# 账号

本地虚拟机账号

root 000000

test Jsy123..,,

192.168.0.108

学信网 18518505760 890812jsy

GIT maixialingyi 890812jsy@ QQ邮箱

百度网盘 18518505760 19890812jsy

QQ 2602043912 890812jsy

瑞友邮箱 jiangshaoyue0710@rayootech.com JsY@18518505760

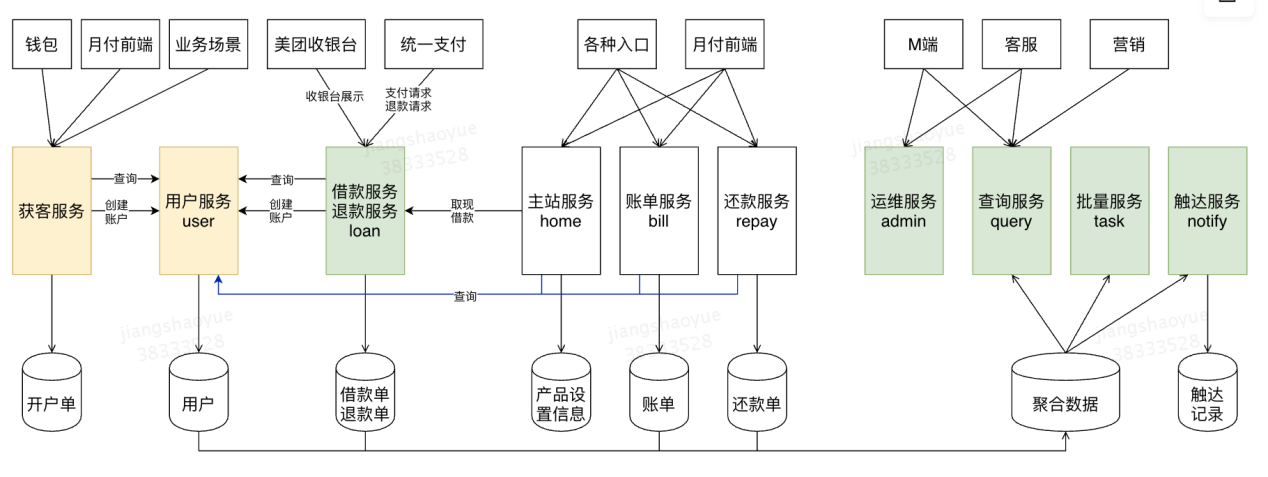
个税APP 18518505760 890812jsy@

Oracle 18518505760@163.com 19890812Jsy

驾驶证 19890812jsy@

京东账号 000000jsy@JSY

# 美团项目设计文档





## 架构文档：



## 业务流程

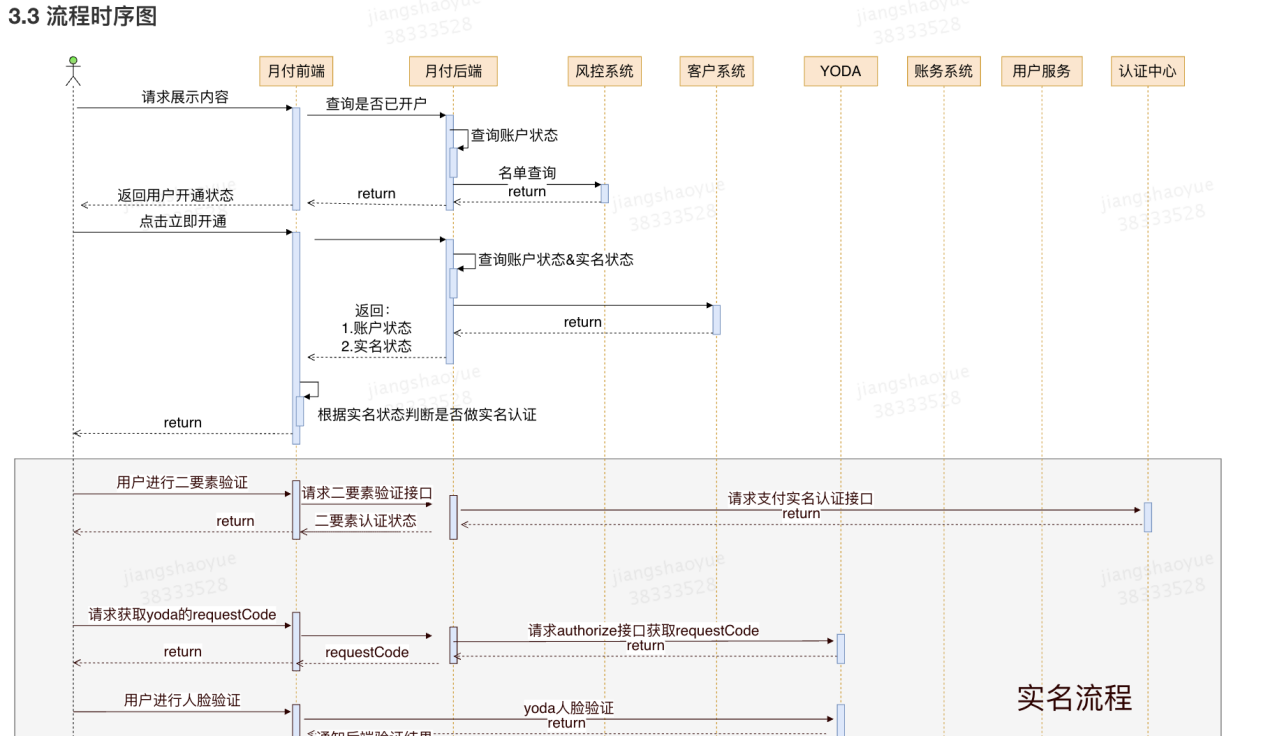
### 获客服务

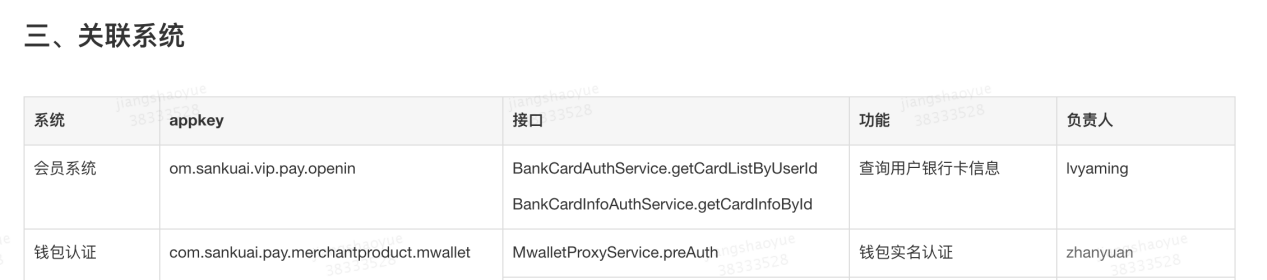






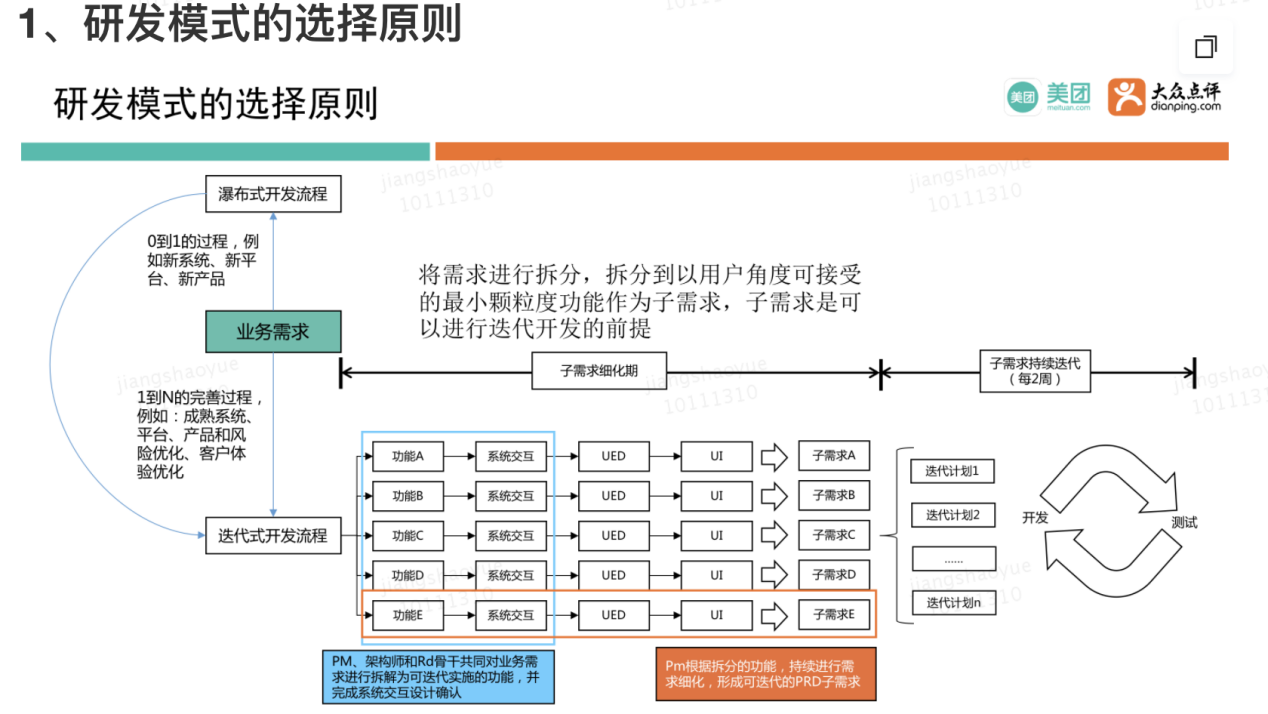


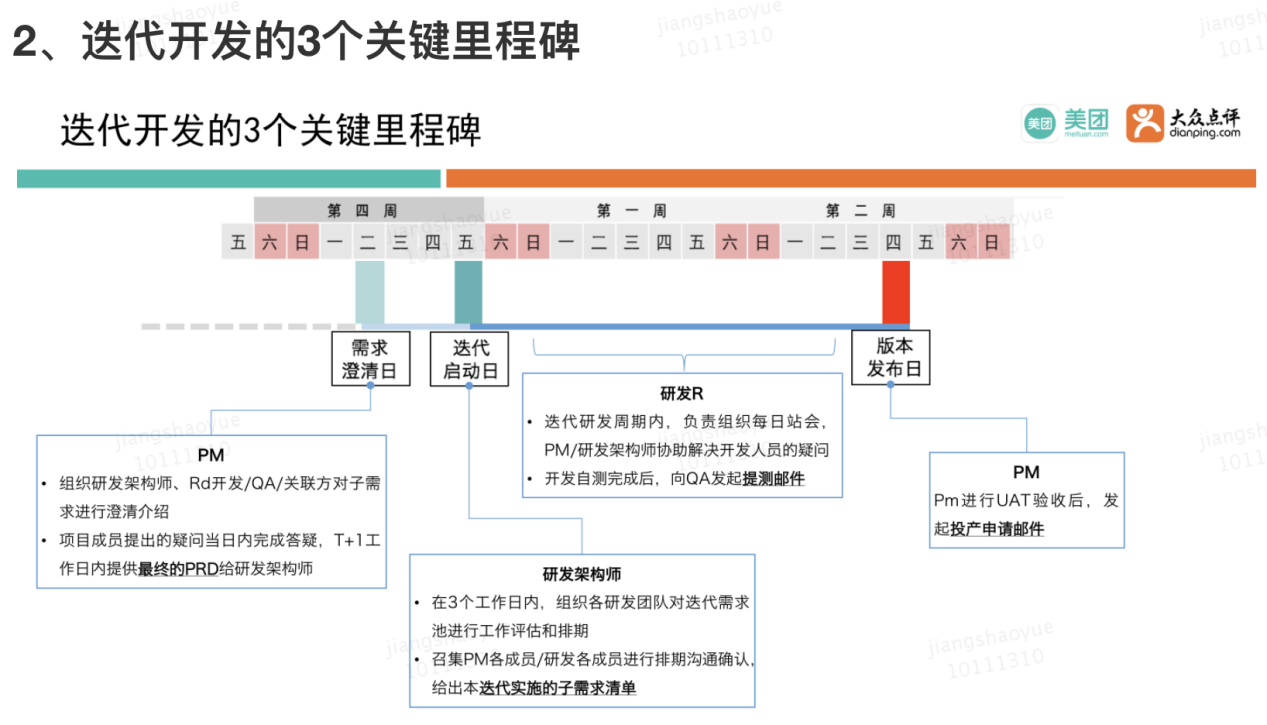


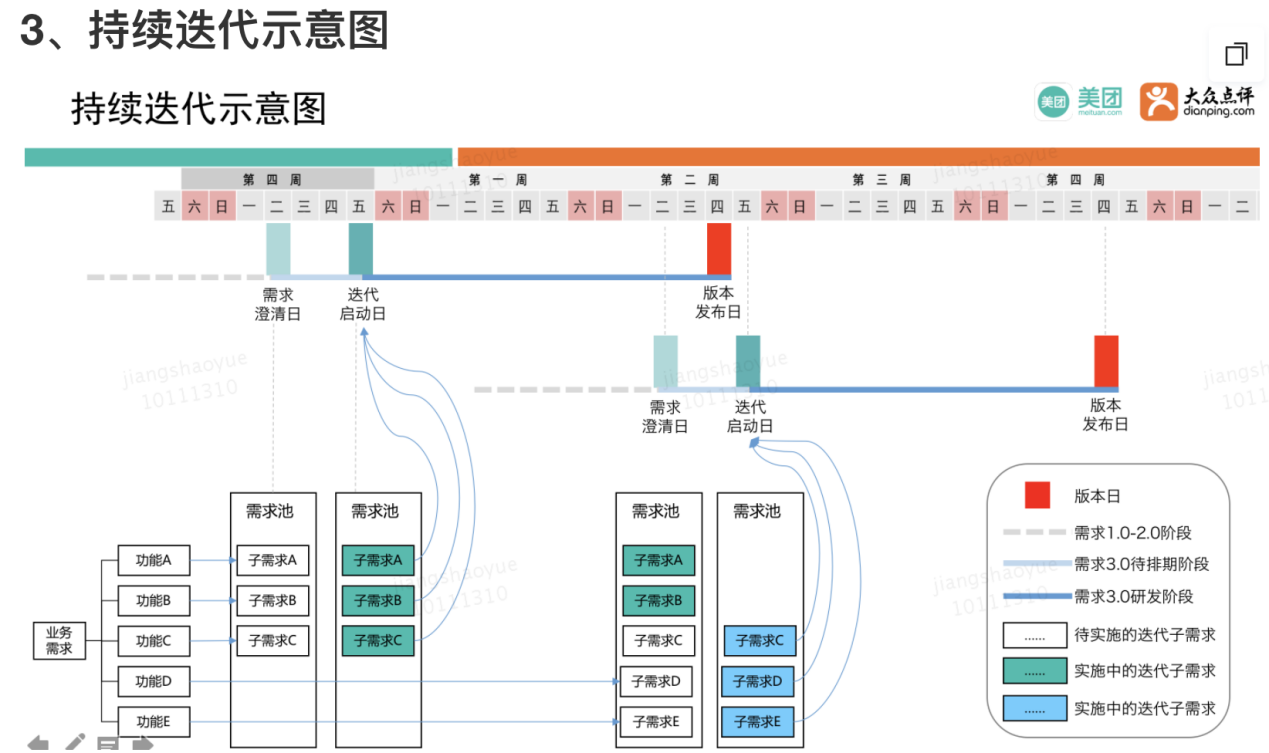


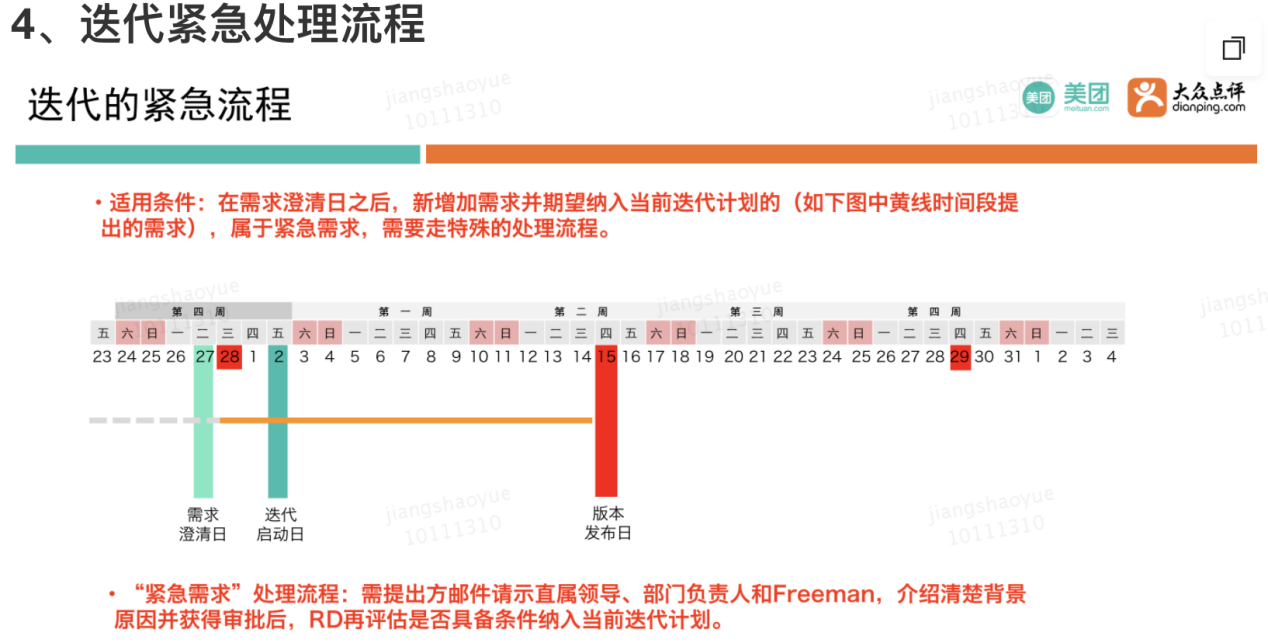
# 研发流程规范

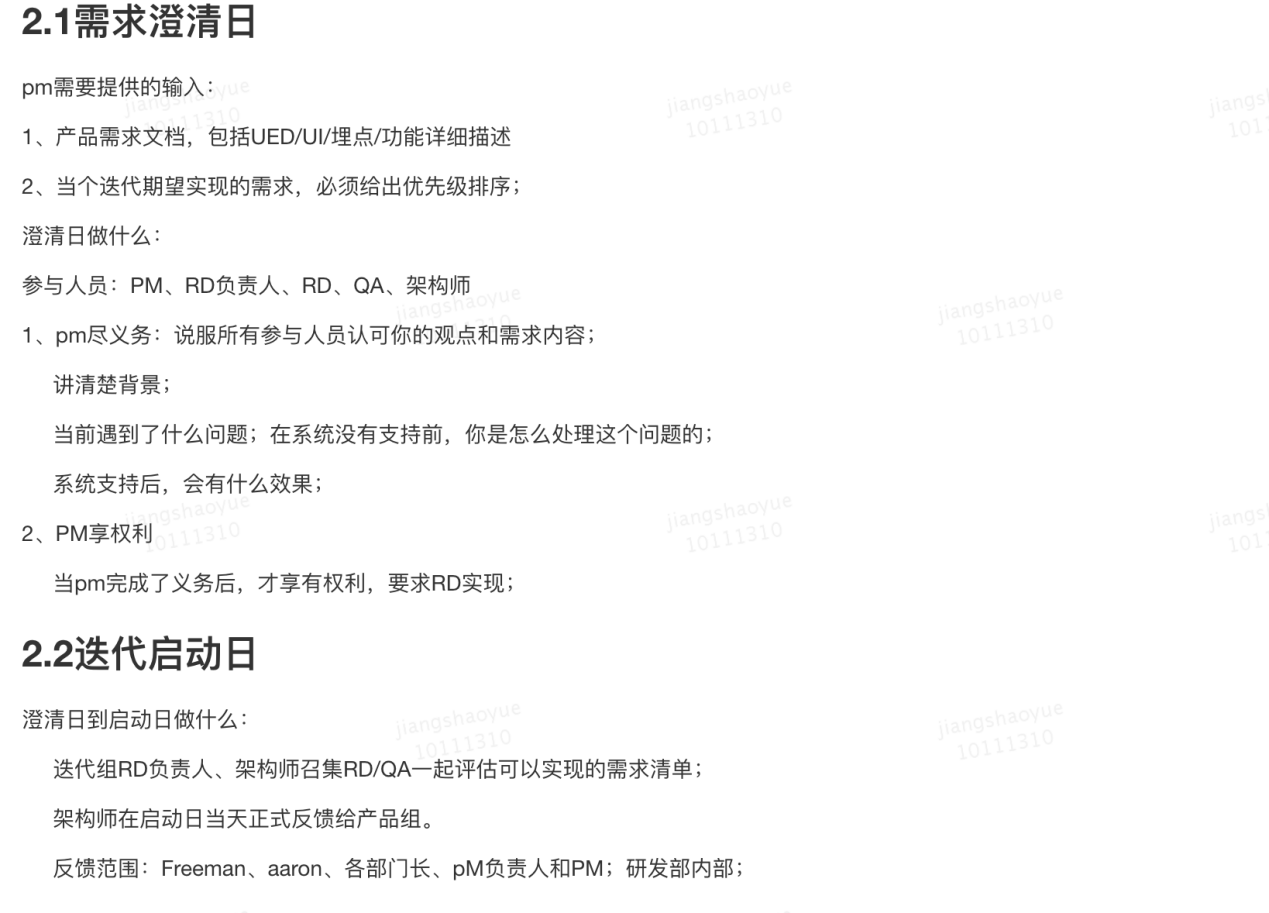
## 迭代需求&项目研发流程











## 需求质量记录

一 需求澄清时PRD评估

研发友好度：

1）纯后端的需求，是否有业务流程图？业务流程图是否正确？

2）涉及多个业务系统的，有没有系统交互逻辑？

3）涉及前端的需求，有没有将页面交互和业务逻辑结合起来描述？

4）描述用词是否专业？是否有名词解释？是否出现了火星文（除了自己能看懂，别人都不懂）？

5）PRD是否将改动处的线上逻辑描述清楚，是否将改动后的业务逻辑描述清楚？

逻辑完整度：

1）业务逻辑的各种分支是否都枚举出来，没有遗漏？

2）业务逻辑是否与现存逻辑兼容？

