面试偏门知识总结

一、 接口文档生成工具

RAP2

二、 SpringBoot的核心功能

1、 独立运行的Spring项目

Spring Boot可以以jar包的形式进行独立的运行，使用：java -jar xx.jar 就可以成功的运行项目，或者在应用项目的主程序中运行main函数即可；

2、 内嵌的Servlet容器

内嵌容器，使得我们可以执行运行项目的主程序main函数，并让项目的快速运行

3、 提供starter简化Manen配置

Spring Boot提供了一系列的starter pom用来简化我们的Maven依赖

4、 自动配置Spring

Spring Boot会根据我们项目中类路径的jar包/类，为jar包的类进行自动配置Bean，这样一来就大大的简化了我们的配置。当然，这只是Spring考虑到的大多数的使用场景，在一些特殊情况，我们还需要自定义自动配置

5、 应用监控

Spring Boot提供了基于http、ssh、telnet对运行时的项目进行监控

三、 单用户登入

思路：登入前先在cas服务器找到用户的登入票据（TGT）注销掉，由于CAS本身就对TGT和ST做了关联，所以TGT注销后通过它生成的ST会统统自动注销，完成对上一个登陆终端的踢出

参考文档： https://blog.csdn.net/qq_34125349/article/details/80272826

面试参考这个：https://www.cnblogs.com/hihtml5/p/7101475.html

四、 nginx

1、 反向代理配置，通过域名访问服务器

```
  #sina
    server {
        listen       80;
        server_name  www.sina.com.cn;

        #charset koi8-r;

        #access_log  logs/host.access.log  main;

        location / {
            proxy_pass  http://192.168.25.142:8080;
            index  index.html index.htm;
        }
```

2、 负载均衡配置

```
  upstream tomcat-sina-route{
       server 192.168.25.142:8080 weight=1;
       server 192.168.25.142:8083 weight=2;
    }

    #sina
    server {
        listen       80;
        server_name  www.sina.com.cn;

        #charset koi8-r;

        #access_log  logs/host.access.log  main;

        location / {
            proxy_pass   http://tomcat-sina-route;
            index  index.html index.htm;
        }

```

nginx五种负载均衡策略：

```
1、轮询（默认）
每个请求按时间顺序逐一分配到不同的后端服务器，如果后端服务器down掉，能自动剔除。
upstream backserver {
server 192.168.0.14;
server 192.168.0.15;
}

2、指定权重
指定轮询几率，weight和访问比率成正比，用于后端服务器性能不均的情况。
upstream backserver {
server 192.168.0.14 weight=10;
server 192.168.0.15 weight=10;
```

```
}

3、IP绑定 ip_hash
每个请求按访问ip的hash结果分配，这样每个访客固定访问一个后端服务器，可以解决session的问题。
upstream backserver {
ip_hash;
server 192.168.0.14:88;
server 192.168.0.15:80;
}

4、fair（第三方）
按后端服务器的响应时间来分配请求，响应时间短的优先分配。
upstream backserver {
server server1;
server server2;
fair;
}

5、url_hash（第三方）
按访问url的hash结果来分配请求，使每个url定向到同一个后端服务器，后端服务器为缓存时比较有效。
upstream backserver {
server squid1:3128;
server squid2:3128;
hash $request_uri;
hash_method crc32;
}
```

其他--> 最少并发数： 根据每台服务器的连接数记录，将新的连接将传递给连接数最少的服务器

五、 dubbo

注册中心基于接口名查询服务提供者的IP地址；B系统在注册中心将自己的Url注册进去，然后注册中心将Url返还给系统A

在实际调用过程中，Provider的位置对于Consumer来说是透明的，上一次调用服务的位置（IP地址）和下一次调用服务的位置，是不确定的。这个地方就是实现了软负载。

六、 mq最终消息一致性

详情看分布式事务

https://www.jianshu.com/p/eb7a36d25b2a

分区容忍性(Partition Tolerance)：分区容忍性就是允许系统通过网络协同工作，分区容忍性要解决由于网络分区 导致数据的不完整及无法访问等问题。

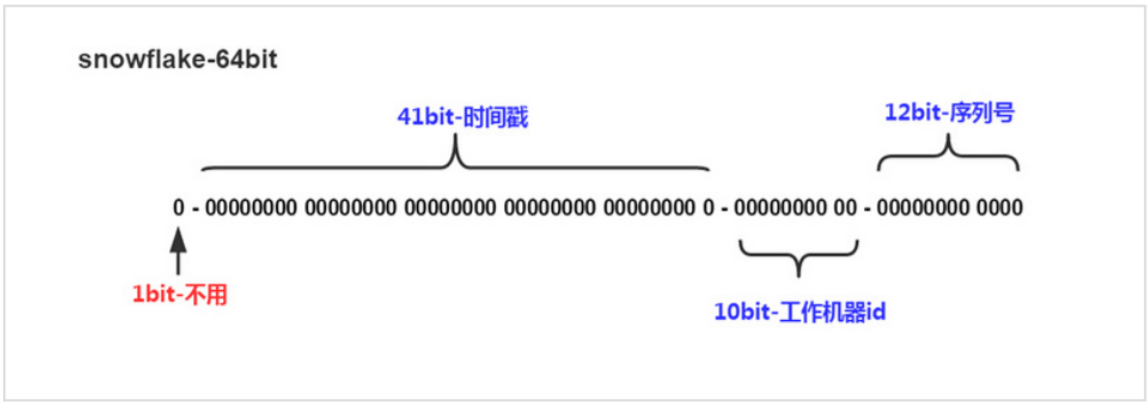
在保证分区容忍性的前提下一致性和可用性无法兼顾，如果要提高系统的可用性就要增加多个结点，如果要保证数据的一致性就要实现每个结点的数据一致，结点越多可用性越好，但是数据一致性越差。 所以，在进行分布式系统设计时，同时满足“一致性”、“可用性”和“分区容忍性”三者是几乎不可能的。

七、 高并发问题

八、 分布式id

1. 雪花算法

SnowFlake算法生成id的结果是一个64bit大小的整数，它的结构如下图：



- 1位，不用。二进制中最高位为1的都是负数，但是我们生成的id一般都使用整数，所以这个最高位固定是0
- 41位，用来记录时间戳（毫秒）。
  - 41位可以表示$2^{41}-1$个数字，
  - 如果只用来表示正整数（计算机中正数包含0），可以表示的数值范围是：0 至 $2^{41}-1$，减1是因为可表示的数值范围是从0开始算的，而不是1。
  - 也就是说41位可以表示$2^{41}-1$个毫秒的值，转化成单位年则是$(2^{41}-1)/(1000*60*60*24*365)=69$年
- 10位，用来记录工作机器id。
  - 可以部署在$2^{10}=1024$个节点，包括5位datacenterId 和 5位workerId
  - 5位（bit）可以表示的最大正整数是$2^5-1=31$，即可以用0、1、2、3、....31这32个数字，来表示不同的datecenterId或workerId
- 12位，序列号，用来记录同毫秒内产生的不同id。
  - 12位（bit）可以表示的最大正整数是$2^{12}-1=4096$，即可以用0、1、2、3、....4095这4096个数字，来表示同一机器同一时间截（毫秒)内产生的4096个ID序号

