J a v a

姓名：季旋
日期：2019.02.22
座右铭：不忘初心，方得始终

如何解决多线程之间线程安全问题?
什么是多线程之间同步?
同步代码块与同步函数区别?
同步函数与静态同步函数区别?
Condition 用法
线程三大特性
线程池四种创建方式
Java 程序初始化顺序:
说说 JDK1.5 并发包
锁的种类
自旋锁
互斥锁
可重入锁
信号量
synchronized（S） VS lock（L）
非聚集索引与聚集索引
https
怎么判断一个 mysql 中 select 语句是否使用了索引
数据库要复习的地方
URL 中可以存在中文吗?
现在需要测试系统的高并发性能，如何模拟高并发?
jdk1.8bin 目录下的东西
若发现 sql 语句执行很慢
什么是 Socket?
接口和抽象类的区别?
Integer 和 int 的区别
反射:
创建对象的方式:
CGLIB 与 JDK 动态代理区别
什么是数据交换格式?
JSON 解析框架有哪些?
XML 解析方式?
Dom4j 与 Sax 区别
反射机制获取类有三种方法
反射创建对象的方式
反射创建 api
使用反射为类私有属性赋值
什么是虚拟机参数配置
堆的参数配置
设置新生代与老年代优化参数
设置新生代比例参数
内存溢出解决办法
设置堆内存大小
设置栈内存大小
Tomcat 内存溢出在 catalina.sh 修改
JVM 堆内存大小
JVM 参数调优总结
MySQL 如何优化
数据库三大范式
慢查询
查询所用使用率
SQL 调优
Myisam 注意事项
数据库数据备份
分表分库
Sevlet 的生命周期（重点）
怎么证明 Servlet 是单例的?
Servlet 的多线程并发问题
转发与重定向区别?
JavaWeb 有哪些会话技术
解决 java 集群的 session 共享的解决方案:
Cookie 的实现原理
Cookie 的应用场景
Session 的实现原理
什么是 token
什么是 UUID
什么是 Filter
Http 格式的分类
https(2443)与 http(80)区别
https 请求方式
GET vs POST 区别
客户端模拟 http 请求工具
服务器模拟 http 请求工具
前端 ajax 请求
Spring 概述
什么是 SpringIOC
什么是 SpringAOP
SpringAOP 创建方式
Spring 是单例还是多例?
IOC 容器创建对象
依赖注入有哪些方式
Spring 事物控制
Spring 事物传播行为
Mybatis 与 Hibernate 区别

Mybatis
安全与防御部分
表单重复提交解决方案(防止 Http 重复提交。)
如何防御 XSS 攻击
跨域实战解决方案
什么是 SQL 语句注入
什么是 NOSQL?
什么是 Redis?
Redis 应用场景
Redis
nginx
Nginx 与 Ribbon 的区别
防盗链机制
集群情况下 Session 共享解决方案
高并发解决方案
消息中间件
activeMQ 底层实现原理:
MQ 产品的分类
Spring Boot 是什么?
Hystrix 隔离策略?
什么是 RPC 远程调用?
什么是 SOA? 与 SOAP 区别是什么?
什么是微服务架构
微服务与 SOA 区别
RPC 远程调用有哪些框架?
什么是 SpringCloud
网关?
ZooKeeper
动态网站与静态网站区别
什么是滑动窗口计数器?
令牌桶算法的原理?
Web 前端有哪些优化方案
什么是 CDN
Dubbo

如何解决多线程之间线程安全问题?

答:使用多线程之间同步或使用锁(lock)。

什么是多线程之间同步？

答:当多个线程共享同一个资源,不会受到其他线程的干扰。

同步代码块与同步函数区别？

同步代码使用自定锁(明锁)，同步函数使用 this 锁

同步函数与静态同步函数区别?

注意:有些面试会这样问：例如现在一个静态方法和一个非静态静态怎么实现同步？

同步函数使用 this 锁，静态同步函数使用字节码文件，也就是类.class

Condition 用法

Condition 的功能类似于在传统的线程技术中的,Object.wait()和 Object.notify()的功能,

代码:

```
Condition condition = lock.newCondition();
res. condition.await();   类似 wait
res. Condition. Signal()  类似 notify
Signalall notifyALL
```

线程三大特性

多线程有三大特性，原子性、可见性、有序性
原子性:保证数据一致性，线程安全。
可见性:对另一个线程是否可见
有序性:线程之间执行有顺序

线程池四种创建方式

Java 通过 Executors（jdk1.5 并发包）提供四种线程池，分别为：
newCachedThreadPool 创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。
newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