3）业务逻辑的变更是否够平滑，不至于导致用户的感知较为跳跃？

4）业务逻辑的定义是否足够清晰，并考虑到未来可能的拓展情况？

业务友好度：

1）需求价值的衡量是否有逻辑推理过程 ？

2）需求目标的设定逻辑是否成立？

3）需求效果复盘计划是否完整？

用户友好度：

1）是否有明显的体验缺陷？

2）是否考虑到客诉场景，并给客服提出需求

## 如何进行技术方案评审？

一 技术方案前做什么？

需求分析是否完成？新需求对原有的业务逻辑的影响是什么？影响点是否总结完毕？

二 技术方案评审评什么？

1 功能性需求-完整性

业务功能的逻辑分支覆盖率，是否覆盖了所有的业务逻辑分支？是否有兜底的业务处理分支？

2 非功能性需求-广度和深度

2.1 高内聚、低耦合

业务系统划分成那些服务？服务和服务之间边界是否清晰？是否做到了高内聚低耦合？

高内聚、低耦合：领域内高内聚、领域间低耦合；模块内高内聚、模块间低耦合；服务内高内均、服务间低耦合；

核心逻辑是：一个领域、模块、服务，应该隐藏领域内逻辑，领域内逻辑不外泄；同时不应该感知或实现不属于自己的范围的业务逻辑；

特别注意的就是系统间的交互方案、接口定义（单一职责：单一功能、单一场景）

2.2 扩展性

一个服务内部不同业务流程之间，是否可以抽象、复用？

一个业务流程是否通过良好的设计具备了一些扩展性？

2.3 健壮性、可用性

一个业务流程对依赖的服务（技术基础服务、业务基础服务、其他业务服务）是否具备容错性？是否可以容忍他们短时间的不可用？

依赖服务不可用性时 ，是否可以继续提供有损服务，做到柔性可用？

服务的调用方都有哪些？性能要求分别是怎样的？采用怎样的技术方案来保证性能？

不用调用方的容量需求是怎样的？采用怎样的技术方案来保证容量？

2.4 一致性&故障处理

故障时，是否可以保证业务数据一致性？无法保证一致性时，对业务的影响是什么？是否可以不影响业务？

故障后，是否可以自动快速的修复业务数据，使得业务数据一致。

2.5 运维成本

上线后人工运维的成本是怎样的，是否合理？

2.6 监控报警

业务流程是否具备了完备的监控报警机制？是否可以及时发现系统故障、业务故障？报警是否能做到不重不漏？报警是否可以定位故障原因？

## 如何进行需求澄清？

一 需求澄清前的工作有哪些？