64位二进制转成十进制有19位数字:

使用了long类型，long类型为8字节工64位。可表示的最大值位2^64-1（18446744073709551615，装换成十进制共20位的长度，这个是无符号的长整型的最大值）。

单常见使用的是long 不是usign long所以最大值为2^63-1（9223372036854775807，装换成十进制共19的长度，这个是long的长整型的最大值）

2. UUID

通用唯一识别码 (Universally Unique Identifier)

UUID由以下几部分的组合： 当前日期和时间 + 时钟序列 + 全局唯一的IEEE机器识别号

UUID的唯一缺陷在于生成的结果串会比较长： 81e853bd-e8e4-4f71-a5ec-8f6ae148b8ff

`去掉-有32位，而雪花算法只有19位`

九、 锁

1、 乐观锁 & 悲观锁

乐观锁：

总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据。

乐观锁适用于多读的应用类型，这样可以提高吞吐量

两种实现方式：

一、版本号机制

一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

二、CAS算法

compare and swap（比较与交换），是一种有名的无锁算法。

CAS机制当中使用了3个基本操作数：内存地址V，旧的预期值A，要修改的新值B。

更新一个变量的时候，只有当变量的预期值A和内存地址V当中的实际值相同时，才会将内存地址V对应的值修改为B。（因为当你改完提交时，可能v被其他线程改了）

cas的缺点:

- **ABA 问题**

  如果内存地址V初次读取的值是A，并且在提交更新时检查到它的值仍然为A，那我们就能说它的值没有被其他线程改变过了吗?

  如果在这段期间它的值曾经被改成了B，后来又被改回为A，那CAS操作就会误认为它从来没有被改变过。这个漏洞称为CAS操作的“ABA”问题。Java并发包为了解决这个问题，提供了一个带有标记的原子引用类“AtomicStampedReference”，它可以通过控制变量值的版本来保证CAS的正确性。因此，在使用CAS前要考虑清楚“ABA”问题是否会影响程序并发的正确性，如果需要解决ABA问题，改用传统的互斥同步可能会比原子类更高效。

- **CPU开销较大** 在并发量比较高的情况下，如果许多线程反复尝试更新某一个变量，却又一直更新不成功，循环往复，会给CPU带来很大的压力。
- **不能保证代码块的原子性** CAS机制所保证的只是一个变量的原子性操作，而不能保证整个代码块的原子性。比如需要保证3个变量共同进行原子性的更新，就不得不使用Synchronized了。

  因为它本身就只是一个锁住总线的原子交换操作。两个CAS操作之间并不能保证没有重入现象。

CAS适用于写比较少的情况下（多读场景，冲突一般较少），synchronized适用于写比较多的情况下（多写场景，冲突一般较多）

悲观锁：

1. 总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。
2. mysql中的行锁，表锁，读锁，写锁等用的就是这种锁机制。
3. Java中 `synchronized` 和 `ReentrantLock` 等独占锁就是悲观锁思想的实现

参考文档： https://blog.csdn.net/qq_34337272/article/details/81072874

cas：http://www.cnblogs.com/Mainz/p/3546347.html

这个cas通俗易懂些：https://blog.csdn.net/qq_35571554/article/details/82892806

十、 接口 & 抽象类

1、 接口

`接口中的变量会被隐式地指定为public static final变量（并且只能是public static final变量，用private修饰会报编译错误）`

1.1 版本直接的区别

jdk8之前：只能声明公开抽象方法（就是我们平常用的）

jdk8：可以定义静态方法，要实现（就是有大括号）

可以定义默认方法，要实现 `interface B{ default void m1(){} }`

jdk9：可以定义私有默认方法，要实现 `interface B{ private void m1(){} }`

1.2 方法引用规则

`在jdk8及9的版本，对象的父类和接口中默认方法相同的情况下的引用规则`

（1）父类优先于接口 （2）子接口优先于父接口 （3）平级接口出现方法覆盖，实现类必须覆盖 （4）出现方法签名冲突，无法解决

参考文档： https://blog.csdn.net/myname07/article/details/81813109

2、 抽象类

2.1 为什么会有抽象方法

`对于一个父类，如果它的某个方法在父类中实现出来没有任何意义，必须根据子类的实际需求来进行不同的实现，那么就可以将这个方法声明为abstract方法，此时这个类也就成为abstract类了`

2.2 抽象类和普通类的主要有三点区别

1. 抽象方法必须为public或者protected（因为如果为private，则不能被子类继承，子类便无法实现该方法），缺省情况下默认为public;
2. 抽象类不能用来创建对象；
3. 如果一个类继承于一个抽象类，则子类必须实现父类的抽象方法。如果子类没有实现父类的抽象方法，则必须将子类也定义为为abstract类。

抽象类与接口之间的大概区别：

抽象类	接口	
抽象类可以有构造器	接口不能有构造器	
抽象类可以有静态方法、默认方法、私有方法	jdk8之前只有抽象方法，之后有静态方法、默认方法	
如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。	如果你往接口中添加方法，那么你必须改变实现该接口的类。	

十一、RESTful

是一个软件的风格，用这种风格可以让我们的路径变得更加简洁更加方便；像以前我们用的不是这种风格的参数是用？带过去的，它就不是这样带过去的，它充分利用了http协议的四种请求方式：

GET：一般是做查询用的， 安全，幂等

POST：一般是做保存提交用的，不安全，不幂等

PUT：一般是做更新用的，不安全，幂等

DELETE：一般是做删除的，不安全，幂等

`安全：数据库的数据不会改变 幂等：多次操作对服务端没有影响`

十二、JSP内置对象

常用的有request、response、session（就说很久没用了）

参考文档：https://www.cnblogs.com/smyhvae/p/4065790.html

十三、redis

参考文档：https://www.cnblogs.com/jasontec/p/9699242.html

redis是一个nosql的key-value数据库，它支持五种数据类型，有两种持久化方式 RDB（默认） 和AOF ，再比较优缺点；然后再讲下redis的几种架构模式，比较它们的优缺点；redis分布式锁的实现；redis做异步队列；缓存穿透，雪崩以及如何避免。。。

