个参数是 `props`。正如在一个标准组件中所期望的那样，`setup` 函数中的 `props` 是响应式的，当传入新的 prop 时，它将被更新。

```
export default {
  props: {
    title: String
  },
  setup(props) {
    console.log(props.title)
  }
}
```

但是，因为 `props` 是响应式的，你**不能使用 ES6 解构**，因为它会消除 prop 的响应性。

Context: 传递给 `setup` 函数的第二个参数是 `context`。`context` 是一个普通的 JavaScript 对象，它暴露组件的三个 property：

```

export default {
  setup(props, context) {
    // Attribute (非响应式对象)
    console.log(context.attrs)

    // 插槽 (非响应式对象)
    console.log(context.slots)

    // 触发事件 (方法)
    console.log(context.emit)
  }
}

```

访问组件中的 property:

vue 2.0 生命周期对比 3.0 生命周期

2.0 周期名称	3.0 周期名称	说明
	setup	
beforeCreate	beforeCreate	组件创建之前
created	created	组件创建完成
beforeMount	onBeforeMount	组件挂载之前
mounted	onMounted	组件挂载完成
beforeUpdate	onBeforeUpdate	数据更新，虚拟 DOM 打补丁之前
updated	onUpdated	数据更新，虚拟 DOM 渲染完成
beforeDestroy	onBeforeUnmount	组件销毁之前
destroyed	onUnmounted	组件销毁后

执行 `setup` 时，组件实例尚未被创建。因此，你只能访问以下 property:

- `props`
- `attrs`
- `slots`
- `emit`

也就是无法访问: data computed methods...

但 `setup` 可以将其中定义的属性 变量 方法 返回出去.

如果 `setup` 返回一个对象，则可以在组件的模板中可以像 `props` 一样访问直接使用该对象的 property:

```

<template>
  <div>{{ readersNumber }} {{ book.title }}</div>
</template>

<script>
  import { ref, reactive } from 'vue'

  export default {
    setup() {
      const readersNumber = ref(0)
      const book = reactive({ title: 'Vue 3 Guide' })

      // expose to template
      return {
        readersNumber,
        book
      }
    }
  }

```

```
        book
    }
}
}
</script>
```

注意，从 `setup` 返回的 `refs` 在模板中访问时是被自动浅解包的，因此不应在模板中使用 `.value`。

关于this:

在 `setup()` 内部，`this` 不会是该活跃实例的引用，因为 `setup()` 是在解析其它组件选项之前被调用的，所以 `setup()` 内部的 `this` 的行为与其它选项中的 `this` 完全不同。这在和其它选项式 API 一起使用 `setup()` 时可能会导致混淆。

总结

1.Vue3的一大特性函数 ---- setup

2.setup函数是 Composition API (组合API) 的入口

3.在setup函数中定义的变量和方法最后都是需要 return 出去的 不然无法在模板中使用

SpringBoot Result对象

前后端分离项目中，后端输出Result对象并封装为Json数据，传递给前端进行处理。

从而Springboot项目中需要定义Result对象进行传递。

一：定义响应码枚举

```
/**
 * @Description: 响应码枚举，参考HTTP状态码的语义
 * @author zty
 * @date 2021/4/16 09:42
 */
public enum RetCode {
    // 成功
    SUCCESS(200),
    // 失败
    FAIL(400),
    // 未认证（签名错误）
    UNAUTHORIZED(401),
    // 接口不存在
    NOT_FOUND(404),
    // 服务器内部错误
    INTERNAL_SERVER_ERROR(500);

    private final int code;

    RetCode(int code) {
        this.code = code;
    }
    public int code() {
        return code;
    }
}
```

二：创建返回对象Result实体（泛型）

```
/**
 * @Description: 统一API响应结果封装，返回对象实体
 * @author zty
 * @date 2021/4/16 09:43
 */
public class RetResult<T> {

    public int code;

    private String msg;

    private Object data;

    public RetResult<T> setCode(RetCode retCode) {
        this.code = retCode.code;
        return this;
    }

    public int getCode() {
        return code;
    }

    public RetResult<T> setCode(int code) {
        this.code = code;
        return this;
    }

    public String getMsg() {
        return msg;
    }

    public RetResult<T> setMsg(String msg) {
        this.msg = msg;
        return this;
    }

    public T getData() {
        return data;
    }

    public RetResult<T> setData(T data) {
        this.data = data;
        return this;
    }

    public String toString() {
        // 使用FastJson 输出为json数据给前台
        return JSON.toJSONString(this);
    }

}
```

四：返回结果数据格式封装 / 响应结果生成工具

```
/**  
 * @Description: 将结果转换为封装后的对象  
 * @author  
 * @date 2021/4/16 09:45  
 */  
public class RetResponse {  
  
    private final static String SUCCESS = "操作成功";  
  
    public static <T> RetResult<T> makeOKRsp() {  
        return new RetResult<T>().setCode(RetCode.SUCCESS).setMsg(SUCCESS);  
    }  
  
    public static <T> RetResult<T> makeOKRsp(T data) {  
        return new RetResult<T>()  
            .setCode(RetCode.SUCCESS).setMsg(SUCCESS).setData(data);  
    }  
  
    public static <T> RetResult<T> makeErrRsp(String message) {  
        return new RetResult<T>().setCode(RetCode.FAIL).setMsg(SUCCESS);  
    }  
  
    public static <T> RetResult<T> makeRsp(int code, String msg) {  
        return new RetResult<T>().setCode(code).setMsg(msg);  
    }  
  
    public static <T> RetResult<T> makeRsp(int code, String msg, T data) {  
        return new RetResult<T>().setCode(code).setMsg(msg).setData(data);  
    }  
  
    public static void genHttpResult(HttpServletRequest response, Integer  
        httpCode, String msg) {  
        response.setCharacterEncoding("UTF-8");  
        response.setHeader("Content-type", "application/json;charset=UTF-8");  
        response.setStatus(httpCode);  
        try {  
            response.getWriter().write(JSON.toJSONString(msg));  
        } catch (IOException ex) {  
        }  
    }  
}
```