排序的代码

```
// 冒泡排序 从小到大排
public static void maopao(int[] arr) {
    int len = arr.length;
    ;
    for (int i = 0; i < len - 1; i++)
        for (int j = 0; j < len - i - 1; j++) {
            if (arr[j] > arr[j + 1])
                swap(arr, j, j + 1);
        }
}

// 选择排序 从小到大排
public static void xuanze(int[] arr) {
    int len = arr.length;
    for (int i = 0; i < len - 1; i++)
        for (int j = i + 1; j < len; j++) {
            if (arr[j] < arr[i])
                swap(arr, j, i);
        }
}
```

```
// 快速排序 从小到大排
public static void kuaisu(int[] arr) {
    int len = arr.length;
    int low = 0;
    int high = len - 1;
    qsort(arr, low, high);
}

private static void qsort(int[] arr, int low, int high) {
    int pos;
    if(low<high){
        pos=partition(arr,low,high);
        qsort(arr,low,pos-1);
        qsort(arr,pos+1,high);
    }

}

private static int partition(int[] arr, int low, int high) {
    // 选择第一个元素作为基准
    int key=arr[low];
    while(low<high){
        while(arr[high]>=key&&high>low)
            high--;
        swap(arr,low,high);

        while(arr[low]<=key&&high>low)
            low++;
        swap(arr,low,high);

    }
    return high;
}
```

```java
// 直接插入排序 从小到大排
public static void zhijiecharu(int[] arr) {
    int len = arr.length;
    for (int i = 1; i < len; i++)
        for (int j = i; j > 0 && arr[j] < arr[j - 1]; j--) {
            swap(arr, j, j - 1);
        }
}

// 归并排序 从小到大排
    public static void guibing(int[] arr) {
        int len = arr.length;
        mergesort(arr,0,len-1);
    }



private static void mergesort(int[] arr, int start, int end) {
    // TODO Auto-generated method stub
    if(start<end){
        int mid=(start+end)/2;
        mergesort(arr,start,mid);
        mergesort(arr,mid+1,end);
        mergepaihaoxu(arr,start,mid,mid+1,end);
    }

}
private static void mergepaihaoxu(int[] arr, int start1, int end1, int start2, int end2)
    // TODO Auto-generated method stub
    int i=start1;
    int j=start2;
    int[] t=new int[end2-start1+1];
    int k=0;
    while(i<=end1&&j<=end2){
        if(arr[i]<arr[j]){
            t[k]=arr[i];
            i++;
            k++;
        }
        else{
            t[k]=arr[j];
            j++;
            k++;
        }
    }

    while(i<=end1){
        t[k++]=arr[i++];
    }
    while(j<=end2){
        t[k++]=arr[j++];
    }
    for(int m=0;m<t.length;m++)
        arr[m+start1]=t[m];
}
```


Java 程序初始化顺序：

父类静态变量　　父类静态代码块
子类静态变量　　子类静态代码块
父类非静态变量　　父类非静态代码块　　父类构造函数
子类非静态变量　　子类非静态代码块　　子类构造函数

说说 JDK1.5 并发包

名称	作用
Lock	**锁**
Executors	**线程池**
ReentrantLock	**一个可重入的互斥锁定 Lock，功能类似 synchronized，**

	但要强大的多。
Condition	Condition 的功能类似于在传统的线程技术中的,Object.wait()和 Object.notify()的功能,
ConcurrentHashMap	分段 HasMap
AtomicInteger	原子类
BlockingQueue	BlockingQueue 通常用于一个线程生产对象,而另外一个线程消费这些对象的场景
ExecutorService	执行器服务

并发包

1）Executor、ExecutorSerice、AbstractExecutorSerice、ThreadPoolExecutor。

2）copyOnWriteArrayList、copyOnWriteSet。

3）BlockingQueue

4）CycleBarrier、CountDownLatch、Semophore

5）Future、callable

6）Lock

mysql 默认可重复

Java8 新特性:1）接口的默认方法，java 8 允许我们给接口添加一个非抽象方法，只需使用 default 关键字。2）lambda 表达式，在 java8 之前，若想将行为传入函数，仅有的选择是匿名类，而定义行为最重要的那行代码，却混在中间不够突出。lambda 表达式取代了匿名类，编码更清晰。3）函数式接口：指仅仅只有一个抽象方法的接口，每一个该类型的 lambda 表达式都会被匹配到这个抽象方法。每一个 lambda 表达式都对应一个类型，通常是接口类型，我们可以把 lambda 表达式当作任意只包含一个抽象方法的接口类型，为了确保接口一定达到这个要求（即有一个抽象方法），你只需要给你的接口加上 @FunctioalInterface 注释（编译器若发现标注了这个注释的接口有多于一个抽象方法，则报错）。4）lambda 作用域，在 lambda 表达式中访问外层作用域和老版本的匿名对象中的方法很相似，你可以直接访问标记了 final 的外层局部变量或实例的字段以及静态变量。lambda 表达式对外层局部变量只可读不可写，对类实例变量可读也可写。5）dateAPI：java8 在 java.time 包中包含一组全新日期 API。6）annotation 注释，java8 支持可重复注解，相同的注解可以在同一地方使用多次