五种数据类型：redis是一个key-value数据库，数据类型指的都是value的数据类型

string 字符串，最基本类型，它可以存储任何形式的字符串，其它的四种类型都是string类型的不同形式

hash string类型的field和value的映射表，键值对集合相当于是一个map，特别适合用于存储对象 （常用）

list 字符串列表，按照插入顺序排序，内部使用双向链表实现； key对应一个list

set 是string类型的无序集合，是通过哈希表实现的；key对应一个set

sorted set 是string类型的有序集合，每个元素都会关联一个double类型的分数，通过分数来排序的

持久化方式：RDB（默认） 在指定的时间间隔内将内存中的数据集快照写入磁盘

快照：关于指定数据集合的一个完全可用拷贝，该拷贝包括相应数据在某个时间点的映像。作用，进行快速的数据恢复，将数据恢复某个可用的时间点的状态。

```
# 时间策略
save 900 1       //900秒内如果超过1个key被修改，则发起快照保存
save 300 10      //300秒内如果超过10个key被修改，则发起快照保存
save 60 10000    //60秒内如果超过10000个key被修改，则发起快照保存
save ""          //禁用RDB

可以配置持久化出错，主进程是否停止写入，保护持久化的数据一致性问题
```

AOF（手动开启） 会将每一个收到的写命令保存到aof文件中，当redis重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容

```
# 是否开启aof
appendonly yes

# 同步方式
appendfsync everysec          //有三种同步模式：
                                  always：把每个写命令都立即同步到aof，很慢，但是很安全
                                  everysec：每秒同步一次 （一般采用这种，可以兼顾速度与安全，最多损失1s的数据）
                                  no：redis不处理交给OS来处理，非常快，但是也最不安全

aof 的方式也同时带来了另一个问题。持久化文件会变的越来越大。例如我们调用incr test命令100次，文件中必须保存全部的100条命令，其实有99条都是多余的。因为要恢复数据库的状态其实文件中保存一条set test 100就够了。

为了压缩aof的持久化文件。redis提供了bgrewriteaof命令。收到此命令redis将使用与快照类似的方式将内存中的数据以命令的方式保存到临时文件中，最后替换原来的文件
```

参考文档：https://www.cnblogs.com/xingzc/p/5988080.html

两个的比较： aof文件比rdb更新频率高，优先使用aof还原数据。

aof比rdb更安全也文件也更大

rdb性能比aof好

如果两个都配了优先加载AOF

---

redis架构模式：1、单机

2、 主从复制

3、 哨兵

`是Redis 的高可用性解决方案：由一个或多个Sentinel 实例 组成的Sentinel 系统可以监视任意多个主服务器，以及这些主服务器属下的所有从服务器，并在被监视的主服务器进入下线状态时，自动将下线主服务器属下的某个从服务器升级为新的主服务器`

4、 集群（proxy）

5、 集群（直连型）

`redis 3.0之后版本支持redis-cluster集群，Redis-Cluster采用无中心结构，每个节点保存数据和整个集群状态,每个节点都和其他所有节点连接`

其结构特点：

1、所有的redis节点彼此互联(PING-PONG机制)，内部使用二进制协议优化传输速度和带宽。

2、节点的fail是通过集群中超过半数的节点检测失效时才生效。

3、客户端与redis节点直连，不需要中间proxy层.客户端不需要连接集群所有节点，连接集群中任何一个可用节点即可。

4、redis-cluster把所有的物理节点映射到[0-16383]slot上（不一定是平均分配），cluster 负责维护node<->slot<->value。

5、Redis集群预分好16384个桶，当需要在 Redis 集群中放置一个 key-value 时，根据 CRC16(key) mod 16384的值，决定将一个key放到哪个桶中。

衍生话题：一致性哈希

参考文档：https://www.cnblogs.com/jasontec/p/9699242.html

缓存穿透：指查询一个一定不存在的数据，由于缓存中查不到，会去数据库查询，而数据库查不到的数据是不写入缓存的，这将导致这个不存在的数据每次请求都要到数据库去查询，失去了缓存的意义

解决方案：

1、最常见的是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉

2、对查询结果为空的情况也进行缓存，缓存时间设置短一点

缓存雪崩：当缓存服务器重启或大量缓存集中在某一个时间段失效，这样在失效的时候，会给数据库带来很大压力，导致数据库崩溃

解决方案：

1、通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。从而避免失效时大量的并发请求落到数据库上

2、做二级缓存，A1为原始缓存，A2为拷贝缓存，A1缓存失效时间设置为短期，A2设置为长期，A1失效时，可以访问A2

3、不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀

缓存击穿：是指某一个key过期时，恰好在这个时间点对这个key有大量的并发请求过来，瞬间把数据库压垮

参考文档：https://blog.csdn.net/zeb_perfect/article/details/54135506

---

redis异步队列：

一般使用list结构作为队列，rpush生产消息，lpop消费消息。当lpop没有消息的时候，要适当sleep一会再重试。

使用brpop和blpop指令解决了频繁调用Jedis的rpop和lpop方法造成的资源浪费问题；brpop指令，这个指令只有在有元素时才返回，没有则会阻塞直到超时返回null。

Redis提供了发布/订阅模式的指令：

发布：publist

订阅：subscribe 取消订阅：unsubscribe psubscribe 基于通配符的消息订阅，可以重复订阅

使用redis做消息队列可能会出现消息丢失的情况，因为没有消息接收的确认机制。

参考文档：https://blog.csdn.net/qq_34212276/article/details/78455004

---

redis分布式锁：

```
setnx  //设置 key 对应的值为 string 类型的 value。如果 key 已经存在，返回 0，nx 是 not exist 的意思。
Expire //设置过期时间（单位秒）

1、先拿setnx来争抢锁，抢到之后，再用expire给锁加一个过期时间防止锁忘记了释放造成死锁
缺点：由于setnx() 和 expire()不具备原子性，如果程序在执行完setnx()之后突然崩溃，导致锁没有设置过期时间。那么将会发生死锁。

2、用set()，有五个形参
jedis.set(lockKey, requestId, SET_IF_NOT_EXIST, SET_WITH_EXPIRE_TIME, expireTime)
第一个，用key来当锁
第二个，解铃还须系铃人，客户端用来解锁的
第三个，当key不存在时，我们进行set操作；若key已经存在，则不做任何操作
第四个，给这个key加一个过期的设置
第五个，代表key的过期时间
```

参考文档：https://www.cnblogs.com/linjiqin/p/8003838.html

十四、 分布式锁

参考文档：https://www.cnblogs.com/austinspark-jessylu/p/8043726.html

https://www.cnblogs.com/seesun2012/p/9214653.html

十五、vue

几个钩子

十六、lucene & solr & elasticsearch

Lucene是一个全文检索引擎工具包，在使用时需要关注搜索引擎系统，例如数据获取、解析、分词等方面的东西，而solr和elasticsearch都是基于该工具包做的一些封装。

总结：

- 你在项目中如何开发搜索模块(结合简历)

  我们使用的是Elasticsearch. 从数据库中把相关的数据导入到索引库中. 例如把文章表的数据导入到索引库里面. 把文章的标题, 摘要进行分词,存储,进行索引. 然后在页面中把搜索结果展示出来, 我们使用的是Spring Data ElasticSearch 框架来实现对 ElasticSearch的操作

- 你如何实现数据库与索引库的同步?
  - 方式一(solar): 通过solr连接数据库，实现数据的定时同步.
  - 方式二(Elasticsearch): 使用 Logstash. Logstash 是一个开源的数据收集引擎，它具有近实时数据传输能力。它可以统一过滤来自不同源的数据，并按照开发者的制定的规范输出到目的地。Logstash 通过管道进行运作，管道有两个必需的元素，输入和输出，还有一个可选的元素，过滤器。我们编写配置文件,通过定时表达式, 定时的从mysql查询出数据作为输入, 把索引库作为输出.
  - 方式三: 使用消息队列(solar和Elasticsearch)

- 你如何实现索引库的分词

  IK里面提供了两种算法 最少切分 最细切分.