大型需求，PM在澄清之前把PRD发到相关RD，供提前阅读，以提高需求澄清效率。

需求澄清之前PM创建ones任务状态：待澄清；指派人：PM自己；需求类型：产品需求

需求澄清后，如果RD&FE&QA都没有待确认问题，则需求的技术主R将ones的需求状态改为需求已经澄清。

如果需求本身有待确认问题、或RD&FE&QA有业务逻辑需要PM确认，PM不能不及时确认，则需求的技术主R将ones的需求状态改为澄清不通过。澄清不通过的需求，PM'需要再次组织需求澄清。

常规迭代需求，自确认需求已经澄清的3个工作日内给出研发排期。专项需求也尽量保证自需求澄清一周内给出研发排期。

二 需求澄清的内容有哪些？

需求产生的业务背景是什么？

需求不做有什么问题？

需求做了有哪些业务收益？业务收益的衡量逻辑是什么？

需求的业务收益如何复盘？后续如何迭代和演进？注意：区分清楚需求迭代与需求bugfix的区别，需求bugfix不属于业务迭代。

需求都有哪些内容？大体的逻辑是什么？

需求都涉及哪些系统、哪些服务？

涉及系统之间的需求逻辑分工是怎样的？

需求都涉及哪些页面？

进入页面和跳出页面的逻辑是怎样的？

页面内部的逻辑都有哪些？逻辑是否完整？

页面之间的交互是怎样的？

页面之间的交互逻辑是怎样的？

涉及页面的需求描述是否图文结合、通俗易读？

纯后端的需求，业务流程是否描述清楚？

所有需求变更，都需要先描述清楚涉及业务模块的线上现状是怎样的？本次需求的改动点是什么？改动后的整体逻辑是怎样的？

每次需求确认澄清后，主R的RD应该确保：其他没有参与本次需求的同学可以通过本PRD了解清楚需求背景、价值、目标、需求改动前线上现状、需求改点是什么、需求改动后的整体逻辑是什么。

## 如何主R一个项目？

1. 【首先理清需求】不明确的需求点跟产品确认好；如果是已有功能则需要了解现状再结合需求确认需求是否有问题，有疑问的找相关同学确认。

2. 【确定技术方案】明确需求后自己设计出大体技术方案，如果是老业务可以先跟老同学确认，有外部关联方时再跟外部关联方RD确认。

3. 【拆分需求对应功能接口&评估出工时】技术方案确定之后设计出技术接口文档，包括接口参数，SLA参数，有关联方时需要找关联方再次确认接口参数等技术细节，有出入则沟通解决并修改文档。评估工时先按照自己全力投入的工时评估再加一定buffer, 有其它并行工作时需要考虑进去。