锁的种类

自旋锁

自旋锁是采用让当前线程不停地的在循环体内执行实现的，当循环的条件被其他线程改变时 才能进入临界区。如下

```
public class SpinLock {
    private AtomicReference<Thread> sign = new AtomicReference<>();
    public void lock() {
        Thread current = Thread.currentThread();
        while (!sign.compareAndSet(null, current)) {
        }
    }
    public void unlock() {
```

```
            Thread current = Thread.currentThread();
            sign.compareAndSet(current, null);
        }
}
```

互斥锁

所谓互斥锁，指的是一次最多只能有一个线程持有的锁. 在jdk1.5之前，我们通常使用 synchronized 机制控制多个线程对共享资源 Lock 接口及其实现类 ReentrantLock

可重入锁

可重入锁，也叫做递归锁，指的是同一线程 外层函数获得锁之后 ，内层递归函数仍然有获取该锁的代码，但不受影响。
在 JAVA 环境下 ReentrantLock 和 synchronized 都是 可重入锁

信号量

信号量(Semaphore)，有时被称为信号灯，是在多线程环境下使用的一种设施，它负责协调各个线程，以保证它们能够正确、合理的使用公共资源。

synchronized（S） VS lock（L）

1） L 是接口，S 是关键字 2）S 在发生异常时，会自动释放线程占有的锁，不会发生死锁。L 在发生异常时，若没有主动通过 unlock（）释放锁，则很有可能造成死锁。所以用 lock 时要在 finally 中释放锁。3)L 可以当等待锁的线程响应中断，而 S 不行，使用 S 时，等待的线程将会一直等下去，不能响应中断。4）通过 L 可以知道是否成功获得锁，S 不可以。5）L 可以提高多个线程进行读写操作的效

非聚集索引与聚集索引

聚集索引：数据按索引顺序存储，叶子节点存储真实的数据行，不再有另外单独的数据页。在一张表上只能创建一个聚集索引，因为真实数据的物理顺序只能有 1 种，若一张表没有聚集索引，则他被称为堆集，这样表的数据行无特定的顺序，所有新行将被添加到表的末尾。非聚集索引与聚集索引的区别：1）叶子节点并非数据节点 2）叶子节点为每一个真正的数据行存储一个“键-指针”对 3）叶子节点中还存储了一个指针偏移量，根据页指针及指针偏移可以定位到具体的数据行。4）在除叶节点外的其他索引节点，存储的是类似内容，只不过是指向下一级索引页。

https

https 其实是由两部分组成：http+ssl/tls，也就是在 http 上又加了一层处理加密信息的模块，服务端和客户端的信息传输都会通过 tls 加密，传输的数据都是加密后的数据。加解密过程：1）客户端发起 https 请求（就是用户在浏览器里输入一个 https 网址，然后连接到 server 的 443 端口）2）服务端的配置（采用 https 协议的服务器必须要有一塔数字证书，可以自己制作，也可以向组织申请，这套证书就是一对公钥和私钥）。3）传输证书（这个证书就是公钥，只是包含了很多信息）4）客户端解析证书（由客户端 tls 完成，首先验证公钥是否有效，若发现异常，则弹出一个警示框，提示证书存在问题，若无问题，则生成一个随机值，然后用证书对随机值进行加密）5）传输加密信息（这里传输的是加密后的随机值，

目的是让服务端得到这个随机值，以后客户端和服务端的通信就可以通过这个随机值来进行加密了）6）服务端解密信息（服务端用私钥解密后得到了客户端传来的随机值，then 把内容通过该值进行对称加密。所谓对称加密就是，将信息和私钥通过某种算法混在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够彪悍，私钥够复杂，数据就够安全）7）传输加密的信息 8）客户端解密信息，用随机数来解。问：在客户端抓包抓到的是加密的还是没加密的？不知道是哪个面试官问的，我就乱说是加密的，然后面试官说错了，是没有加

怎么判断一个 mysql 中 select 语句是否使用了索引

可以在 select 语句前加上 explain，比如 explain select * from tablename;返回的一列中，若列名为 key 的那列为 null，则没有使用索引，若不为 null，则返回实际使用的索引名。让 select 强制使用索引的语法：select * from tablename from index(index_name

数据库要复习的地方

隔离级别；2）索引；3）范式[1NF：属性不可分；2NF：非主键属性完全依赖于主键属性；3NF：非主键属性之间无传递依赖；4NF：主键属性之间无传递依赖]；

URL 中可以存在中文吗？

可以，先将中文进行编码，tomcat 默认解码为 iso8859-1，这样就会出现乱码，我们可以再用 iso8859-1 进行编码，再用指定码表解码（post 和 get 都可以）。对于 post，可以使用 requset 的 setCharacterEncodeing 方法设置指定的解码表