五：返回Result功能测试

```

@RestController
@Api(tags = "用户管理")
@RequestMapping("/users")
public class UserController {
    @PostMapping("/selectById")
    public RetResult<UserInfo> selectById(Integer id){
        UserInfo userInfo = userInfoService.selectById(id);
        return RetResponse.makeOKRsp(userInfo);
    }
}

```

前端请求返回数据格式

```
{
    "code": 200,
    "msg": "success",
    "data": {
        "id": 1,
        "userName": "1"
    }
}
```

maven pom文件配置

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <!--项目的构件标识符，版本。子项目不需要配置-->
    <modelVersion>4.0.0</modelVersion>
    <groupId>cn.xxx1</groupId>
    <version>1.0.0</version>

    <!--父项目的坐标。如果项目中没有规定某个元素的值，那么父项目中的对应值即为项目的默认值。坐标包括group ID, artifact ID和 version。-->
    <parent>
        <groupId>cn.xxxx</groupId>
        <artifactId>xxx</artifactId>
        <version>1.0.0</version>
    </parent>

    <artifactId>xxx1</artifactId>

    <properties>
        <!-- 项目字符集编码 -->
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <!-- 项目输出字符集编码 -->
        <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
        <!-- 项目java版本 -->
        <java.version>1.8</java.version>
        <!-- 项目中的XXX架包版本 -->
        <xxx.version>1.5.9.RELEASE</springboot.version>
        ...
    </properties>

```

```

</properties>
<!-- 继承自该项目的所有子项目的默认依赖信息。这部分的依赖信息不会被立即解析，而是当子项目声明一个依赖（必须描述group ID和artifact ID信息），如果group ID和artifact ID以外的一些信息没有描述，则通过group ID和artifact ID 匹配到这里的依赖，并使用这里的依赖信息。-->
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>xxx</groupId>
            <artifactId>xxx</artifactId>
            <version>xxx</version>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>xxx</groupId>
        <artifactId>xxx</artifactId>
        <version>xxx</version>
    </dependency>
</dependencies>

<!--构建项目需要的信息-->
<build>
    <!--子项目可以引用的默认插件信息。该插件配置项直到被引用时才会被解析或绑定到生命周期。
给定插件的任何本地配置都会覆盖这里的配置-->
    <pluginManagement>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.1</version>
            <configuration>
                <source>${java.version}</source>
                <target>${java.version}</target>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <!-- 热部署 -->
                <fork>true</fork>
            </configuration>
        </plugin>
    </pluginManagement>
</build>

<!-- 定义项目模块-子项目 -->
<modules>
    <module>deltaalpha-core</module>
    <module>xxx</module>
    ...
</modules>
<!--项目分发信息，在执行mvn deploy后表示要发布的位置。有了这些信息就可以把网站部署到远程服务器或者把构件部署到远程Maven仓库。-->
<distributionManagement>
    <repository>
        <id>xxx</id>

```

```
<name>xxx</name>
<url>xxxx</url>
</repository>
</distributionManagement>
</project>
```

UUID (通用唯一识别码)

UUID 目的是让分布式系统中的所有元素，都能有唯一的辨识信息，而不需要通过中央控制端来做辨识信息的指定。

UUID是指在一台机器上生成的数字，它保证对在同一时空中的所有机器都是唯一的。

标准的UUID格式为：xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx (例如8-4-4-4-12)

直接生成的ID中有“-”存在，如果不需要，可以使用replace()方法去掉。

UUID生成工具类

```
/**
 * UUID工具类
 *
 */
public class UUIDUtils {
    private static final char[] CHAR_ARR = {'A', 'B', 'C', 'D', 'E', 'F', 'G',
    'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
    'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'};

    /**
     * 生成uuid
     *
     * @return
     */
    public static String getUUID() {
        String uid = UUID.randomUUID().toString().replace("-", "");
        return uid;
    }

    /**
     * 获取随机字符串
     *
     * @param len 长度
     * @return
     */
    public static String getRandomChar(int len) {
        StringBuffer sb = new StringBuffer("");
        Random random = new Random();
        for (int i = 0; i < len; i++) {
            sb.append(CHAR_ARR[random.nextInt(CHAR_ARR.length)]);
        }
        return sb.toString();
    }
}
```

RESTful项目登录模块的实现

配置跨域访问配置类

前后端交互数据时,其双方都必须配置其对应跨域配置才可以进行相互访问数据.

SpringBoot后端配置跨域Config类:

```
@Configuration
public class Corsconfig implements WebMvcConfigurer {

    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/**")
            // 设置其允许的前段路径地址
            .allowedOriginPatterns("*")
            // 设置其允许使用的方法
            .allowedMethods("POST", "GET", "PUT", "OPTIONS", "DELETE")
            .maxAge(3600)
            .allowCredentials(true);
    }

}
```

前端配置跨域类(config/index.js)

```
dev: {
    ...
    proxyTable: {
        '/api': {
            // 设置其允许的前段路径地址
            target: 'http://localhost:8081',
            changeOrigin: true,
            pathRewrite: {
                '^/api': '/'
            }
        },
    }
}
```

实现方法

0.定义ResultCode