4. 【内部技术评审】技术文档确认之后，提行事历拉产品测试同学进行技术评审，有问题再次修改技术文档

5. 【输出整个项目各个环节明确的时间点及责任人】技术评审之后需跟进QA和关联方的具体的时间点，时间点包括：假接口给出时间，开发完成时间，联调时间，提测时间，QA测试时间点，st验收时间，上线前灰度时间，上线时间

6. 【周知所有关联方整体排期】

7. 【开发+自测】开发过程中有风险问题及时反馈

8. 【用例评审】

9. 【联调】

10. 【四方验收】

11. 【提测】

12. 【上线前checklist评审】拉QA和关联方确认线上步骤

13. 【上线前资源申请】包括sql工单，mcc配置，消息topic申请，鉴权申请，紧急发布邮件审批

14. 【灰度上线】

15. 【正式上线】

## 项目主R的职责是什么？

1. 【理清&确认项目需求点】

2. 【推动技术方案落地】

3. 【评估项目整体时间点】

4. 【了解掌握项目进度及风险】

5. 【关注项目质量并跟进反馈】

6. 【上线流程梳理】

https://km.sankuai.com/page/292365898

## 开发流程

需求澄清 ：有问题提出来，也可会后再处理，但一定要明确需求

技术文档 ：复杂逻辑要有流程图，影响范围，与对接人确认

技术评审 ：产品，测试，前端，对接方 核对讲解技术细节，影响范围以及风险点

排期时间 ：开发，自测，联调，code review，提测，上线（配置，依赖服务上线时间）

开发 ：无法按时完成的也要提前反馈

联调 ：需联调方确认

测试用例 ：核对测试提供的用例文档，确认是否覆盖完整，是否逻辑一致

测试阶段 ：当天bug当天处理完

上st前 ： 确认sql，配置等脚本，验收账号 各方确认

st验证

上线前 ：确认sql，配置等脚本，验收账号 确认依赖方已准时上线

# java数据结构

指数:幂运算b = aⁿ (a≠0),a为底数，n为指数

对数:指数函数的反函数记做n=logab (a>0 , a !=1) https://gss2.bdstatic.com/9fo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D9/sign=fbee003833dbb6fd215be917091653/279759ee3d6d55fb18cfc2e567224f4a21a4ddec.jpg 叫做“以 https://gss0.bdstatic.com/94o3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D8/sign=3e92cbc5a6af2eddd0f144d88d8afa/d53f8794a4c27d1e9a808e0711d5ad6edcc4384a.jpg 为底 https://gss2.bdstatic.com/9fo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D13/sign=e77a8a8808087bf479ec53eaf0d38797/37d12f2eb9389b50b52d55918f35e5dde6116e27.jpg 的对数”

**大O表示法:粗略的度量方法,如说一辆车只说大和小,不比尺寸**

算法的速度会根据数据项(数据量)而变化,用大O描述算法的速度如何与数据项的个数相联系的比较

## 数组

1.有序的(意思是存放顺序同放入顺序)

2.并且大小不可变

3.基本数据类型放的是值,对象类型放的是引用

4.默认值:基本数据类型放各自默认值,对象类型为null

5.删除:非真正删除即赋空值,无洞原则即后向前移动

6.使用数组为继承结构的数据结构,都遵循没有洞(会增加操作复杂度)

7.查询:下标取最快,下标未知(值无序遍历,值有序二分)

### 二分查找代码:

|  |
| --- |
| //有序数组且值不重复，查找值下标  int binarySearch(int[] nums, int target) {  int left = 0;  int right = nums.length - 1; // 注意  while(left <= right) { // 注意  int mid = (right + left) / 2;  if(nums[mid] == target)  return mid;  else if (nums[mid] < target)  left = mid + 1; // 注意  else if (nums[mid] > target)  right = mid - 1; // 注意  }  return -1;  } |

### 冒泡排序:

遵循规则:

1. 比较相邻两个
2. 如果左边大,则交换位置
3. 向右移动一个位置,比较下面两个

数组有N个元素,从左开始,

第一次循环下标0跟1比较,如果左大于右换位,小于或等于不变,操作向右移一位,最终把最大的移动到最右边,操作N-1次

第二次循环把第二大的移动到倒数第二位,操作N-2次

………..

|  |
| --- |
| **int** out, in;  **for** (out = arr.length - 1; out > 1; out--) {  **for** (in = 0; in < out; in++) {  **if** (arr[in] > arr[in + 1]) {  **int** temp = arr[in];  arr[in] = arr[in + 1];  arr[in + 1] = temp;  }  }  } |

不变性:每次开始循环时,本次循环的结束下标(out)后的数据都是不变有序的

大O:

有N个数据项,第一次比较N-1次 类推……..

比较次数 = (N-1)+(N-2)+(N-3)+…+1=N\*(N-1)/2 约 N2/2 (1可以忽略尤其N很大时)

交换次数是随机的大概为一半

交换次数 = N2/4

比较和交换次数都和N2 成正比,大O表示法中常数不算,可忽略2,4

🡪O(N2)

只要两循环嵌套,都可以怀疑为O(N2)级 (外循环N次,内循环N次或N/x)

### 选择排序:

定义一个变量为最小值min 放最小值下标

第一次循环,先假设左第一个为最小值,依次与其后比较,如果被比较值小放入min,第一次循环完毕,交换arr[0] 与arr[min] 值,此时最左边为最小值

第二次循环从左第二位开始,步骤同上

………..

|  |
| --- |
| **int** out, in, min;  **for** (out = 0; out < arr.length - 1; out++) {  min = out;  **for** (in = out + 1; in < arr.length; in++) {  **if** (arr[in] < arr[min]) {  min = in;  }  }  **int** temp = arr[out];  arr[out] = arr[min];  arr[min] = temp;  } |

不变性:每次开始循环时,开始位置前的顺序是有序固定的

🡪O(N2)

选择排序与冒泡排序比较次数相同,但交换次数要少很多只需N此交换

### 插入排序:

****

选择排序是向未排序的方向遍历比较,插入排序是向已经排序的方向比较

用变量存储标记数据项,未找到被比较项右移,比较到不大于时插入并停止

第一次:从第二位开始,先把第二位值那出来放入变量out,与第一位比较如果小于等于第一位,第一位向后移动一位,插入标记数据--in 下标为0 while条件不成立跳出循环

第二次:从第三位开始,同上

|  |
| --- |
| **int** in, out;  **for** (out = 1; out < arr.length; out++) {  **int** temp = arr[out];  in = out;  **while** (in > 0 && arr[in - 1] >= temp) {  arr[in] = arr[in - 1];  --in;  }  arr[in] = temp;  } |

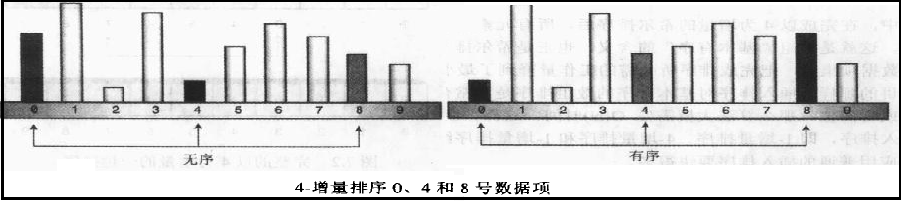
不变性:标记位置前都是局部有序的

🡪O(N2)

局部有序的情况下,比较次数,交换次数看有序情况

### 希尔排序

插入排序是相邻比较,比较元素间隔为1,基本有序数据越多,排序越快,希尔排序是先用大间隔排序,造成基本有序,在用插入排序即间隔为1



间隔序列公式 h=h\*3+1 1 4 13 40 131

取小于数组长度的最大值 h<arr.length,

比如100长度的数组,40,13,4,1依次排序,为1时即插入排序 (间隔1)

|  |
| --- |
| public static void shellSort(long[] arr) {  int inner, outer;  long temp;  int h = 1; //生成最大间隔  while (h <= arr.length / 3){  h = h \* 3 + 1;  }  while (h > 0) {  for (outer = h; outer < arr.length; outer++) {  temp = arr[outer];  inner = outer;  while (inner > h - 1 && arr[inner - h] >= temp) {  arr[inner] = arr[inner - h];  inner -= h;  }  arr[inner] = temp;  }  h = (h - 1) / 3;//间隔缩小,最终为1  }  } |