现在需要测试系统的高并发性能，如何模拟高并发？

使用 cyclicbarrier，cyclicbarrier 初始化时规定一个数目，then 计算调用了 cyclicbarrier.await()进入等待的线程数，当线程数达到这个数目时，所有进入等待状态的线程将被唤醒并继续。Cyclic 就像他的名字一样，可看成是一个屏障，所有线程必须到达后才可以一起通过这个屏障。

jdk1.8bin 目录下的东西

Java.exe javac.exe Javadoc.exe jar.exe jstack.exe (打印所有 java 线程堆栈跟踪信息)

若发现 sql 语句执行很慢

怎么找出这条 sql 语句查

什么是 Socket？

Socket 就是为网络服务提供的一种机制。
通讯的两端都有 Sokcet
网络通讯其实就是 Sokcet 间的通讯
数据在两个 Sokcet 间通过 IO 传输。

接口和抽象类的区别？

异：抽象类可以有构造方法和普通成员变量，而接口不行。
一个类可以实现多个接口，但是只能继承一个类
同：①都不能被实例化，可以将接口或抽象类作为引用类型
②如果某个类继承了抽象类或者实现了某个接口，必须重写里面的抽象方法，否则该类还被声明为抽象的
③接口（1.8 引进）和抽象类可以有静态方法
④接口（1.9 引进）和抽象方法可以有私有方法
接口默认成员变量： public static final
接口默认类是：public abstract

Integer 和 int 的区别

①Integer 是 int 的包装类，int 是 java 的基本数据类型
②Integer 变量必须实例化才能使用，int 不需要
③Integer 实际是对象的引用，new 一个 Integer，实际生成一个指针指向该对象，int 直接存放数据值
④Integer 的默认值是 null，int 默认值是 0

Integer

Integer 里有一个静态内部类，里面有一个 cache[]可缓存的数组，默认情况下[-128---127],大小可以通过 JVM 参数设置，(byte 类型的范围)，在第一次使用时(类加载)被自动装箱在 cache 数组里。

使用 Integer.valueOf 进行构造时，就使用了 cache[]缓存数组，如果用此方法的对象在缓存区间内，就返回同一个引用，自然就是同一个对象，如果不在此区间或者 new 一个 Integer，不使用缓存，直接生成一个新的对象。

反射：

反射就是把 java 的各种成分映射成一个对象，反射首先获得字节码：
①Class.forName("类的路径")
②类名.Class
③实例.getClass
优点：扩展性高缺点：初始化对象效率低
用处：①jdbc:Class.forName("com.mysql.jdbc.Driver.class")
②mybatis ORM 映射
③spring:IOC,<bean id="" class="">
④自定义注解
无参：Class<?> forName=Class.forName("cm.jixuan.User");
User user=forName.newInstance();
有参： Class<?> forName=Class.forName("cm.jixuan.User");
Class<?> constructor=forName.getConstructor(String.class)(clazz);
User user=(User)constructor.newInstance("12","12");
禁止 Java 反射：将构造函数私有化 public--->private

创建对象的方式：

①、使用 new 关键字

②、使用反射机制
③、使用 clone 方法
④、使用反序列化

CGLIB 与 JDK 动态代理区别

jdk 动态代理是由 Java 内部的反射机制来实现的，cglib 动态代理底层则是借助 asm 来实现的。总的来说，反射机制在生成类的过程中比较高效，而 asm 在生成类之后的相关执行过程中比较高效（可以通过将 asm 生成的类进行缓存，这样解决 asm 生成类过程低效问题）。还有一点必须注意：jdk 动态代理的应用前提，必须是目标类基于统一的接口。如果没有上述前提，jdk 动态代理不能应用。

注:asm 其实就是 java 字节码控制.

注解分类：内置注解(也成为元注解 jdk 自带注解)、自定义注解（Spring 框架）

什么是数据交换格式？

客户端与服务器常用数据交换格式 xml、json、html

JSON 解析框架有哪些？

fastjson(阿里)、gson(谷歌)、jackson(SpringMVC 自带)

XML 解析方式？

Dom4j、Sax、Pull

Dom4j 与 Sax 区别

dom4j 不适合大文件的解析，因为它是一下子将文件加载到内存中，所以有可能出现内存溢出，sax 是基于事件来对 xml 进行解析的，所以他可以解析大文件的 xml，也正是因为如此，所以 dom4j 可以对 xml 进行灵活的增删改查和导航，而 sax 没有这么强的灵活性，所以 sax 经常是用来解析大型 xml 文件，而要对 xml 文件进行一些灵活（crud）操作就用 dom4j。