```
public enum ResultCode {
    SUCCESS(200),
    FAIL(400),
    UNAUTHORIZED(401),
    NOT_FOUND(404),
    INTERNAL_SERVER_ERROR(500);

    private final int code;

    private ResultCode(int code) {
        this.code = code;
    }
}
```

```

    }

    public int code() {
        return this.code;
    }
}

```

1、项目的控制层中

```

@RestController
@Api(tags = "用户管理")
@RequestMapping("/users")
public class UserController {
    @PostMapping("/login")
    @ApiOperation(value = "登录", notes = "用户登录")
    @SysLog("登录")
    public Result login(@RequestBody LoginReq req) throws Exception {
        // 获取验证码缓存
        String cacheVerifyCode = imgValidService.get(req.getValidKey());
        // 判断验证码正确性
        if(!req.getVerifyCode().equals(cacheVerifyCode)) {
            // 抛出自定义错误，其内容为发送操作失败Result。
            // 自定义错误 在其SpringMvcConfig中配置
            throw new ServiceException("验证码错误");
        }
        // userService.Login返回登陆凭证 将其登陆凭证返回Result
        return ResultGenerator.genSuccessResult(userService.login(req));
    }
}

```

2.LoginReq 登陆请求类

```

/**
 * 用于登陆请求
 *
 */
@ApiModel
public class LoginReq {

    @ApiModelProperty(value = "用户名", dataType = "String")
    private String userName;
    @ApiModelProperty(value = "用户密码", dataType = "String")
    private String password;
    @ApiModelProperty(value = "验证码", dataType = "String")
    private String verifyCode;
    @ApiModelProperty(value = "验证码验证key", dataType = "String")
    private String validKey;

    public String getVerifyCode() {
        return verifyCode;
    }
    public void setVerifyCode(String verifyCode) {
        this.verifyCode = verifyCode;
    }
    public String getValidKey() {

```

```
        return validKey;
    }
    public void setValidkey(String validkey) {
        this.validkey = validkey;
    }
    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

3.返回Result类

```
public class Result {
    private int code;
    private String message;
    private Object data;
    private Object todoTasks;

    public Result() {
    }

    public Result setResultCode(ResultCode resultCode) {
        this.code = resultCode.code();
        return this;
    }

    public int getCode() {
        return this.code;
    }

    public String getMessage() {
        return this.message;
    }

    public Result setMessage(String message) {
        this.message = message;
        return this;
    }

    public Object getData() {
        return this.data;
    }

    public Result setData(Object data) {
        this.data = data;
        return this;
    }
}
```

```

public void setCode(int code) {
    this.code = code;
}

public Object getTodoTasks() {
    return this.todoTasks;
}

public void setTodoTasks(Object todoTasks) {
    this.todoTasks = todoTasks;
}

public String toString() {
    // 使用FastJson 输出为json数据给前台
    return JSON.toJSONString(this);
}
}

```

4.ResultGenerator-Result生成类

```

public class ResultGenerator {
    private static final String DEFAULT_SUCCESS_MESSAGE = "操作成功";

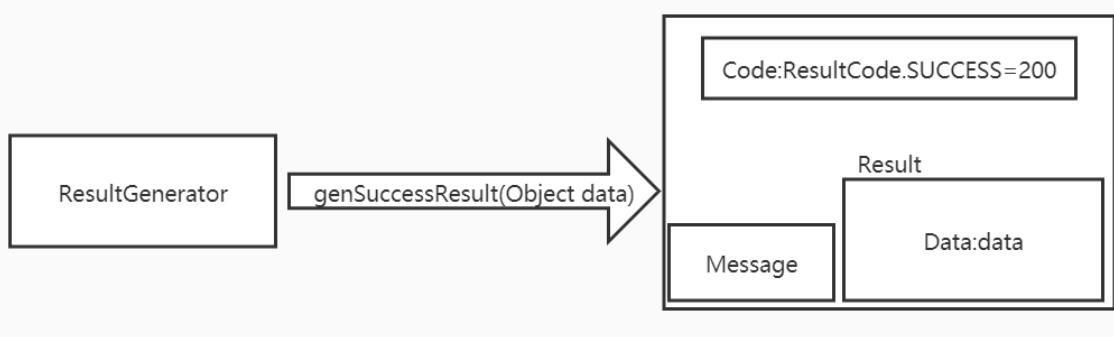
    public ResultGenerator() {
    }

    public static Result genSuccessResult() {
        return (new Result()).setCode(ResultCode.SUCCESS).setMessage("操作成功");
    }
    //生成操作成功的Result并且携带数据
    public static Result genSuccessResult(Object data) {
        return (new Result()).setCode(ResultCode.SUCCESS).setMessage("操作成
功").setData(data);
    }
    //生成操作错误的Result并且携带错误信息
    public static Result genFailResult(String message) {
        return (new Result()).setCode(ResultCode.FAIL).setMessage(message);
    }

    public static void genHttpResult(HttpServletRequest response, Integer
httpCode, String msg) {
        response.setCharacterEncoding("UTF-8");
        response.setHeader("Content-type", "application/json;charset=UTF-8");
        response.setStatus(httpCode);

        try {
            response.getWriter().write(JSON.toJSONString(msg));
        } catch (IOException var4) {
        }
    }
}