中等规模(几千个项)的数组排序表现良好,非常大的不是最优选择

### 快速排序:

取数组中最右值,一次为基数把数组分为两段,此时基数为有序的,再递归调用方法,

## 栈

定义:先进后出,只能访问最后一个进入的

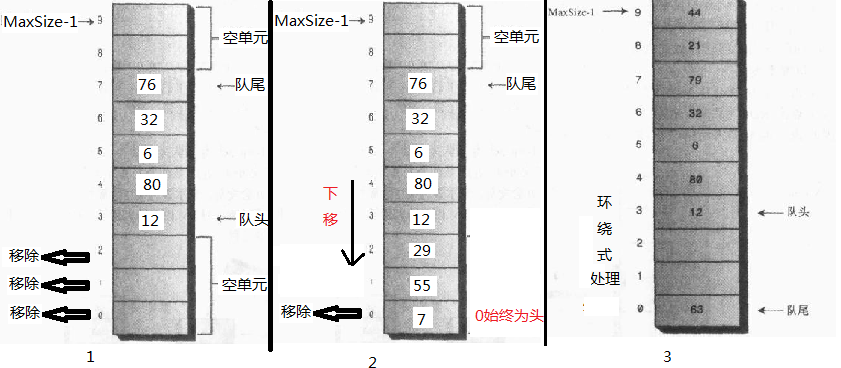
进栈 出栈

|  |
| --- |
|  |
| 4 栈顶 |
| 3 |
| 2 |
| 1 |
| 0 栈底 |

## 队列

头出尾进 场景：队列例如站口取票,打印机队列,键盘输入队列

**用数组实现队列**:



方式1:当插入数据个数为数组长度时也就报废了

方式2:移除一个就要后续全部向下移动,步数过多

方式3:

1. 变量定义: 数组,长度,队头,队尾,数据个数

增加:从队尾下一位开始加,从队头取,第一次下标为0,得出队尾初始值为-1,队头初始值为0

当 个数=长度 时队尾恢复初始值 -1

队尾加1,为队尾加1赋值 🡪 arr[++队尾] = data

个数++

删除:取队头,队头移1 🡪 return arr[头++]

查看: arr[头]

用个数得出 🡪 isEmpty() isFull() size()

-🡪 O(1)

## 链表

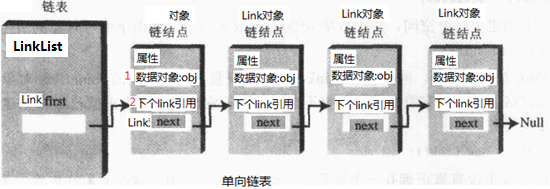
除非频繁通过下标随机访问各个数据,否则在很多使用数组的地方都可以用链表代替

特性: 链表只能从头到尾一个一个找

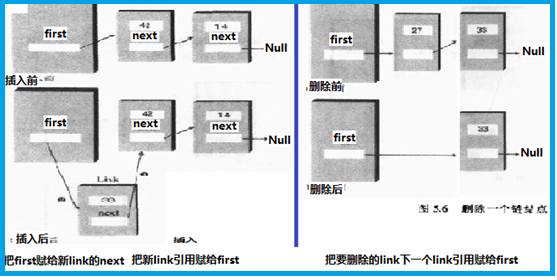
操作: 插入(表头最简单,任意位置) 查找 删除

链表 添加/删除 基本逻辑,用中间变量存储遍历当前节点的上节点变量,引用赋值调换

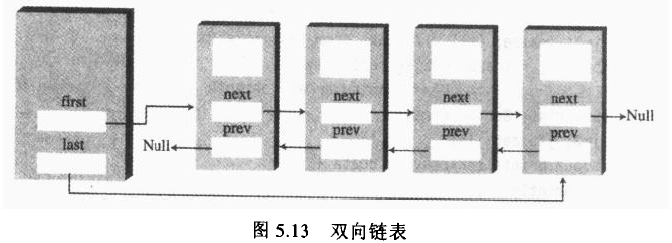
* **单链表**



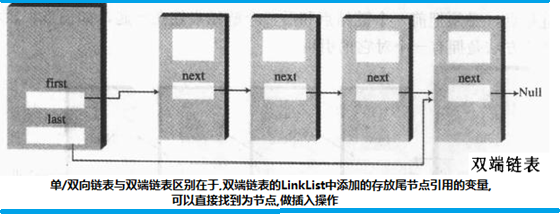
插入: 删除



* **双端双向链表**



* **双端链表**



## 哈希表

结构:数组+链表

缺点:当数组快填满时效率特别低下,扩容(扩大数组容量)copy耗时,数据是通过key哈希值添加到数组下标上的链表中所以没有顺序,不能按数据值进行大小遍历

场景:不需要有序遍历数据,并可提前预测数据量定义表大小即数组大小

禁止重复key,如果放会覆盖

哈希是通过hash单词音译过来的，也可以称为散列表

**出现原因:**查找数组中对象id=100的,步骤为遍历数组,获取对象,比较对象id值,获取下标取出对象,即使有序数组二分查找也需要遍历比较,如果让比较值直接做为数组下标则会省去遍历步骤直接定位(使用数组最大优势)

理想场景:1000个员工,编号id从1—1000,其他属性无关紧要,可以直接把对象放到数组下标为其id值的位置上Person p = arr[id]; 但这种情况是极其特殊的1.属性值唯一且是数字 2.连续紧密(如果其中有id=10000则需要创建长度为10000的数组,其中8999个为洞,浪费内存),

如果要存字典,有50000个单词,可以创建一个50000长度的数组,但是单词”goog”跟下标是没有关系的,如何把关键字遇下标建立关系?

ASCII编码中,a是97,b是98,…z是122

自己定义a>1,…z>26 空格>0 假设单词长度最长为10,

**方式一:** 相加(下标太少)

把数值相加最小a=0+0+……+1=1,最大zzzzzzzzzz = 26+26+….+26 = 260,260长度数组是放不了50000个单词的,如果考虑放子数组或链表,则同一个下标会平均放5000/260=192个数据,存取快但降低查找/删除速度,还有问题就是子数组中也需要再添加子数组

**方式二:** 幂的连乘(能表示唯一但下标太大)

“cats” 3\*273 + 1\*272 +20\*271 +19\*0 = 60337

“zzzzzzzzzz” 26\*279 +… 只是279就超过了700000000000 不可能创建这么大的数组

**方式三:**  哈希(把幂的连乘压缩到数组可以接受的范围)

取余方式 下标 = hash(key)%数组长度 < 数组长度

数组长度最好定义为质数(整数在一个大于1的自然数中，除了1和此整数自身外，没法被其他自然数整除的数)减少哈希值冲突,使哈希值更分散

不同单词哈希化后哈希值相同,即冲突,跟相加情况相同,有两种方式处理

1. 开放地址法:线性探测 二次探测 再哈希法

线性探测: x,x+1,x+2,x+3……

添加:加入哈希值为1,但下标1上已经放值,则向第二位直到为null的,如果到尾还没null则扩容,因为哈希函数为key%length现在length变了,之前下标哈希就不能用了,要遍历旧数组,重新添加,这是避免不了的

查询:定位到下标1,如果不是则一次找,如果遇到null停止查找,表示没找到

删除:因为查找遇null停止,所以删除不为null,用数组数据不会出现的值标识,如正数数组用-1



一段连续的位置叫聚集,放入的数据多后,聚集就会增多,影响效率

x,x+12,x+22,x+32……

**链地址法**:

数组+链表(单链表,二叉树,红黑数等)

重复值:允许重复值,但如果要插在表头则只会找到第一个对象

java容器: HashMap

数组+红黑树

哈希函数:

**装载因子** = 填入的数据个数/数组长度

hashmap 0.75 是大量数据测试出来的

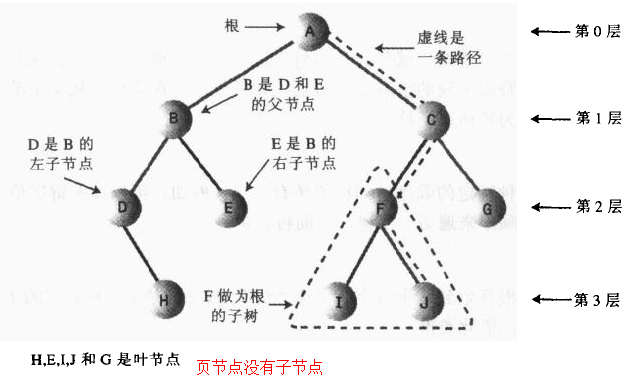
如果放1000个数据到map中初始化应该多大?