  如果是一些新的词汇， 为了查询的准确性， 我们会把新的关键词添加到 IKAnalyzer 的扩展词典中。

- Solr 和Elasticsearch区别
  - Solr 利用 Zookeeper 进行分布式管理，而 Elasticsearch 自身带有分布式协调管理功能;
  - Solr 支持更多格式的数据，而 Elasticsearch 仅支持json文件格式;
  - Solr 官方提供的功能更多，而 Elasticsearch 本身更注重于核心功能，高级功能多有第三方插件提供;
  - Solr 在传统的搜索应用中表现好于 Elasticsearch，但在处理实时搜索应用时效率明显低于 Elasticsearch。
  - Solr 是传统搜索应用的有力解决方案，但 Elasticsearch 更适用于新兴的实时搜索应用。

    https://www.cnblogs.com/chowmin/articles/4629220.html

- 谈谈你对分词,索引,存储的理解

  分词(tokenized): 是否分词，就看在搜索时候是要整体匹配还是单词匹配. 单词匹配就需要分词

- 适合进行分词的: 商品名称、商品描述等，这些内容用户要输入关键字搜索，由于搜索的内容格式大、内容多需要分词后将语汇单元建立索引
- 不适合进行分词的:商品id、订单号、身份证号等

  索引(indexed): 是否进行索引, 就看在搜索时候是否需要被搜索到; 需要被搜索到, 就需要进行索引

- 适合进行索引的: 商品名称、商品描述分析后进行索引，订单号、身份证号不用分词但也要索引，这些将来都要作为查询条件。
- 不适合进行索引的: 图片路径、文件路径等，不用作为查询条件的不用索引。

  存储(stored): 是否存储, 就看在搜索的结果页面上是否需要展示. 需要展示,就要存储

- 适合进行存储的：商品名称、订单号，凡是将来要从Document中获取的Field都要存储。

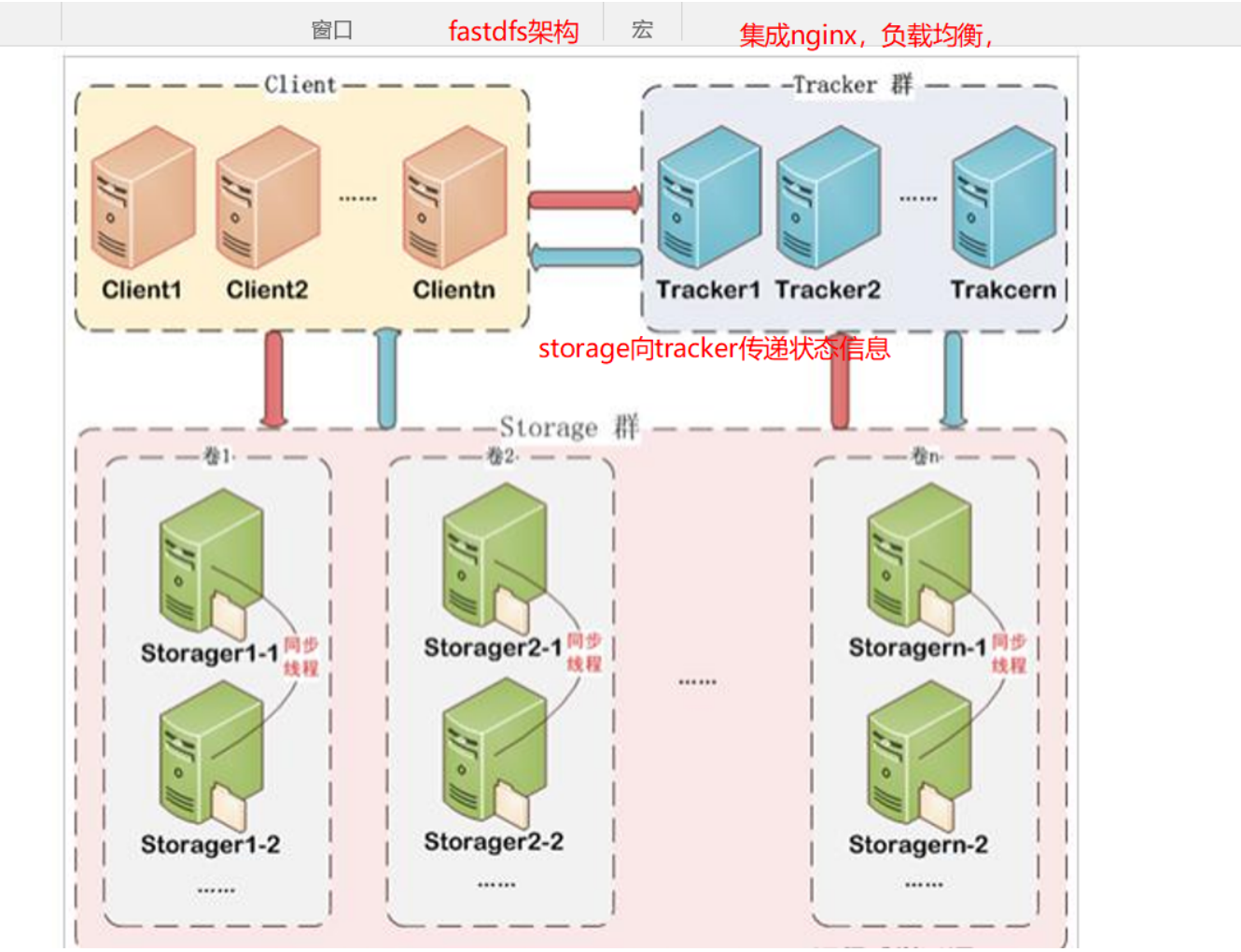
- 不适合进行存储的: 商品描述，内容较大不用存储。如果要向用户展示商品描述可以从系统的关系数据库中获取。

十七、 FastDFS

FastDFS是一个分布式文件系统，充分考虑了高可用、负载均衡（集成nginx）、线性扩容(想怎么加服务器就怎么加)等机制。FastDFS架构包括 Tracker server和Storage server，客户端请求Tracker server进行文件上传、下载，通过Tracker server调度最终由Storage server完成文件上传和下载