```



5.UserService中,进行登陆信息判断并返回登陆凭证.

```

@Service
@Transactional
public class UserServiceImpl extends AbstractService<User> implements
UserService {
    @Resource
    private UserMapper userMapper;
    @Autowired
    OnLineService onLineService;
    ...

    /**
     * 用户密码登陆
     */
    @Override
    public LoginResp login(LoginReq loginReq) throws Exception {
        // 从登陆请求中获取用户输入的各个信息
        String loginName = loginReq.getUserName();
        String password = loginReq.getPassword();
        if (validateUtil.isEmpty(loginName) || validateUtil.isEmpty(password)) {
            throw new ServiceException("登陆名和密码不能为空");
        }
        // 将用户输入的明文密码转换为MD5格式 (为了将其密码复杂化将其头部加入用户名)
        String enPwd = MD5Tools.MD5(loginName + password);

        // 利用TKmybatis,生成User表的Condition
        Condition con = new Condition(User.class);
        // 将其条件拼接
        // 相当于 'loginName=xxx and password=xxx and status=xx'
        // UserStatusEnum.NORMAL.getCode() 获取用户账号的状态
        con.createCriteria().andEqualTo("loginName",
            loginReq.getUserName()).andEqualTo("password",
            enPwd).andEqualTo("status", UserStatusEnum.NORMAL.getCode());

        //Tkmybatis中selectByCondition按条件查询用户
        List<User> userLt = userMapper.selectByCondition(con);

        User user = new User();
        // 判断用户是否存在
        if (validateUtil.isEmpty(userLt)) {
            throw new ServiceException("用户名或密码错误");
        } else {
            user = userLt.get(0);
        }
    }
}

```

```

// 登陆
StringBuffer sbBuffer = new StringBuffer();
sbBuffer.append(loginName);
sbBuffer.append(UUIDUtils.getUUID());
// 生成登陆凭证(为了安全话将其MD5加密)
String voucher = MD5Tools.MD5(sbBuffer.toString());

// 生成数据传输类
UserDTO userRedis = new UserDTO();
// 将用户信息复制给 数据传输类(移除密文,确保后续传输安全性).
BeanUtils.copyProperties(user, userRedis);
//利用onLineService设置用户的在线信息
onLineService.add(voucher, userRedis, null);

//登陆凭证
LoginResp resp = new LoginResp();
resp.setVoucher(voucher);
resp.setUserInfo(userCoverter.covert(userRedis));
return resp;
}
}

```

Ehcache 缓存

EhCache 是一个纯Java的进程内缓存框架，具有快速、精干等特点，是Hibernate中默认 CacheProvider。 Ehcache是一种广泛使用的开源Java分布式缓存。主要面向通用缓存,Java EE和轻量级容器。它具有内存和磁盘存储,缓存加载器,缓存扩展,缓存异常处理程序,一个gzip缓存servlet过滤器,支持REST和SOAP api等特点。

ehcache 和 redis 比较

ehcache直接在jvm虚拟机中缓存，速度快，效率高；但是缓存共享麻烦，集群分布式应用不方便。

redis是通过socket访问到缓存服务，效率比Ehcache低，但比数据库要快很多，处理集群和分布式缓存方便，有成熟的方案。

如果是单个应用或者对缓存访问要求很高的应用，用ehcache。

如果是大型系统，存在缓存共享、分布式部署、缓存内容很大的，建议用redis。

Maven引入依赖

```

<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>2.10.6</version>
</dependency>

```

主要方法:

创建一个配置文件 ehcache.xml,默认情况下Ehcache会自动加载classpath根目录下名为ehcache.xml文件，也可以将该文件放到其他地方在使用时指定文件的位置

```

<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

```

```

xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd">

<!-- 磁盘缓存位置 -->
<diskStore path="java.io.tmpdir/ehcache"/>

<!-- 默认缓存 -->
<defaultCache
    maxEntriesLocalHeap="10000"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    maxEntriesLocalDisk="10000000"
    diskExpiryThreadIntervalSeconds="120"
    memoryStoreEvictionPolicy="LRU">
    <persistence strategy="localTempSwap"/>
</defaultCache>

<!-- helloworld缓存 -->
<cache name="HelloWorldCache"
    maxElementsInMemory="1000"
    eternal="false"
    timeToIdleSeconds="5"
    timeToLiveSeconds="5"
    overflowToDisk="false"
    memoryStoreEvictionPolicy="LRU"/>
</ehcache>

```

```

// 1. 创建缓存管理器
CacheManager cacheManager =
CacheManager.create("./src/main/resources/ehcache.xml");

// 2. 获取缓存对象
Cache cache = cacheManager.getCache("HelloWorldCache");

// 3. 创建元素
Element element = new Element("key1", "value1");

// 4. 将元素添加到缓存
cache.put(element);

// 5. 获取缓存
Element value = cache.get("key1");
System.out.println("value: " + value);
System.out.println(value.getObjectValue());

// 6. 删除元素
cache.remove("key1");

Dog dog = new Dog("xiaohei", "black", 2);
Element element2 = new Element("dog", dog);
cache.put(element2);
Element value2 = cache.get("dog");
System.out.println("value2: " + value2);
Dog dog2 = (Dog) value2.getObjectValue();
System.out.println(dog2);

System.out.println(cache.getSize());

```

```
// 7. 刷新缓存  
cache.flush();  
  
// 8. 关闭缓存管理器  
cacheManager.shutdown();
```