## 二叉树又称B树

结合两种数据结构优点,查询跟有序数组一样快,添加删除跟链表一样快

定义:只有一个根节点且每个节点最多只能有两个子节点

左子节点关键字值<父节点<右子节点关键字值(此处按升序)



缺点:非平衡树

## 红黑树又称平衡二叉树

即平衡二叉树 规则:

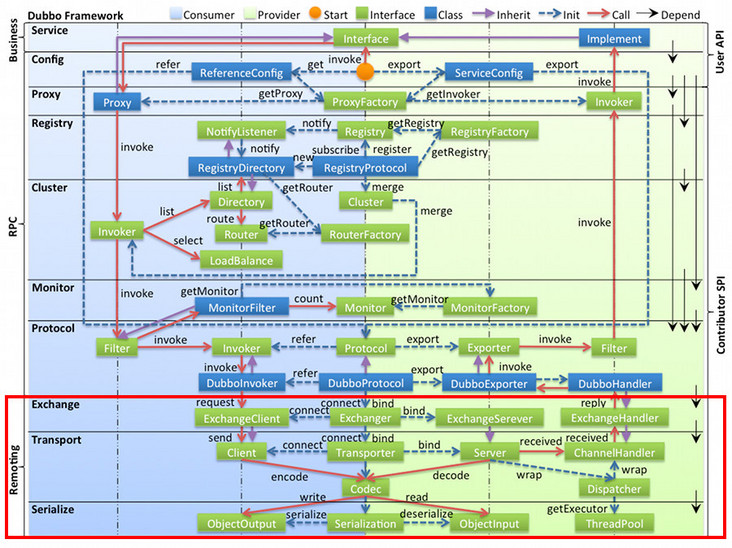
1. 根总是黑色的
2. 每个节点不是红色的就是黑色的
3. 如果节点是红色的,则他的子节点必须是黑色的(反之不一定必须为真)
4. 从根到叶节点或空子节点的每条路径,必须包含相同数目的黑色节点

通过此规则写出来的二叉树就是平衡的

# Dubbo

分为服务调用和服务管理，通过SPI机制实现功能组件扩展，通过URL做为数据总线

## 模块与功能



Dubbo 框架设计一共划分了 10 个层：

### 服务接口层（Service）：

业务接口与实现

### 配置层（Config）：

#### 注册配置类到spring

如：Application / Registry / ProtocolConfig

1. spring加载配置文件内容到Environment [inˈvīəmənt]
2. 用注解 映射 配置文件属性与配置类关系

@EnableConfigurationBeanBinding(prefix="dubbo.application",type=XxxConfig.class)

1. 处理类实现ImportBeanDefinitionRegistrar，创建配置类beanDefinition，从Environment 中获取属性值，注册beanDefinition到spring [defə ˈ niSH(ə)n]

#### @DubboService注解处理，服务暴露

1. 注册@DubboService处理器ServiceAnnotationBeanPostProcessor到spring

-> 扫描包路径下注解类，注册ServiceBean到spring

2. 注册监听事件DubboBootstrapApplicationListener

-> spring初始化完成, 触发监听, 进行服务暴露export()与注册

一.启动DubboBootstrap，配置检查，配置对象集中到ConfigManager

二.遍历ServiceConfig{ 服务启动，注册

配置校验，默认初始化

组装URL

加载监控器

获取代理对象 @SPI ProxyFactory return AbstractProxyInvoker

服务启动@SPI Protocol 只启动一个服务端

aop:ProtocolFilterWrapper.export() :组装invoke()过滤器链@SPI Filter@Activate(group=

DubboProtocol.export(){ 启动服务端

1.初始化请求处理器ExchangeHandlerAdapter **服务端调用代理方法返回响应结果**

2.Exchangers.bind(url, handler) -> **信息交换层外观模式**

@SPI Exchanger -> HeaderExchanger.bind(url, handler) ->

new HeaderExchangeServer(

3.Transporters.bind( url, **传输层外观模式**

new DecodeHandler( 装饰链：解码处理器

new HeaderExchangeHandler(handler)))); 装饰链：协议头处理器

@SPI Transporter ->NettyTransporter.bind(new NettyServer(url, handler));

NettyServer构造方法初始化{

添加过滤器链： @SPI Dispatcher 装饰链：线程池分发处理器 默认All

HeartbeatHandler 装饰链：心跳功能处理器

MultiMessageHandler装饰链：多消息功能处理器

线程池

编解码器Codec2

超时等

doOpen() 启动服务端

}

RegistryProtocol.export(){ 注册，订阅，监听

@SPI RegistryFactory -> ZookeeperRegistryFactory

-> new ZookeeperRegistry 注册，订阅，监听

#### @DubboReference注解处理，服务引用

1.注册@DubboReference处理器ReferenceAnnotationBeanPostProcessor到spring

-> 扫描包路径下注解类，注册ServiceReference到spring

ReferenceConfig.get() {

配置校验，默认初始化

组装URL

服务引用@SPI Protocol

aop:ProtocolFilterWrapper.

RegistryProtocol.refer(){

获取注册中心registry对象

@SPI Cluster集群 放入invoker

}

DubboProtocol.refer(){

获取ExchangeClient ->

Exchangers.connect(url, handler) -> **信息交换层外观模式**

@SPI Exchanger -> HeaderExchanger.connect(url, handler) ->

new HeaderExchangeClient(

Transporters.connect( url, **传输层外观模式**

new DecodeHandler( 装饰链：解码处理器

new HeaderExchangeHandler(handler)))); 装饰链：协议头处理器

@SPI Transporter ->NettyTransporter.connect(new NettyClient(url, handler));

NettyClient构造方法初始化{

添加过滤器链： @SPI Dispatcher 装饰链：线程池分发处理器 默认All

HeartbeatHandler 装饰链：心跳功能处理器

MultiMessageHandler装饰链：多消息功能处理器

线程池

编解码器Codec2

超时等

doOpen() 启动客户端

}

创建DubboInvoker

}

@SPI ProxyFactory 获取代理对象，放入spring

MigrationInvoker

### 服务代理层（Proxy）：

@SPI ProxyFactory 提供jdk和javassist 两种实现

Jdk: 定义：接口，目标类，InvocationHandler实现类

**消费端**

Proxy.newProxyInstance(

接口类加载器,

实现的接口列表,

new InvokerInvocationHandler(目标类实例对象));

调用：

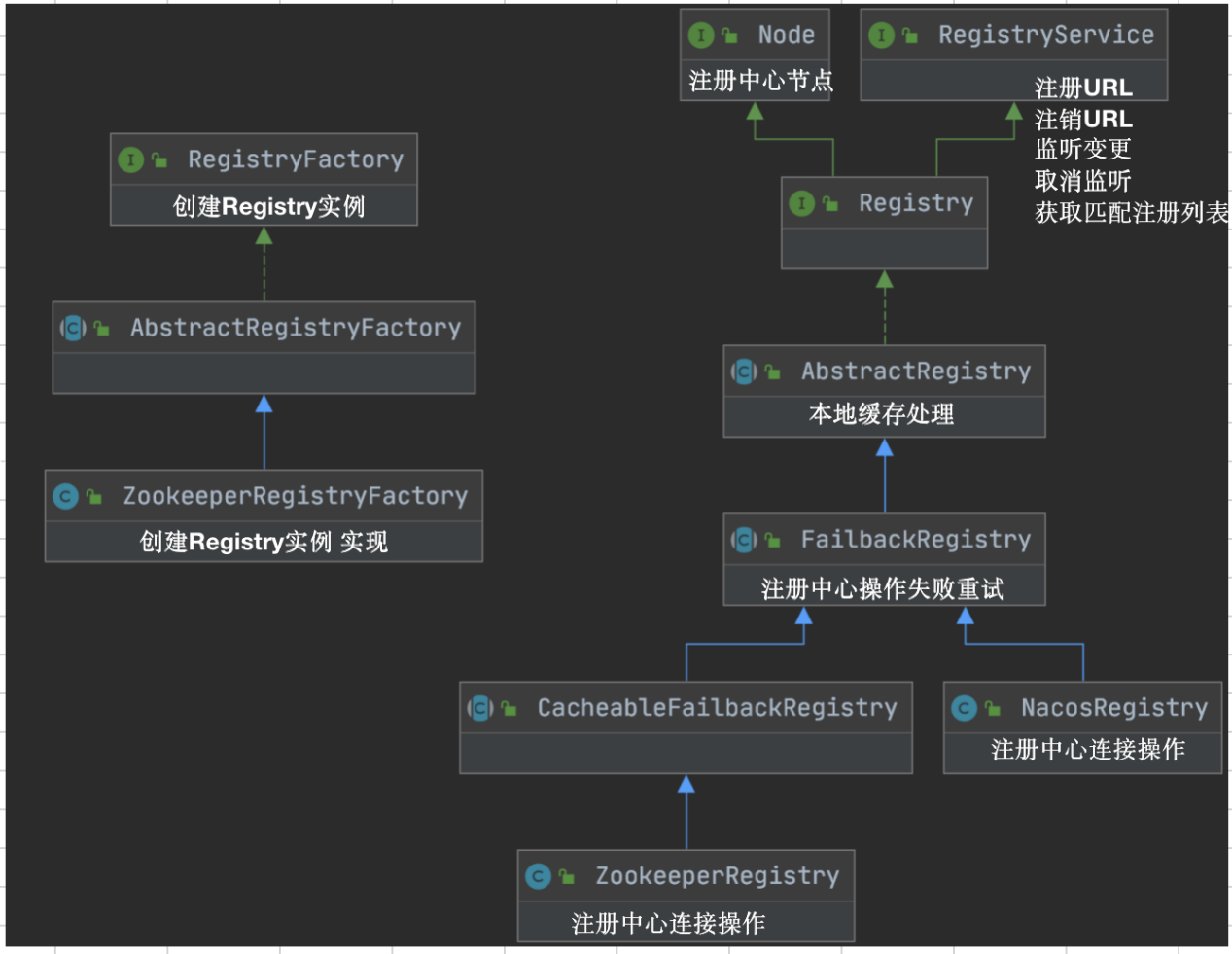
代理对象调用同名方法 -> 方法中调用InvokerInvocationHandler.invoke() ->目标类方法