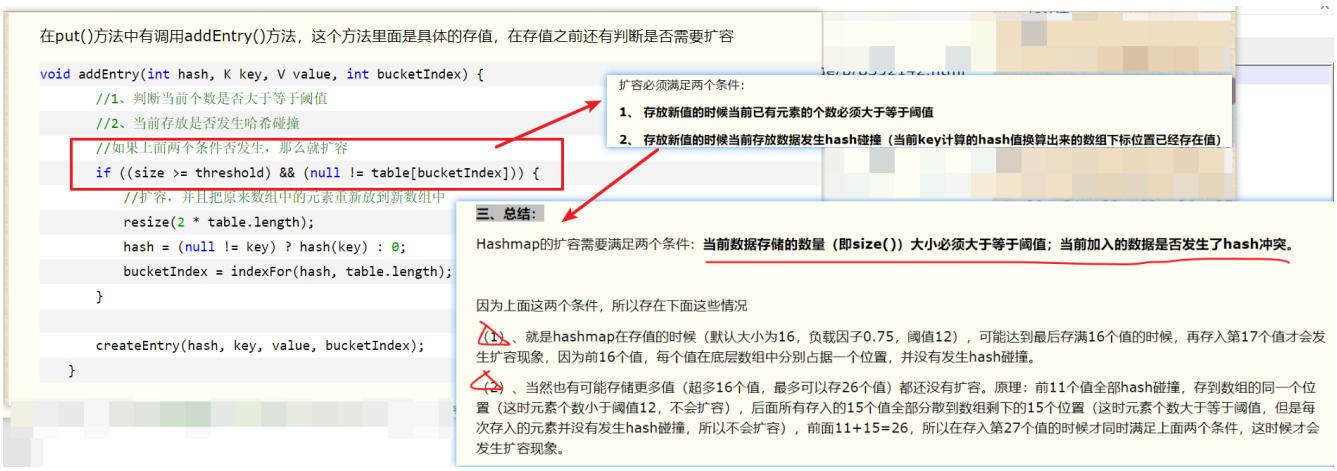
Tracker server之间是相互平等关系同时提供服务，Tracker server不存在单点故障

Storage集群采用了分组存储方式。storage集群由一个或多个组构成，一个组由一台或多台存储服务器组成



十八、 hashmap

1、 jdk7扩容原理详解



参考文档：https://www.cnblogs.com/yanzige/p/8392142.html

在JDK8里面，HashMap的底层数据结构已经变为数组+链表+红黑树的结构了，因为在hash冲突严重的情况下，链表的查询效率是O(n)，所以JDK8做了优化对于单个链表的个数大于8的链表，会直接转为红黑树结构算是以空间换时间，这样以来查询的效率就变为O(logN)

2、 jdk7 & jdk8 hashmap扩容区别

不同点比较：https://blog.csdn.net/u010454030/article/details/82458413

a、扩容条件的不同

jdk7，必须要满足两个条件，`(size >= threshold) && (null != table[bucketIndex])`即数据存储数量大于阀值and发生hash冲突，并且有两种极端情况超过16或26时才扩容。

jdk8，`(++size > threshold)`是先插入数据，插入之后如果超过当前的阈值，就扩容。

oldTab --》扩容前的哈希桶数组

oldCap --》扩容前数组长度 newCap

oldThr --》扩容前的threshold newThr

size --》已存入元素个数

b、迁移扩容策略的不同

jdk7，底层实现都需要新生成一个数组，然后拷贝旧数组里面的每一个Entry链表到新数组里面，这个方法在单线程下执行是没有任何问题的，但是在多线程下面却有很大问题，主要的问题在于基于头插法的数据迁移，会有几率造成链表死循环。

死循环详解：https://blog.csdn.net/zhuqiuhui/article/details/51849692

```
1  void transfer(Entry[] newTable) {
2      Entry[] src = table;                   //src引用了旧的Entry数组
3      int newCapacity = newTable.length;
4      for (int j = 0; j < src.length; j++) { //遍历旧的Entry数组
5          Entry<K, V> e = src[j];             //取得旧Entry数组的每个元素
```

```
        if (e != null) {
            src[j] = null;//释放旧Entry数组的对象引用（for循环后，旧的Entry数组不再引用任何对象）
            do {
                Entry<K, V> next = e.next;
                int i = indexFor(e.hash, newCapacity); //! ! 重新计算每个元素在数组中的位置

                e.next = newTable[i]; //标记[1]
                newTable[i] = e;      //将元素放在数组上
                e = next;             //访问下一个Entry链上的元素
            } while (e != null);
        }
    }
}
```

jdk8，没有采用头插法转移链表数据，而是保留了原始链表顺序，过滤出来扩容后位置不变的元素，放在一起loHead，过滤出来扩容后位置改变到（index+oldCap）的元素，放在一起hiHead 。

详看：https://blog.csdn.net/login_sonata/article/details/76598675

3、 确定哈希桶数组索引位置

```
// 方法一，jdk1.8 & jdk1.7都有:
static final int hash(Object key) {
     int h;
     return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
// 方法二，jdk1.7有，jdk1.8没有这个方法，但是实现原理一样的:
static int indexFor(int h, int length) {
     return h & (length-1);
}
```

这里的Hash算法本质上就是三步： (1) **取key的hashCode值**，h = key.hashCode()； (2) **高位参与运算**，h ^ (h >>> 16)； (3) **取模运算**，h & (length-1)。

通过hashCode()的高16位异或低16位实现的：(h = k.hashCode()) ^ (h >>> 16)

当length总是2的n次方时，h& (length-1)运算等价于对length取模，也就是h%length，但是&比%具有更高的效率

4、 jdk7、jdk8 ConcurrentHashMap区别

JDK7，采用的是分段锁的机制，ConcurrentHashMap维护了一个Segment数组，Segment这个类继承了重入锁ReentrantLock，并且该类里面维护了一个 HashEntry<K,V>[] table数组，在写操作put，remove，扩容的时候，会对Segment加锁，所以仅仅影响这个Segment，不同的Segment还是可以并发的，所以解决了线程的安全问题，同时又采用了分段锁也提升了并发的效率。

JDK8，彻底抛弃了分段锁的机制，主要使用了Unsafe类的CAS自旋赋值+synchronized同步+LockSupport阻塞等手段实现的高效并发，最大的区别在于JDK8的锁粒度更细，理想情况下talbe数组元素的大小就是其支持并发的最大个数。将锁的级别控制在了更细粒度的table元素级别，也就是说只需要锁住这个链表的head节点，并不会影响其他的table元素的读写，好处在于并发的粒度更细，影响更小，从而并发效率更好

而在JDK7里面最大并发个数就是Segment的个数，默认值是16，可以通过构造函数改变，一经创建不可更改，这个值就是并发的粒度，每一个segment下面管理一个table数组，加锁的时候其实锁住的是整个segment，这样设计的好处在于数组的扩容是不会影响其他的segment的，简化了并发设计，不足之处在于并发的粒度稍粗。

ConcurrentHashMap :