反射机制获取类有三种方法

```
//第一种方式：
 Classc1 = Class.forName("Employee");
 //第二种方式：
 //java中每个类型都有class 属性.
 Classc2 = Employee.class;
 //第三种方式：
 //java语言中任何一个java对象都有getClass 方法
 Employeee = new Employee();
 Classc3 = e.getClass(); //c3是运行时类 (e的运行时类是Employee)
```

反射创建对象的方式

```
Class<?> forName = Class.forName("com.itmayiedu.entity.User");
     // 创建此Class 对象所表示的类的一个新实例 调用了User的无参数构造方法.
```

```
Object newInstance = forName.newInstance();
```

实例化有参构造函数

```
Class<?> forName = Class.forName("com.itmayiedu.entity.User");
Constructor<?> constructor = forName.getConstructor(String.class, String.class);
User newInstance = (User) constructor.newInstance("123", "123");
```

反射创建 api

方法名称	作用
getDeclaredMethods []	获取该类的所有方法
getReturnType()	获取该类的返回值
getParameterTypes()	获取传入参数
getDeclaredFields()	获取该类的所有字段
setAccessible	允许访问私有成员

使用反射为类私有属性赋值

```
// 获取当前类class地址
Class<?> forName = Class.forName("com.itmayiedu.entity.User");
// 使用反射实例化对象 无参数构造函数
Object newInstance = forName.newInstance();
// 获取当前类的 userId字段
Field declaredField = forName.getDeclaredField("userId");
// 允许操作私有成员
declaredField.setAccessible(true);
// 设置值
declaredField.set(newInstance, "123");
User user = (User) newInstance;
System.out.println(user.getUserId());
```

什么是虚拟机参数配置

在虚拟机运行的过程中，如果可以跟踪系统的运行状态，那么对于问题的故障

排查会有一定的帮助，为此，在虚拟机提供了一些跟踪系统状态的参数，使用

给定的参数执行 Java 虚拟机，就可以在系统运行时打印相关日志，用于分析实际

问题。我们进行虚拟机参数配置，其实就是围绕着堆、栈、方法区、进行配置。

堆的参数配置

-XX:+PrintGC 每次触发 GC 的时候打印相关日志
-XX:+UseSerialGC 串行回收
-XX:+PrintGCDetails 更详细的 GC 日志
-Xms 堆初始值
-Xmx 堆最大可用值
-Xmn 新生代堆最大可用值
-XX:SurvivorRatio 用来设置新生代中 eden 空间和 from/to 空间的比例.
含以-XX:SurvivorRatio=eden/from=den/to
总结:在实际工作中，我们可以直接将初始的堆大小与最大堆大小相等，

这样的好处是可以减少程序运行时垃圾回收次数，从而提高效率。

-XX:SurvivorRatio 用来设置新生代中 eden 空间和 from/to 空间的比例.

设置新生代与老年代优化参数

-Xmn 新生代大小，一般设为整个堆的 1/3 到 1/4 左右
-XX:SurvivorRatio 设置新生代中 eden 区和 from/to 空间的比例关系 n/1

设置新生代比例参数

参数: -Xms20m -Xmx20m -Xmn1m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC

```
public class JvmDemo02 {
      public static void main(String[] args) {
          //-Xms20m -Xmx20m -Xmn1m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC
           byte [] b = null;
           for (int i = 0; i < 10; i++) {
               b =new byte[1*1024*1024];
           }
      }
}
```

设置新生与老年代代参数

-Xms20m -Xmx20m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC
-Xms20m -Xmx20m -XX:SurvivorRatio=2 -XX:+PrintGCDetails -XX:+UseSerialGC
-XX:NewRatio=2
总结:不同的堆分布情况，对系统执行会产生一定的影响，在实际工作中，
应该根据系统的特点做出合理的配置，基本策略：尽可能将对象预留在新生代，
减少老年代的 GC 次数。
除了可以设置新生代的绝对大小(-Xmn),可以使用(-XX:NewRatio)设置新生代和老年
代的比例:-XX:NewRatio=老年代/新生代

内存溢出解决办法

设置堆内存大小

错误原因: java.lang.OutOfMemoryError: Java heap space
解决办法：设置堆内存大小 -Xms1m -Xmx70m -XX:+HeapDumpOnOutOfMemoryError

```
     public static void main(String[] args) throws InterruptedException {
          List<Object> list = new ArrayList<>();
          Thread.sleep(3000);
          jvmInfo();
          for (int i = 0; i < 10; i++) {
               System.out.println("i:"+i);
               Byte [] bytes= new Byte[1*1024*1024];
               list.add(bytes);
```

```
            jvmInfo();
        }
        System.out.println("添加成功...");
    }
```