项目整合

```
@Component  
public class EhcacheComponent {  
    private static CacheManager cacheManager = null;  
    private static Cache cache = null;  
    //设置默认缓存对象名  
    private static final String CACHE_NAME = "cache";  
    private static final int TIME_TO_LIVE_SECONDS_MAX = 60 * 60 * 24 * 365;  
  
    static {  
        initCacheManager();  
        initCache();  
    }  
  
    /**  
     * 默认过期时长，单位：秒  
     */  
    private final static int DEFAULT_EXPIRE = 60 * 60 * 24;  
    /**  
     * 不设置过期时长  
     * 本组件实际设置过期时长：TIME_TO_LIVE_SECONDS_MAX  
     */  
    private final static int NOT_EXPIRE = -1;  
  
    private static void initCacheManager() {  
        cacheManager = CacheManager.getInstance();  
    }  
  
    private static void initCache() {  
        if (null == cacheManager.getCache(CACHE_NAME)) {  
            cacheManager.addCache(CACHE_NAME);  
        }  
        if (null == cache) {  
            cache = cacheManager.getCache(CACHE_NAME);  
        }  
    }  
    /**  
     * 存放数据 带有过期时间  
     */  
    public void put(Object key, Object value, int timeToIdleSeconds) {  
        if (NOT_EXPIRE == timeToIdleSeconds) {  
            timeToIdleSeconds = TIME_TO_LIVE_SECONDS_MAX;  
        }  
  
        Element element = new Element(key, value, timeToIdleSeconds,  
            TIME_TO_LIVE_SECONDS_MAX);  
        cache.put(element);  
    }
```

```

    /**
     * 存放数据
     *
     */
    public void put(Object key, Object value) {
        Element element = new Element(key, value, DEFAULT_EXPIRE,
TIME_TO_LIVE_SECONDS_MAX);
        cache.put(element);
    }
    /**
     * 取出数据
     *
     */
    public <T> T get(Object key, Class<T> clazz) {
        Element element = cache.get(key);

        if (null != element) {
            Object value = element.getObjectValue();
            return null == value ? null : (T) value;
        } else {
            return null;
        }
    }
    /**
     * 移除数据
     *
     */
    public void remove(Object key) {
        cache.remove(key);
    }
}

```

Nuxt.js

Nuxt.js 是一个基于 Vue.js 的通用应用框架。

通过对客户端/服务端基础架构的抽象组织，Nuxt.js 主要关注的是应用的 **UI 渲染**，特点是支持服务端渲染。

安装和运行

确保安装了 npx（npx 在 NPM 版本 5.2.0 默认安装了）：

```

$ npx create-nuxt-app <项目名>

# 项目名称
$ Project name: mynuxtweb
# 项目语言
$ Programming language: JavaScript
# 项目管理工具
$ Package manager: Npm
# 项目框架

```

```
$ UI framework: None
# 项目通讯模块
$ Nuxt.js modules: Axios - Promise based HTTP client
# 项目js检验工具
$ Linting tools: (Press <space> to select, <a> to toggle all, <i> to invert selection)
# 项目测试工具
$ Testing framework: None
# 项目渲染模式
$ Rendering mode: Universal (SSR / SSG)
# 项目开发服务
$ Deployment target: Server (Node.js hosting)
# 项目开发配置工具
$ Development tools: jsconfig.json (Recommended for vs Code if you're not using typescript)
# Github名
$ What is your GitHub username? zssaer
# 项目版本管理工具
$ Version control system: None

# 运行项目
$ npm run dev

# 编译产品模式
$ npm run build
$ npm run start
```

nuxt的端口号默认为3000

修改默认端口号

package.json

```
{
  ...
  "config": {
    "nuxt": {
      // 地址
      "host": "127.0.0.1",
      // 端口
      "port": "1818"
    }
  }
  ...
}
```

项目文件

/pages	//单独网页文件夹，存放写的页面，我们主要的工作区域
/layouts	// 布局目录，用于组织应用的布局组件，不可更改。
/layouts/default.vue	//默认首页布局
/layouts/error.vue	//默认错误页面布局
	// Nuxt自动生成，临时的用于编辑的文件，build
/assets	// 用于组织未编译的静态资源入LESS、SASS 或
JavaScript	
/components	// 用于自己编写的Vue组件，比如滚动组件，日历组件，分页组件
/middleware	// 用于存放中间件
/plugins	// 用于存放JavaScript插件的地方
/static	// 用于存放静态资源文件，比如图片
/store	// 用于组织应用的Vuex 状态管理
nuxt.config.json	// 用于组织Nuxt.js应用的个性化配置，已覆盖默认配置
package.json	// npm包管理配置文件

引用文件

~ 或 @	srcDir
~~ 或 @@	rootDir

默认情况下，`srcDir` 和 `rootDir` 相同。

提示: 在您的 `vue` 模板中, 如果你需要引入 `assets` 或者 `static` 目录, 使用 `~/assets/your_image.png` 和 `~/static/your_image.png` 方式。

全局css文件设置

1.将自定义全局css文件放置到静态资源文件夹assets下

2.编辑nuxt.config.js文件

```
...
// Global CSS: https://go.nuxtjs.dev/config-css
css: [
  // 放置全局css文件
  '~/assets/css/normalize.css'
],
...
```

使用顺序: 局部CSS内容>全局CSS内容

路由功能

存放在pages下的页面将会自动添加路由跳转功能.

Nuxt.js推荐使用进行路由跳转,而不是使用[标签](#).

```
<!--使用path进行跳转-->
<nuxt-link to="/">Home</nuxt-link>
<!--使用路由name进行跳转-->
<nuxt-link :to="{name:'article'}">article</nuxt-link>
<!--使用路由name并传递参数跳转-->
<nuxt-link :to="{name:'article',params:{id:123}}">article</nuxt-link>
```

动态路由跳转

在 Nuxt.js 里面定义带参数的动态路由，需要创建对应的**以下划线作为前缀**的 Vue 文件或目录。

如: article/_id.vue

```
<li><nuxt-link :to="{name:'article-id',params:{id:123}}">Article123</nuxt-link>
</li>
```

默认模板和默认模块

默认模板

所有页面都统一附加使用该模板.适合添加页面头部内容或者页尾等重复信息

在其项目文件下新建**app.html**文件.Nuxt.js将自动识别为默认模板(父模块).

```
<html>
  <head>
    <!-- 调用Nuxt.config.js文件下的全局head属性 -->
    {{ HEAD }}
  </head>
  <body>
    <p>THIS IS TEMPLATE</p>
    <!-- 其子模块下的内容 -->
    {{ APP }}
  </body>
</html>
```

建立默认模块后,需要重启Nuxt项目.