线程安全，锁分段， 效率比HashTable高（因为HashTable锁太粗， ConcurrentHashMap锁细些） Segment：是可重入锁ReentrantLock，每个Segment守护者一个HashEntry数组 HashEntry：相当于Node ，但它的next元素被final修饰，所以添加删除只能从头部开始

5、 fail-fast

//快速失败，由于HashMap非线程安全，在对HashMap进行迭代时，如果期间其他线程的参与导致HashMap的结构发生变化了（比如put，remove等操作），需要抛出异常ConcurrentModificationException

modCount 用于记录集合操作过程中作的修改次数

在迭代过程中，访问实际元素前，会比较modCount 与expectedModCount的值，当modCount != expectedModCount时就会抛出ConcurrentModificationException异常。

expectedModCount在整个迭代过程除了一开始赋予初始值modCount外，并没有再发生改变，所以可能发生改变的就只有modCount。

> *hashCode*是jdk根据对象的地址或者字符串或者数字算出来的int类型的数值

十九、 二叉树

1、 常见的树类型

> BST：二叉查找（排序）树 AVL：平衡二叉查找树 RBT：红黑树
>
> 红黑树的定义：一种具有自平衡特性的平衡二叉查找树。

2、 二叉查找树(BST)

> a. 左子树，比中间节点要小，右子树，比中间节点要大 b. 左右子树，也符合上面的特性。

缺陷：

插入单调增加的节点，会退化成链表

3、 平衡二叉查找（排序）树(AVL)

> 左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树

需要通过节点的旋转，保持平衡的特性。上面定义是一个规范，没有提供实现。

4、 红黑树(RBT)

> （1）每个节点或者是黑色，或者是红色。 （2）根节点是黑色。 （3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！] （4）如果一个节点是红色的，则它的子节点必须是黑色的。 （5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

注意： 特性(5)，确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。

具有自平衡特性，可以保证创建的树，满足平衡二叉查找树的规范，是平衡二叉查找树的一个实现。

5、 树的遍历

> 这个"序"针对的是根结点的访问时机
>
> 注意这里是一个递归的过程

前序遍历

中左右

中序遍历

左中右

后序遍历

左右中

二十、 RabbitMQ

- 项目里面有没有消息队列(结合简历说具体业务,大概说一下当前业务里面用到的技术,大概实现流程)
  搜索的索引库和数据库同步
  短信验证码
  商城项目里面商品上下架
  ...
- RabbitMQ 有哪几种发送模式
  四种(生产者消息是发给交换机的, 队列需要和交换机进行绑定, 消费者从队列取出消息)
  直接模式
  分列模式
  主题模式
  头模式

- ActiveMQ和RabbitMQ 区别

	JMS	**AMQP**
定义	Java api	高级消息队列协议
跨语言	否	是
跨平台	否	是
Model	提供两种消息模型： （1）、Peer-2-Peer （2）、Pub/sub	提供了四种消息模型： （1）、direct exchange （2）、fanout exchange （3）、topic change （4）、headers exchange ，后四种和JMS的pub/sub模型没有太大差别，仅是在路由机制上做了更详细的划分；
支持消息类型	多种消息类型： TextMessage MapMessage BytesMessage StreamMessage ObjectMessage Message （只有消息头和属性）	byte[] 当实际应用时，有复杂的消息，可以将消息序列化后发送。 java代码里面可以直接写java对象, 把java对象进行序列化
综合评价	JMS 定义了JAVA API层面的标准；在java体系中，多个client均可以通过JMS进行交互，不需要应用修改代码，但是其对跨平台的支持较差；	AMQP定义了wire-level层的协议标准；天然具有跨平台、跨语言特性。
产品	ActiveMQ	RabbitMQ

注: AMQP ： Advanced Message Queue，高级消息队列协议。它是应用层协议的一个开放标准，为面向消息的中间件设计，基于此协议的客户端与消息中间件可传递消息，并不受产品、开发语言等条件的限制

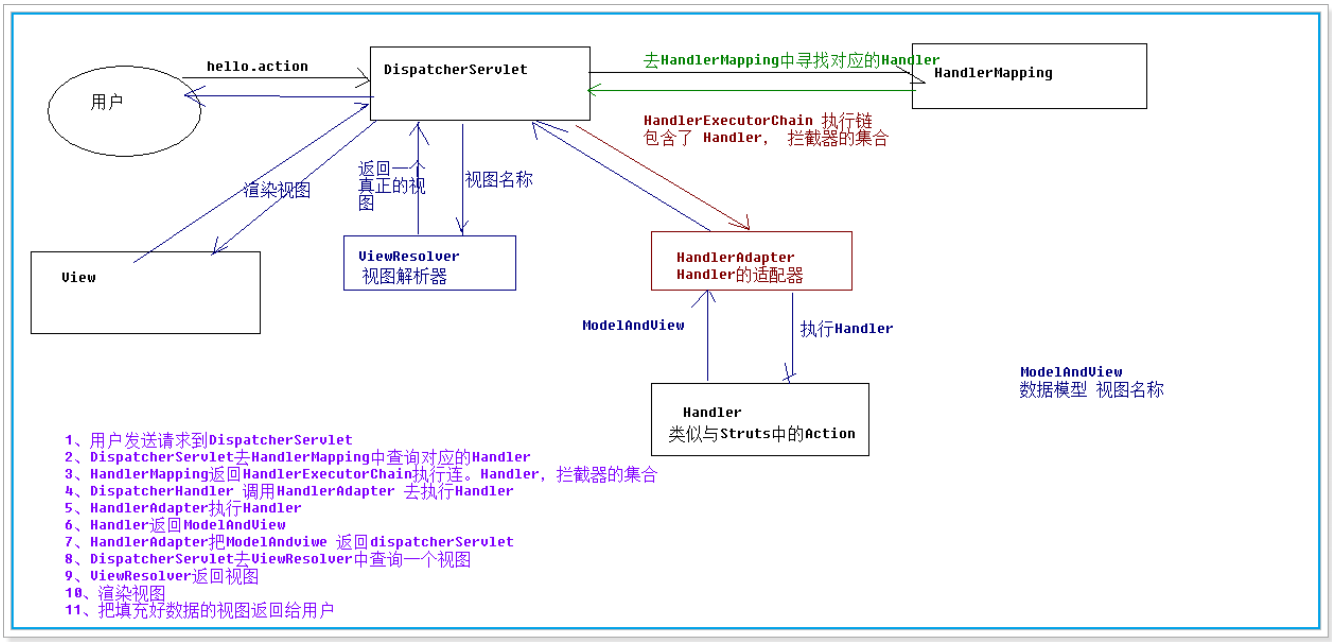
- 用户注册发送短信激活码是怎么实现的

  阿里大于(介绍阿里大于, 介绍一下接入步骤)