**生产端**

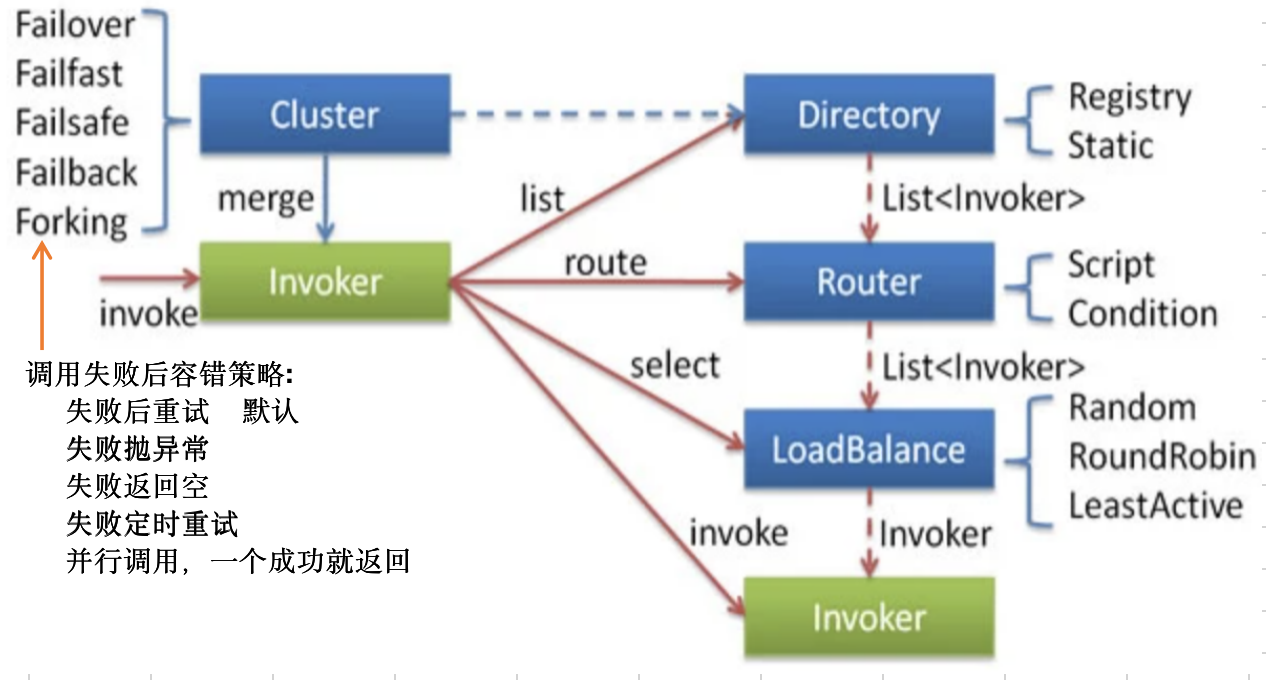
返回new AbstractProxyInvoker 反射调用

### 服务注册层（Registry）：

封装服务地址的注册与发现，以服务 URL为中心。



### 集群层（Cluster）：



1. Directory 资源目录中获取List<invoker>：

方式一：register对象中获取

方式二：静态配置

1. 路由List<invoker>中过滤: 用于应用隔离,读写分离,灰度

3. 负载均衡: List中获取执行invoker

RandomLoadBalance 权重随机

RoundRobinLoadBalance 加权重轮询

LeastActiveLoadBalance 基于最少活跃

ConsistentHashLoadBalance hash一致性

### 监控层（Monitor）：

统计服务调用次数，调用时间的，调用链跟踪的服务。

### 远程调用层（Protocol）：

封将 RPC 协议 @SPI Protocol

提供生产、消费端调用是过滤器 @SPI Filter

数据封装Invocation 和 Result

### 信息交换层（Exchange）：

RpcInvocation -转-> Request

Response -转-> AsyncRpcResult

ExchangeCodec -> Request Response 编解码功能

### 网络传输层（Transport）：

@SPI Dispatcher 线程派发

抽象Client，RemotingServer，Channel 默认netty

#### netty集成实现：

ExchangeCodec 数据编解码



### 序列化层（Serialize）

提供协议消息body序列化反序列化，支持fastjson,jdk,gson等 默认Hessian

## Spi

是一种服务发现机制，调用方定义接口（既一个扩展点），被调用方提供实现，达到组件扩展的目的。

**本质是**指定路径下，接口全限定名（既扩展点）为文件名，实现类全限定名为内容，由加载器根据接口全限定名找到配置文件，解析内容获取实现类全限定名，反射生成实例对象

**Jdk spi** ：

**配置**：在META-INF.services.路径下

如jdbc，jdk定义接口java.sql.Driver，数据库提供接口实现，jdk 驱动管理器类DriverManager的static{ ServiceLoader.load(Driver.class) } 加载实现类

**Dubbo-spi**

1.增加获取默认、指定、适配 扩展点实现功能（jdk加载全部，资源浪费）

2.增加对扩展点aop,ioc（set方式注入）的支持

@SPI

表示一个接口扩展点，对应一个 ExtensionLoader 实例

@Adaptive

扩展点适配器类，如依赖对象工厂节点，有适配器、spi、spring三种实现。加载工厂节点时指定调用适配器实现，获取依赖对象时，会从spi、spring中遍历获取，哪个获取到就返回

**步骤一：创建扩展加载器**

ExtensionLoader{

type = 接口类型

ExtensionFactory加载器加载依赖注入对象工厂节点

AdaptiveExtensionFactory[

SpringExtensionFactory 依赖对象从spring容器中获取

SpiExtensionFactory 生成依赖节点接口动态代理类（jdk，javassist）， 内部通过dubbo URL中获取加载器加载所需参数(接 口名，指定实现名)

]

}

**步骤二：加载扩展实例对象**

1. 单例模式->双层检查锁
2. 指定路径+接口全限定名=文件路径，读文件获取实现类名并加载，放入缓存

适配器@Adaptive类型：

AOP代理类型： 构造器形参为接口类型

普通类型处理

1. 依赖注入ioc

编写：set方式注入

dubbo URL做数据总线，携带具体接口实现类信息

处理：通过依赖对象工厂获取依赖对象，反射赋值

1. 静态代理aop

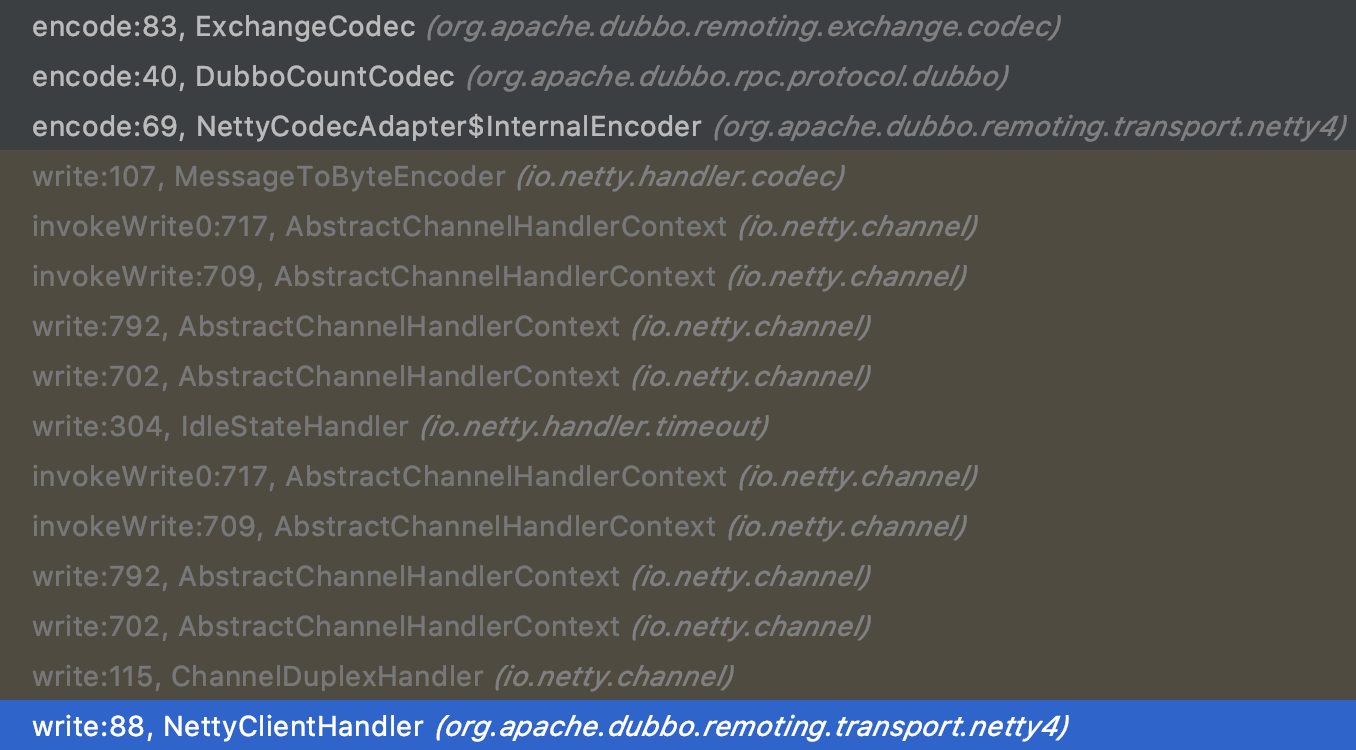
代理类编写： 构造器(接口类型 )

重写方法() {被代理对象方法调用 切面逻辑编写}

处理：扩展加载器缓存获取代理对象，通过反射生成代理对象

## 线程模型

### 消费端请求及响应处理





代理对象 -> InvokerInvocationHandler.invoke()