设置栈内存大小

错误原因: java.lang.StackOverflowError

栈溢出 产生于递归调用，循环遍历是不会的，但是循环方法里面产生递归调用， 也会发生栈溢出。

解决办法:设置线程最大调用深度

-Xss5m 设置最大调用深度

```
public class JvmDemo04 {
     private static int count;
     public static void count(){
         try {
               count++;
               count();
         } catch (Throwable e) {
              System.out.println("最大深度:"+count);
              e.printStackTrace();
         }
     }
     public static void main(String[] args) {
          count();
     }
}
```

Tomcat 内存溢出在 catalina.sh 修改 JVM 堆内存大小

```
JAVA_OPTS="-server -Xms800m -Xmx800m -XX:PermSize=256m -XX:MaxPermSize=512m -XX:MaxNewSize=512m"
```

JVM 参数调优总结

在 JVM 启动参数中，可以设置跟内存、垃圾回收相关的一些参数设置，默认情况不做任何设置 JVM 会工作的很好，但对一些配置很好的 Server 和具体的应用必须仔细调优才能获得最佳性能。通过设置我们希望达到一些目标：

GC 的时间足够的小

GC 的次数足够的少

发生 Full GC 的周期足够的长

前两个目前是相悖的，要想 GC 时间小必须要一个更小的堆，要保证 GC 次数足够少，必须保证一个更大的堆，我们只能取其平衡。

（1）针对 JVM 堆的设置，一般可以通过-Xms -Xmx 限定其最小、最大值，为了防止垃圾收集器在最小、最大之间收缩堆而产生额外的时间，我们通常把最大、最小设置为相同的值

（2）年轻代和年老代将根据默认的比例（1：2）分配堆内存，可以通过调整二者之间的比率 NewRadio 来调整二者之间的大小，也可以针对回收代，比如年轻代，通过 -XX:newSize -XX:MaxNewSize 来设置其绝对大小。同样，为了防

止年轻代的堆收缩，我们通常会把-XX:newSize -XX:MaxNewSize 设置为同样大小

（3）年轻代和年老代设置多大才算合理？这个我问题毫无疑问是没有答案的，否则也就不会有调优。我们观察一下二者大小变化有哪些影响

更大的年轻代必然导致更小的年老代，大的年轻代会延长普通 GC 的周期，但会增加每次 GC 的时间；小的年老代会导致更频繁的 Full GC

更小的年轻代必然导致更大年老代，小的年轻代会导致普通 GC 很频繁，但每次的 GC 时间会更短；大的年老代会减少 Full GC 的频率

如何选择应该依赖应用程序对象生命周期的分布情况：如果应用存在大量的临时对象，应该选择更大的年轻代；如果存在相对较多的持久对象，年老代应该适当增大。但很多应用都没有这样明显的特性，在抉择时应该根据以下两点：(A)本着 Full GC 尽量少的原则，让年老代尽量缓存常用对象，JVM 的默认比例 1：2 也是这个道理 (B)通过观察应用一段时间，看其他在峰值时年老代会占多少内存，在不影响 Full GC 的前提下，根据实际情况加大年轻代，比如可以把比例控制在 1：1。但应该给年老代至少预留 1/3 的增长空间

调优总结

初始堆值和最大堆内存内存越大，吞吐量就越高。

最好使用并行收集器,因为并行手机器速度比串行吞吐量高，速度快。

设置堆内存新生代的比例和老年代的比例最好为 1:2 或者 1:3。

减少 GC 对老年代的回收。

MySQL 如何优化

表的设计合理化(符合 3NF)

添加适当索引(index) [四种：普通索引、主键索引、唯一索引 unique、全文索引]

SQL 语句优化

分表技术(水平分割、垂直分割)

读写[写: update/delete/add]分离

存储过程 [模块化编程，可以提高速度]

对 mysql 配置优化 [配置最大并发数 my.ini, 调整缓存大小]

mysql 服务器硬件升级

定时的去清除不需要的数据,定时进行碎片整理(MyISAM)

数据库三大范式

第一范式：1NF 是对属性的原子性约束，要求属性(列)具有原子性，不可再分解；(只要是关系型数据库都满足 1NF)

第二范式：2NF 是对记录的惟一性约束，表中的记录是唯一的，就满足 2NF，通常我们设计一个主键来实现，主键不能包含业务逻辑。

第三范式：3NF 是对字段冗余性的约束，它要求字段没有冗余。 没有冗余的数据库设计可以做到。

但是，没有冗余的数据库未必是最好的数据库，有时为了提高运行效率，就必须降低范式标准，适当保留冗余数据。具体做法是： 在概念数据模型设计时遵守第三范式，降低范式标准的工作放到物理数据模型设计时考虑。降低范式就是增加字段，允许冗余。