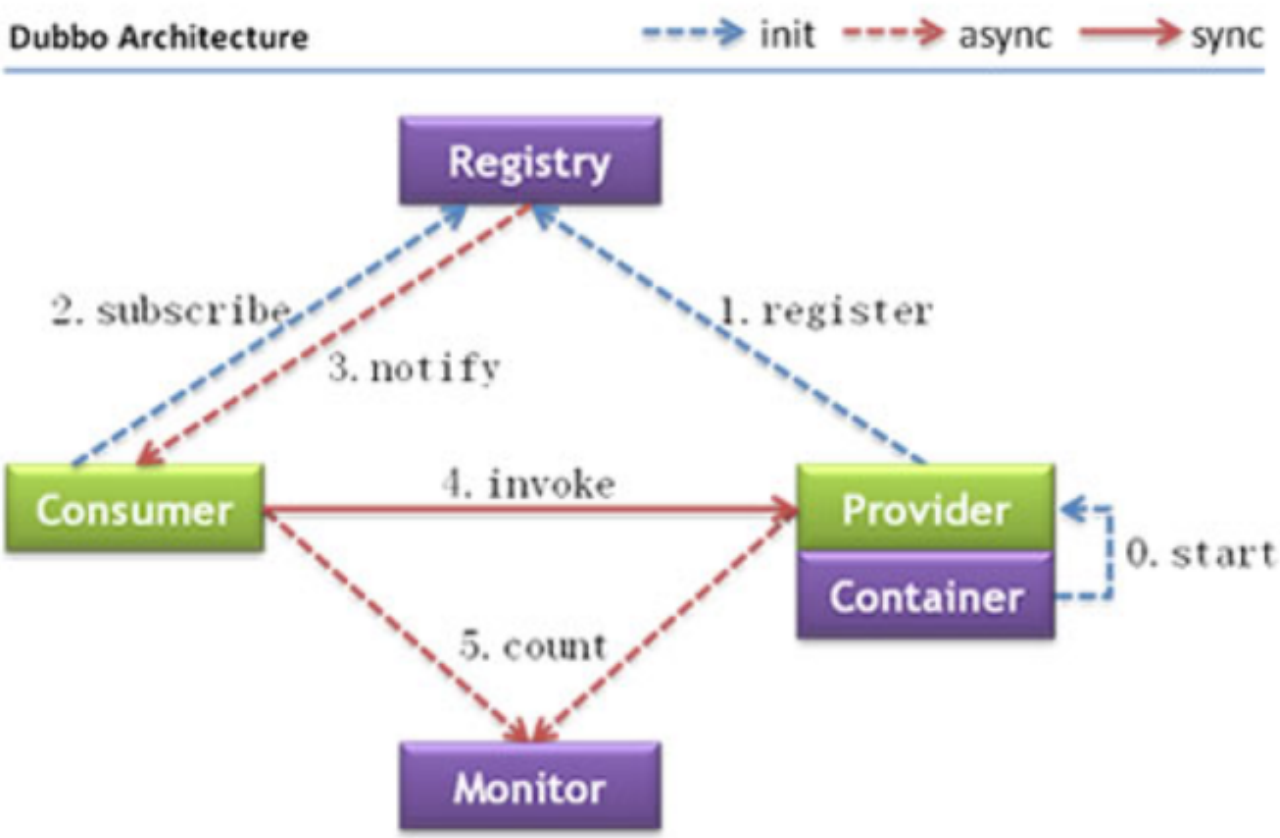
  消息队列

  SpringBoot搭建的一个短信的微服务

二十一、 springmvc


hello.action
DispatcherServlet
去HandlerMapping中寻找对应的Handler
HandlerMapping
用户
HandlerExecutorChain 执行链
包含了 Handler, 拦截器的集合
渲染视图
返回一个真正的视图
视图名称
View
ViewResolver
视图解析器
HandlerAdapter
Handler的适配器
ModelAndView
执行Handler
ModelAndView
数据模型 视图名称
Handler
类似与Struts中的Action
1、用户发送请求到DispatcherServlet
2、DispatcherServlet去HandlerMapping中查询对应的Handler
3、HandlerMapping返回HandlerExecutorChain执行连。Handler, 拦截器的集合
4、DispatcherHandler 调用HandlerAdapter 去执行Handler
5、HandlerAdapter执行Handler
6、Handler返回ModelAndView
7、HandlerAdapter把ModelAndview 返回dispatcherServlet
8、DispatcherServlet去ViewResolver中查询一个视图
9、ViewResolver返回视图
10、渲染视图
11、把填充好数据的视图返回给用户

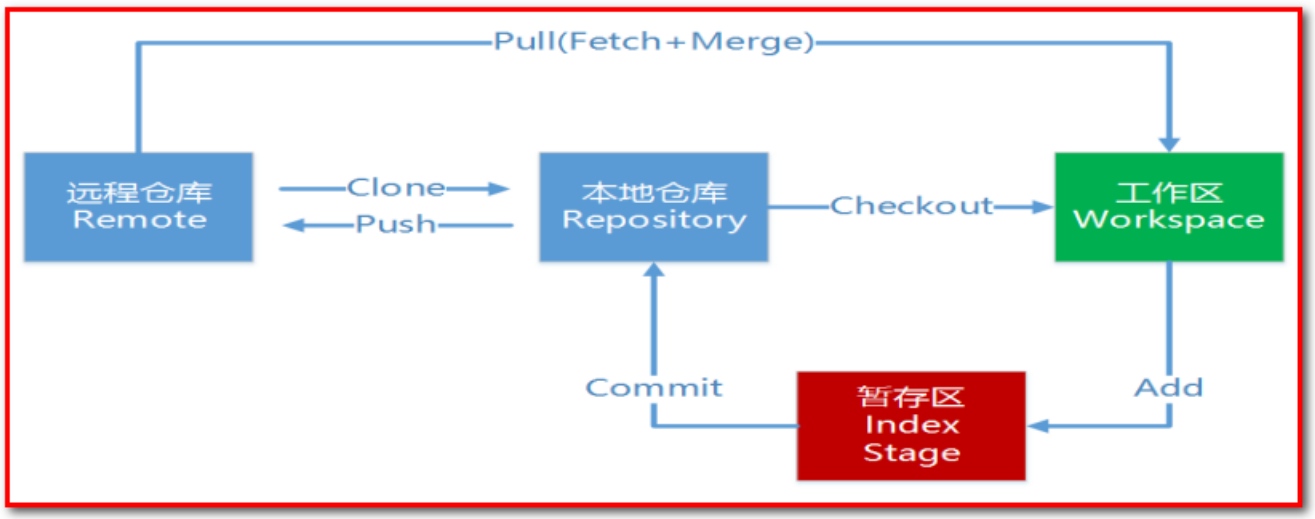
二十二、 dubbo


Dubbo Architecture
init
async
sync
Registry
2. subscribe
1. register
3. notify
Consumer
4. invoke
Provider
Container
0. start
5. count
Monitor