默认布局

默认布局功能级别和默认模板相同,但区别在于 **默认布局无法改变其head部分内容,并且是由vue构建而成.**

默认布局在其项目的layout文件夹下名为 **default.vue**,默认模块项目生成时自动生成无需单独新建.

自定义404页面

在Nuxt下404页面可以自定义设置.

在其layout文件夹下新建error.vue文件,即为自定义404页面

Front-End 前端设计

元素居中,页面锁定大小

```
html {  
  block-size: 100%;  
  inline-size: 100%;  
}  
  
body {  
  min-block-size: 100%;  
  min-inline-size: 100%;  
  margin: 0;  
  display: grid;  
  place-content: center;  
  font-family: system-ui, sans-serif;  
}
```

position(定位)

position 属性规定应用于元素的定位方法的类型。

有五个不同的位置值：

- static
- relative
- fixed
- absolute
- sticky

元素其实是使用 top、bottom、left 和 right 属性定位的。但是，除非首先设置了 position 属性，否则这些属性将不起作用。

根据不同的 position 值，它们的工作方式也不同。

static:静态定位

HTML 元素默认情况下的定位方式为 static (静态)。

静态定位的元素不受 top、bottom、left 和 right 属性的影响。

relative:相对定位

使用relative定位的元素相对于其正常位置进行定位。

顾名思义,相对定位就是相对于一个状态定位,其结果就会是相反,如一个元素 使用left:5px,那么 这个元素就会向右移动5个像素。.

设置相对定位的元素的 top、right、bottom 和 left 属性将导致其偏离其正常位置进行调整。

不会对其余内容进行覆盖,会自动适应元素留下的任何空间。

fixed:固定定位

相对于视口(浏览器)定位的,这意味着即使滚动页面,它也始终位于同一位置。

使用 top、right、bottom 和 left 属性用于定位此元素。

固定定位的元素不会在页面中通常应放置的位置上留出空隙。

由于特性,通常用在固定顶部的导航栏上等.

absolute:绝对定位

相对于最近的定位祖先元素进行定位 (而不是相对于视口定位,如 fixed)。

直接指定固定位置进行定位.会覆盖其他元素,不会实现自适应.

然而,如果绝对定位的元素没有祖先,它将使用文档主体 (body),并随页面滚动一起移动。

注意:“被定位的”元素是其位置除 static 以外的任何元素。

这个 <div> 元素设置了 position: relative;

这个 <div> 元素设置了 position: absolute;

sticky:粘性定位

元素根据用户的滚动位置进行定位。

粘性元素根据滚动位置在相对 (relative) 和固定 (fixed) 之间切换。起先它会被相对定位,直到在视口中遇到给定的偏移位置为止 - 然后将其“粘贴”在适当的位置 (比如 position:fixed)。

注：IE/Edge 15 以及更早的版本不支持粘性定位。Safari 需要 -webkit- 前缀（请参见下面的实例）。您还必须至少指定 **top**、**right**、**bottom** 或 **left** 之一，以便粘性定位起作用。

```
div.sticky {  
    position: -webkit-sticky; /* Safari */  
    position: sticky;  
    top: 0;  
    background-color: green;  
    border: 2px solid #4CAF50;  
}
```

通常使用在动态导航栏上等，移动到元素后会自动固定位置并跟随滚动。

重叠元素

在对元素进行定位时，它们可以与其他元素重叠。

z-index 属性指定元素的堆栈顺序（哪个元素应放置在其他元素的前面或后面）。

元素可以设置正或负的堆叠顺序：

```
img {  
    position: absolute;  
    left: 0px;  
    top: 0px;  
    z-index: -1;  
}
```

上述表示该元素在最低下。具有较高堆叠顺序的元素始终位于具有较低堆叠顺序的元素之前。

注意：如果两个定位的元素重叠而未指定 **z-index**，则位于 HTML 代码中最后的元素将显示在顶部。

浮动和清除

float 属性

float 属性用于定位和格式化内容，例如让图像向左浮动到容器中的文本那里。

float 属性可以设置以下值之一：

- **left** - 元素浮动到其容器的左侧
- **right** - 元素浮动在其容器的右侧
- **none** - 元素不会浮动（将显示在文本中刚出现的位置）。默认值。
- **inherit** - 元素继承其父级的 **float** 值

最简单的用法是，**float** 属性可实现（报纸上）文字包围图片的效果。

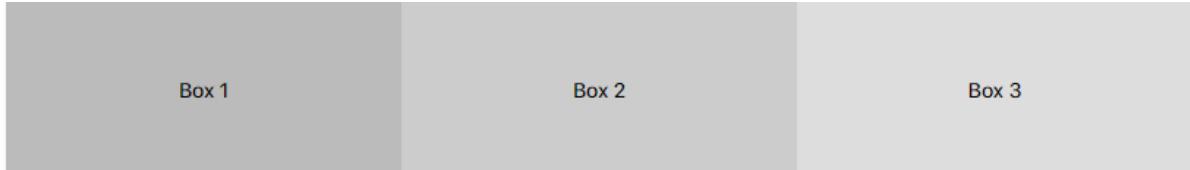
```
img {  
    float: right;  
}
```

在本例中，图像会在段落中向右浮动，而段落中的文本会包围这幅图像。

领先的 Web 技术教程 - 全部免费。找到你所需要的所有的网站建设教程。从基础的 HTML 到 CSS，乃至进阶的 XML、SQL、JS、PHP。我们的参考手册涵盖了网站技术的方方面面。其中包括W3C标准技术：HTML、CSS、XML。以及其他技术，诸如 Javascript、PHP、SQL 等。在 W3School，我们提供上千个实例。通过使用我们的在线编辑器，你可以编辑这些例子，并对代码进行实验。