慢查询

什么是慢查询

MySQL 默认 10 秒内没有响应 SQL 结果,则为慢查询
可以去修改 MySQL 慢查询默认时间

如何修改慢查询

```
--查询慢查询时间
show variables like 'long_query_time';
--修改慢查询时间
set long_query_time=1; ---但是重启 mysql 之后 ,long_query_time 依然是 my.ini 中的值
```

如何将慢查询定位到日志中

在默认情况下，我们的 mysql 不会记录慢查询，需要在启动 mysql 时候，指定记录慢查询才可以

bin₩mysqld.exe --safe-mode --slow-query-log [mysql5.5 可以在 my.ini 指定]（安全模式启动，数据库将操作写入日志，以备恢复）

bin₩mysqld.exe –log-slow-queries=d:/abc.log [低版本 mysql5.0 可以在 my.ini 指定]

先关闭 mysql,再启动，如果启用了慢查询日志，默认把这个文件放在
my.ini 文件中记录的位置
#Path to the database root
datadir=" C:/ProgramData/MySQL/MySQL Server 5.5/Data/"

存储引擎	允许的索引类型
myisam	btree
innodb	btree
memory/yeap	Hash,btree

查询所用使用率

show status like ‘handler_read%’;
大家可以注意：
handler_read_key:这个值越高越好，越高表示使用索引查询到的次数。
handler_read_rnd_next:这个值越高，说明查询低效。

SQL 调优

① 使用 group by 分组查询是，默认分组后，还会排序，可能会降低速度，
在 group by 后面增加 order by null 就可以防止排序.

```
explain select * from emp  group by deptno order by null;
```

② 有些情况下，可以使用连接来替代子查询。因为使用 join，MySQL 不需要在内存中创建临时表。

```
select * from dept, emp where dept.deptno=emp.deptno; [简单处理方式]
```

select * from dept left join emp on dept.deptno=emp.deptno; [左外连接，更ok!]

③ 对查询进行优化，要尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引

应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num is null

最好不要给数据库留 NULL，尽可能的使用 NOT NULL 填充数据库.

备注、描述、评论之类的可以设置为 NULL，其他的，最好不要使用 NULL。

不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了， 不管是否插入值（NULL 也包含在内），都是占用 100 个字符的空间的，如果是 varchar 这样的变长字段， null 不占用空间。

可以在 num 上设置默认值 0，确保表中 num 列没有 null 值，然后这样查询：

select id from t where num = 0

更多 mysql sql 语句调优查看 http://bbs.itmayiedu.com/article/1511164574773

Myisam 注意事项

如果你的数据库的存储引擎是 myisam,请一定记住要定时进行碎片整理

举例说明:

create table test100(id int unsigned ,name varchar(32))engine=myisam;

insert into test100 values(1,'aaaaa');

insert into test100 values(2,'bbbb');

insert into test100 values(3,'ccccc');

我们应该定义对 myisam 进行整理

optimize table test100;

数据库数据备份

手动方式

cmd 控制台:

在环境变量中配置 mysql 环境变量

mysqldump –u -账号 –密码 数据库 [表名 1 表名 2..] > 文件路径

案例: mysqldump -u -root root test > d:₩temp.sql

比如: 把 temp 数据库备份到 d:₩temp.bak

mysqldump -u root -proot test > f:₩temp.bak

如果你希望备份是，数据库的某几张表

mysqldump -u root -proot test dept > f:₩temp.dept.sql

如何使用备份文件恢复我们的数据.

mysql 控制台

source d:₩temp.dept.bak

自动方式

把备份数据库的指令，写入到 bat 文件，然后通过任务管理器去定时调用 bat 文件.

mytask.bat 内容是：

```
@echo off
F:\path\mysqlanzhuang\bin\mysqldump -u root -proot test dept > f:\temp.dept.sql
```

创建执行计划任务执行脚本。

分表分库

垂直拆分

垂直拆分就是要把表按模块划分到不同数据库表中（当然原则还是不破坏第三范式），这种拆分在大型网站的演变过程中是很常见的。当一个网站还在很小的时候，只有小量的人来开发和维护，各模块和表都在一起，当网站不断丰富和壮大的时候，也会变成多个子系统来支撑，这时就有按模块和功能把表划分出来的需求。其实，相对于垂直切分更进一步的是服务化改造，说得简单就是要把原来强耦合的系统拆分成多个弱耦合的服务，通过服务间的调用来满足业务需求看 ,因此表拆出来后要通过服务的形式暴露出去 ,而不是直接调用不同模块的表，淘宝在架构不断演变过程，最重要的一环就是服务化改造，把用户、交易、店铺、宝贝这些核心的概念抽取成独立的服务，也非常有利于进行局部的优化和治理，保障核心模块的稳定性