调用关系说明

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

二十三、 git



二十四、 冒泡排序

第一版

> 使用双循环来进行排序。外部循环控制所有的回合，内部循环代表每一轮的冒泡处理，先进行元素比较，再进行元素交换
>
> 缺点：最后面两轮多余，其实已经排好（不能判断出数列已经有序）

```
1  public class One {
2      public static void main(String[] args) {
3  //         int[] array = new int[]{5, 8, 6, 3, 9, 2, 1, 7};
4          int[] array = {5, 8, 6, 3, 9, 2, 1, 7};
5          sort(array);
6          System.out.println(Arrays.toString(array));
7
8      }
```

```
    public static void sort(int array[]) {
        int tmp = 0;
        for (int i = 0; i < array.length; i++) {

            for (int j = 0; j < array.length - i - 1; j++) {
                if (array[j] > array[j + 1]) {
                    tmp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = tmp;
                }
            }
        }
    }
}
```

第二版

> 改进第一版的缺点：判断出数列已经有序，并且做出标记，剩下的几轮排序就可以不必执行

```
public class Two {

    public static void main(String[] args) {
        int[] array = {5, 8, 6, 3, 9, 2, 1, 7};
        sort(array);
        System.out.println(Arrays.toString(array));
    }

    public static void sort(int array[]) {
        int tmp = 0;
        for (int i = 0; i < array.length; i++) {
            //给个标记
            boolean isSorted = true;
            for (int j = 0; j < array.length - i - 1; j++) {
                if (array[j] > array[j + 1]) {
                    tmp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = tmp;
                    //有元素交换就改为false
                    isSorted = false;
                }
            }
            //判断，只要isSorted = true 说明这一轮没有交换过元素，就表示已经是有序的了
            if (isSorted) {
                break;
            }
            System.out.println("第" + i + "轮");
        }
    }
}
```

第三版

> 之前的思路，每一轮遍历完确定有序区为1（即只能确定一个元素不需要比较了），然而真实情况数列可能有几个序列是有序的了，这样每次都会白白比较很多次。
>
> 如何避免：我们可以在每一轮排序的最后，记录下最后一次元素交换的位置，那个位置也就是无序数列的边界，再往后就是有序区了。

```java
public class Three {
    public static void main(String[] args) {
        int[] array = {5, 8, 6, 3, 9, 2, 1, 7};
        sort(array);
        System.out.println(Arrays.toString(array));
    }

    public static void sort(int array[]) {
        int tmp = 0;

        //记录最后一次交换的位置
        int lastExchangeIndex = 0;

        //无序数列的边界，每次比较只需要比到这里为止
        int sortBorder = array.length - 1;

        for (int i = 0; i < array.length; i++) {
            //有序标记，每一轮的初始是true
            boolean isSorted = true;

            for (int j = 0; j < sortBorder; j++) {
                if (array[j] > array[j + 1]) {
                    tmp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = tmp;
                    //有元素交换，所以不是有序，标记变为false
                    isSorted = false;
                    //把无序数列的边界更新为最后一次交换元素的位置
                    lastExchangeIndex = j;
                }
            }
            sortBorder = lastExchangeIndex;
            if (isSorted){
                break;
            }
        }
    }
}
```

详看：https://mp.weixin.qq.com/s/wO11PDZSM5pQ0DfbQjKRQA

鸡尾酒排序（升级）

快速排序（更牛逼）

二十五、 业务相关

1、 三种减库存方式

> 使用预扣最佳，详情看秒杀模块

拍下减库存（拍减）

拍减方式是指在买家提交订单的时候库存就减掉了，拍减需要防恶拍。

付款减库存（付减）

付减方式是指提交订单时不减库存只生成订单，当买家付款完毕后才减库存；付减需要防超卖，超卖是指买家付完款后减库存时库存已经没有了，导致买家付了款但买不到货。因为承若付了款就有货，没货就赔钱，所以必须解决付减模式的超卖问题。

预扣

预扣是指在买家提交订单时，先预扣库存，保留一定时间让买家付款。如果买家在保留时间内付款，那货品肯定是有的（类似拍减），如果买家在保留时间内未付款，那预扣库存会被释放出来重新供大家购买。

2、 如何检测用户是否禁用cookie

js的navigator.cookieEnabled会返回一个布尔值。如果浏览器支持Cookie，就会返回true，否则返回false。

```
var res = "浏览器是否支持Cookie?" + navigator.cookieEnabled;
console.log(res);
```

参考：http://www.sohu.com/a/154839233_99897596

二十六、 三态面试

1、 dubbo链路追踪

https://blog.csdn.net/coolsky600/article/details/63684046

2、 dubbo容错机制、分布式事务

容错机制：https://blog.csdn.net/licwzy/article/details/82054298

分布式事务：https://blog.csdn.net/s13554341560b/article/details/79180327

3、 数据库多版本机制（MVCC）

分为 **快照读**：读取的是记录的可见版本（可能是历史版本，即最新的数据可能正在被当前执行的事务并发修改），不会对返回的记录加锁；

当前读：读取的是记录的最新版本，并且会对返回的记录加锁，保证其他事务不会并发修改这条记录。

> MVCC只在提交读和重复读两个隔离级别下工作

https://blog.csdn.net/winterfeng123/article/details/79048524

4、 使用mybatis的动态sql完成分页查询

```
<mapper namespace="com.yw.test12.StudentMapper">

    <select id="findStudent" resultType="Student" parameterType="Condition" >
        SELECT * FROM student
        <if test="name !=null || name !='' " >
        where    name like #{name}
        </if>
        limit #{start},#{count}

    </select>

```

https://blog.csdn.net/japanstudylang/article/details/51692943

https://www.cnblogs.com/jiyukai/p/9445902.html

二十七、 集群下的session共享

https://uule.iteye.com/blog/2236466

二十八、 zookeeper

zookeeper角色

角色		描述
领导者（Leader）		领导者负责进行投票的发起和决议，更新系统状态
学习者（Learner）	跟随者（Follower）	Follower 用于接收客户请求并向客户端返回结果，在选主过程中参与投票
	观察者（ObServer）	ObServer 可以接收客户端连接，将写请求转发给 leader 节点。但 ObServer 不参加投票过程，只同步 leader 的状态。ObServer 的目的是为了扩展系统，提高读取速度
客户端（Client）		请求发起方

zookeeper的原子广播

Zookeeper的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和Server具有相同的系统状态。

https://www.cnblogs.com/felixzh/p/5869212.html

02.26 -- 面试汇总

1、 springboot如何用tomcat部署

2、 sql执行计划

3、 git分支合并

02.27 -- 面试汇总

1、 jpa 实体类映射的注解

2、 spring用到的注解，说下ioc，aop

3、 mybatis动态sql标签

4、 怎么和前端交流的

就说通过接口文档，举个例子，我们微服务这个项目就是先确定接口文档，在前后端分离开发的

5、 说下cas

讲下流程就行了

6、 springboot你们是根据什么来划分的，依据是什么

根据业务来划分的，每个微服务都有自己单独的业务。是架构师根据表的拆分来确定微服务粒度的，然后分发任务的