通过使用 float 属性，也可以轻松地并排浮动内容框：



```
* {
  box-sizing: border-box;
}

.box {
  float: left;
  width: 33.33%; /* 三个框（四框使用 25%，两框使用 50%，以此类推） */
  padding: 50px; /* 如果需要在图片间增加间距 */
}
```

clear 属性

指定哪些元素可以浮动于被清除元素的旁边以及哪一侧。

clear 属性可设置以下值之一：

- none - 允许两侧都有浮动元素。默认值
- left - 左侧不允许浮动元素
- right - 右侧不允许浮动元素
- both - 左侧或右侧均不允许浮动元素
- inherit - 元素继承其父级的 clear 值

使用 clear 属性的最常见用法是在元素上使用了 float 属性之后。

如果一个元素比包含它的元素高，并且它是浮动的，它将“溢出”到其容器之外：

然后我们可以向包含元素添加 overflow: auto;，来解决此问题

```
.clearfix {
  overflow: auto;
}
```

`overflow: auto` 的 clearfix 类：



请为包含元素添加一个带有 `overflow: auto;` 的 `clearfix` 类，以解决此问题：

`overflow: auto` 的 clearfix 类：



只要您能够控制外边距和内边距（否则您可能会看到滚动条），`overflow: auto` `clearfix` 就会很好地工作。但是，新的现代 `clearfix` hack 技术使用起来更安全，以下代码被应用于多数网站：

```
.clearfix::after {  
    content: "";  
    clear: both;  
    display: table;  
}
```

Flexbox 布局模块

Flexbox是CSS3的新布局.

弹性框布局模块，可以更轻松地设计灵活的响应式布局结构，而无需使用浮动或定位。

它可以自动拉伸框使其与最长的框一样长.

它在 Internet Explorer 10 或更早版本中不起作用。

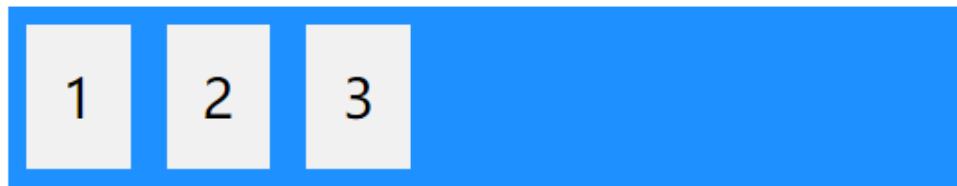
定义 Flex 容器:

弹性布局中必须有一个 `*display*` 属性设置为 `*flex*` 的父元素。

```
.flex-container {  
    display: flex;  
    background-color: DodgerBlue;  
}
```

```
<div class="flex-container">  
    <div>1</div>  
    <div>2</div>  
    <div>3</div>  
</div>
```

弹性容器的直接子元素会自动成为弹性项目。



弹性布局中必须有一个 `display` 属性设置为 `flex` 的父元素。

弹性容器的直接子元素会自动成为弹性项目。

TweenMax(GSAP)动画

TweenLite/TweenMax是GreenSock 动画平台中的核心动画工具，可用来构建补间动画(tween)。

引用/导入

CDN: `<script src="https://cdnjs.cloudflare.com/ajax/libs/gsap/3.6.1/gsap.min.js"></script>`

NPM: `npm install gsap`

默认（主）文件是**gsap.js**，其中包括大多数**缓动**以及核心插件（如 CSSPlugin、AttrPlugin、SnapPlugin、ModifiersPlugin）以及所有[实用方法](#)（如 interpolate()、mapRange() 等）

```
// typical import
import gsap from "gsap";

// or get other plugins:
import Draggable from "gsap/Draggable";
import ScrollTrigger from "gsap/ScrollTrigger";
```

TweenLite和TweenMax区别

我们的API文档大部分以TweenMax为例。如果你使用的是简约版的TweenLite，把TweenMax改为TweenLite即可。

例如`.to()`方法是公用的

TweenMax对象`.play()`

```
tween = new TweenMax('.box', 3, {
  x: 500,
  paused: true
});
playBtn.onclick = function() {
  tween.play();
}
```

执行当前对象动画

动画结构

TweenMax.to()

此方法用于创建一个从当前属性到指定目标属性的TweenMax动画对象。

```
TweenMax.to('#title', 5, {x:300}); //ID为title的对象5秒钟内向右移动300px
```

接收参数:

参数名	类型	是否必填	描述
target	object	是	需要动画的对象
duration	number	是	动画持续时间, 一般是秒
vars	object	是	动画参数 (CSS属性、延迟、重复次数等)

TweenMax.from()

通过设置动画起始点来初始化一个TweenMax, 相当于动画从设置点开始。

```
TweenMax.from('#title', 1.5, {opacity:0, delay:2});
```

接收参数同上

TweenMax.fromTo()

通过设置动画起始点和结束点来初始化一个TweenMax, 相当于动画从设置点到结束点。

```
TweenMax.fromTo([element1, element2], 1, {x:200}, {x:500});
```

接收参数:

参数名	类型	是否必填	描述
target	object	是	需要动画的对象
duration	number	是	动画持续时间, 一般是秒
fromVars	object	是	起始点动画参数
toVars	object	是	结束点动画参数

TweenMax.staggerTo()

stagger系列方法为多个目标制作一个有间隔的动画序列，相当于多个TweenMax的数组。需要设置每个动画的开始间隔。如不设置则为零，同时开始动画。

```
var objects = [obj1, obj2, obj3, obj4, obj5];
TweenMax.staggerTo(objects, 1, {y:"+=150", opacity:0}, 0.2);
```