创建 RpcInvocation ，URL设置进本地上下文

Cluster集群层，ClusterInvoker 容错调用，获取invoke列表，路由，负载均衡

Protocol协议层，DubboInvoker

单向调用

双向调用

获取线程池：

异步：Executor.newCachedThreadPool(0,max,,,SynchronousQueue)线程池

同步：new ThreadlessExecutor( newCachedThreadPool ) 包装线程池

Exchange信息交换层

RpcInvocation -转-> Request 会生成唯一请求ID AtomicLong

创建DefaultFuture

Transport网络传输层

NettyChannel发送，Request编码，body序列化

DefaultFuture等待响应结果

同步：ThreadlessExecutor.waitAndDrain() 业务线程阻塞与阻塞队列，等待响应

ThreadlessExecutor{

LinkedBlockingQueue queue; 用于同步阻塞请求线程，等待响应返回放入任务

线程池

CompletableFuture

waitAndDrain(){

queue.take(); 阻塞获取队列任务

}

execute(Runnable runnable){ 响应返回后，分发线程池调用

异步 线程池.execute(runnable) 放入线程池

同步 queue.add(runnable) 放入队列

}

}

DefaultFuture extends CompletableFuture{

static ConcurrentHashMap<request.id , Channel>

static ConcurrentHashMap<request.id , DefaultFuture>

线程池

请求ID

Channel

Request

超时时间

发送时间

Timeout 检查超时定时任务

}

**接收响应：**

ExchangeCodec.decode 协议解码 -> Response

NettyClientHandler.channelRead() 接收泳道消息

MultiMessageHandler.received() 多消息处理

HeartbeatHandler.received() 处理心跳请求、响应事件

AllChannelHandler.received(){ 线程池分发 @SPI Dispatcher

通过Response中requestID,从DefaultFuture集合中获取DefaultFuture

从DefaultFuture获取线程池，执行.execute(new ChannelEventRunnable())

同步：ThreadlessExecutor.execute( 会放入阻塞队列 )

请求线程获取到任务,并执行，跳出阻塞

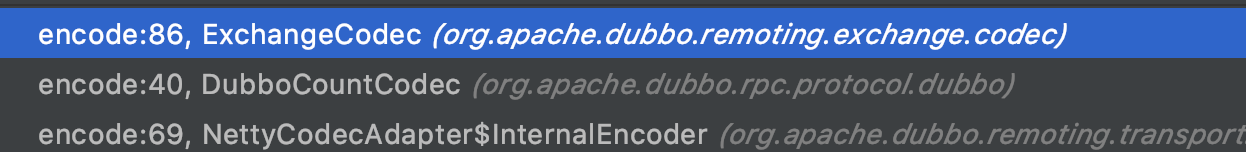
ChannelEventRunnable{

run(){响应，连接，取消连接，发送，异常 处理}

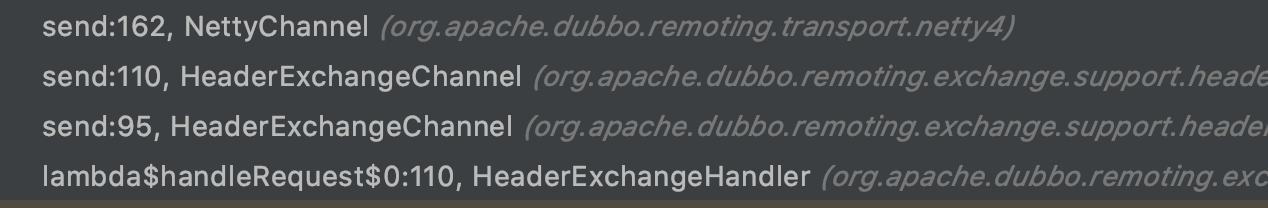
}

### 生产端接收与响应处理：

NeetyServerWorker线程池 创建连接，轮询Channel

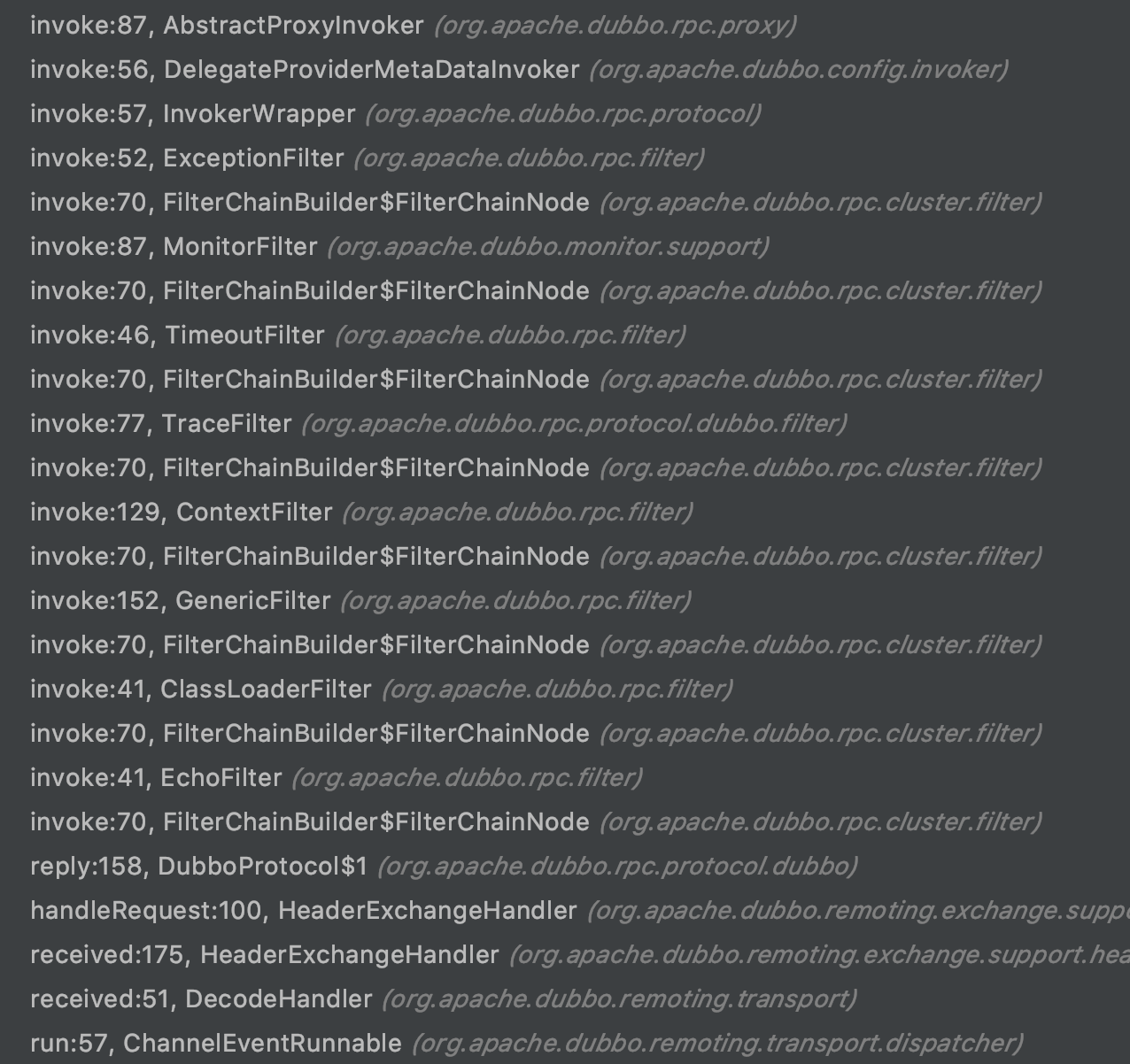


截屏2021-06-04 下午8.22.20



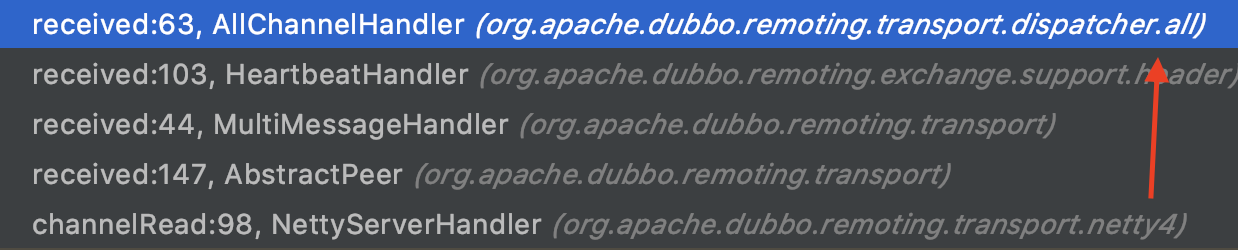


经过过滤器，获取代理对象并执行业务逻辑



截屏2021-06-04 下午8.24.54

线程池派发，执行ChannelEventRunnable



本科：河北科技学院 计算科学与技术

专科：石家庄职工大学 计算科学与技术

基础课：微积分、线代、复变与积分变换、概率论与数理统计、大物、电路原理、模拟电子基础、数字电子基础、英语、大学计算机基础

专业基础课：C语言、汇编、离散、数据结构、信息技术导论、计算思维、信号与系统、操作系统、串并行结构设计、计算机组成原理、数据库系统原理、计算机系统结构