接收参数：

参数名	类型	是否必填	描述
targets	Array	是	要进行动画的对象，可以有多个，以数组形式传入
duration	number	是	动画持续的秒数（或帧数，如果设置了useFrames:true）
vars	object	是	设置动画的一些属性及其他参数
stagger	Number	否	每个动画的起始间隔，默认是0
onCompleteAll	Function	否	当所有显示对象都完成动画后要调用的函数
onCompleteAllParams	Array	否	onCompleteAll函数的参数，以数组形式传入
onCompleteAllScope		否	onCompleteAll函数的作用域，this

注意：stagger系列方法可用于TweenMax、TimelineLite、TimelineMax，不可用于TweenLite。

TweenMax.staggerFromTo()

通过设定序列动画的起点和终点来初始化一个TweenMax。

```
var objects = [obj1, obj2, obj3, obj4, obj5];
TweenMax.staggerFromTo(objects, 1, {opacity:1}, {opacity:0}, 0.2);
```

接收参数同上

TweenMax.delayedCall()

提供一种在设定的时间（或帧）后调用函数的简单方法。

```
//1秒后执行myFunction并传递两个参数：
TweenMax.delayedCall(1, myFunction, ["param1 value", "param2
value"], document, true);
function myFunction(param1, param2) {
    console.log(param1+param2+this)
}
```

接收参数：

参数名	类型	是否必填	描述
delay	Number	是	要延迟的秒数（或帧数，如果设置了useFrames:true）
callback	Function	是	要延迟执行的函数
params	Array	否	传递给onComplete函数的参数，以数组形式传入
scope	*	否	函数的作用域，this的指向，默认为空
useFrames	Boolean	否	设定延迟的时间模式是基于秒数还是帧数，默认false：秒

TweenMax.set()

立即设置目标的属性值而不产生过渡动画，相当于0的动画时间。返回TweenMax对象。

```
//以下两个设置作用相同
TweenMax.set(myObject, {x:100});
TweenMax.to(myObject, 0, {x:100});
```

接收参数:

参数名	类型	是否必填	描述
target	object	是	需要移动的对象
vars	object	是	动画参数

3D效果

在平时的动效开发中，为了使动效具有立体的效果，一般会用到CSS3中的3D transform这一属性。在TweenMax中也提供了3D transform功能，支持CSS3D的全部属性，使用起来比CSS3更加方便。

perspective和**transformPerspective**两个属性。它们是TweenMax中运行的基础，使用它们才能使元素具有一个3D空间透视的表现能力。

transformPerspective是针对单个元素使用的。

```
TweenMax.set(".nbox", {transformPerspective:300}); //为nbox直接设置3D效果
```

而perspective则是使用在父级元素上，使用它会使该父级元素下的子元素具有3D空间透视的一个展现形式。所以只需要在父级使用perspective的方法，可以同时使它的子元素都具有3D的展现。

transformOrigin :用来设置元素在做transform位移变换时的原点的。默认值是元素的中心点即("50%,50%")。

transformOrigin有三个值(x,y,z)，z值是一个可选项。也可以使用"top", "left", "right",或者"bottom"值亦或是百分值

```
TweenMax.to('.box', 3, {rotationY:360, transformOrigin:"left top"})
```

动画参数

repeat

动画在第一次完成后应重复的次数。

例如，如果repeat为1，则动画将总共播放两次（初始播放加1次重复）。要无限期重复，请使用-1。repeat应该始终是一个整数。

```
TweenMax('.box', 3, {
  x: 500,
  repeat: 3, //动画将进行4次
});
```

delay

动画开始之前的延迟秒数（以帧为单位时则为帧数）。

```
TweenMax('.box', 3, {
  x: 500,
  delay: 3 //延迟三秒开始动画
});
```

实例属性

.timeline

获取动画的父级时间轴对象（只读）。

```
mytween = TweenMax.to('.box', 3, {x:500});
console.log(mytween.timeline);
```

每个动画都放在时间轴上（默认为根时间轴），并且只能有一个父时间轴。一个实例不能同时存在于多个时间轴中。

Locomotive Scroll(视差滚动)

视差滚动简单的认识就是移动滚动条或者滚动鼠标滑轮，dom盒子移动的速度加减的差别造成页面的不同效果。

GitHub项目地址: <https://github.com/locomotivemtl/locomotive-scroll>

安装\引用

```
npm install locomotive-scroll
<!--引用-->
<script src="https://cdn.jsdelivr.net/npm/locomotive-
scroll@3.5.4/dist/locomotive-scroll.min.js"></script>
```

使用

属性	描述
data-scroll-container	定义容器
data-scroll	检测是否在视野中
data-scroll-speed="number"	元素视差速度,负数反转
data-scroll-target="#xxx"	目标元素在视图中的位置
data-scroll-direction="vertical horizontal"	视差的方向

```
<div data-scroll-container>
  <div data-scroll-section>
    <h1 data-scroll>Hey</h1>
    <p data-scroll>👋</p>
  </div>
  <div data-scroll-section>
    <h2 data-scroll data-scroll-speed="1">What's up?</h2>
    <p data-scroll data-scroll-speed="2">🌐</p>
  </div>
</div>
```

添加Locomotive Scroll Css文件

```
https://github.com/locomotivemtl/locomotive-scroll/blob/master/dist/locomotive-
scroll.css
```

js:

```
import LocomotiveScroll from 'locomotive-scroll';

const scroll = new LocomotiveScroll({
  el: document.querySelector('[data-scroll-container]'),
  smooth: true
});
```