垂直拆分用于分布式场景。

水平拆分

上面谈到垂直切分只是把表按模块划分到不同数据库，但没有解决单表大数据量的问题 ,而水平切分就是要把一个表按照某种规则把数据划分到不同表或数据库里。例如像计费系统，通过按时间来划分表就比较合适，因为系统都是处理某一时间段的数据。而像 SaaS 应用，通过按用户维度来划分数据比较合适，因为用户与用户之间的隔离的，一般不存在处理多个用户数据的情况，简单的按 user_id 范围来水平切分

通俗理解：水平拆分行，行数据拆分到不同表中， 垂直拆分列，表数据拆分到不同表中

水平分割案例

思路:在大型电商系统中，每天的会员人数不断的增加。达到一定瓶颈后如何优化查询。

可能大家会想到索引，万一用户量达到上亿级别，如何进行优化呢？

使用水平分割拆分数据库表。

如何使用水平拆分数据库

使用水平分割拆分表，具体根据业务需求，有的按照注册时间、取摸、账号规则、年份等。

什么是读写分离

在数据库集群架构中 ,让主库负责处理事务性查询 ,而从库只负责处理 select 查询，让两者分工明确达到提高数据库整体读写性能。当然，主数据库另外一个

功能就是负责将事务性查询导致的数据变更同步到从库中，也就是写操作。

读写分离的好处

1）分摊服务器压力，提高机器的系统处理效率

2）增加冗余，提高服务可用性，当一台数据库服务器宕机后可以调整另外一台从库以最快速度恢复服务

主从复制原理

依赖于二进制日志，binary-log.

二进制日志中记录引起数据库发生改变的语句

Insert 、delete、update、create table

环境搭建

1. 准备环境

   两台 windows 操作系统 ip 分别为: 172.27.185.1(主)、172.27.185.2(从)

2. 连接到主服务(172.27.185.1)服务器上，给从节点分配账号权限。

   GRANT REPLICATION SLAVE ON *.* TO 'root'@'172.27.185.2' IDENTIFIED BY 'root';

3. 在主服务 my.ini 文件新增

```
server-id=200
log-bin=mysql-bin
relay-log=relay-bin
relay-log-index=relay-bin-index
```

   重启 mysql 服务

4. 在从服务 my.ini 文件新增

```
server-id = 210
replicate-do-db =itmayiedu #需要同步数据库
```

   重启 mysql 服务

5. 从服务同步主数据库

```
stop slave;
change
master to master_host='172.27.185.1',master_user='root',master_password='root';
start slave;
show slave status;
```

什么是 Mycat

是一个开源的分布式数据库系统，但是因为数据库一般都有自己的数据库引擎，而 Mycat 并没有属于自己的独有数据库引擎，所有严格意义上说并不能算是一个完整的数据库系统，只能说是一个在应用和数据库之间起桥梁作用的中间件。

在 Mycat 中间件出现之前，MySQL 主从复制集群，如果要实现读写分离，一般是在程序段实现，这样就带来了一个问题，即数据段和程序的耦合度太高，如果数据库的地址发生了改变，那么我的程序也要进行相应的修改，如果数据库不小心挂掉了，则同时也意味着程序的不可用，而对于很多应用来说，并不能接

受；

引入 Mycat 中间件能很好地对程序和数据库进行解耦，这样，程序只需关注数据库中间件的地址，而无需知晓底层数据库是如何提供服务的，大量的通用数据聚合、事务、数据源切换等工作都由中间件来处理；

Mycat 中间件的原理是对数据进行分片处理，从原有的一个库，被切分为多个分片数据库，所有的分片数据库集群构成完成的数据库存储，有点类似磁盘阵列中的 RAID0.

静态资源和动态资源的区别

静态资源： 当用户多次访问这个资源，资源的源代码永远不会改变的资源。

动态资源：当用户多次访问这个资源，资源的源代码可能会发送改变。

Sevlet 的生命周期（重点）

构造方法： 创建 servlet 对象的时候调用。

默认情况下，第一次访问 servlet 的时候创建 servlet 对象

只调用 1 次。证明 servlet 对象在 tomcat 是单实例的。

init 方法： 创建完 servlet 对象的时候调用。只调用 1 次。

service 方法： 每次发出请求时调用。调用 n 次。

destroy 方法： 销毁 servlet 对象的时候调用。停止服务器或者重新部署 web 应用时销毁 servlet 对象。只调用 1 次。

怎么证明 Servlet 是单例的？

因为 Servlet 是通过 Java 反射机制，读取 web.xml 配置中的 servlet-class 完整路径，进行反射默认执行无参构造函数，所以只要 servlet 类执行无参构造函数永远只执行一遍，则 Servlet 是单例的。

Servlet 的多线程并发问题

注意： servlet 对象在 tomcat 服务器是单实例多线程的。

因为 servlet 是多线程的，所以当多个 servlet 的线程同时访问了 servlet 的共享数据，如成员变量，可能会引发线程安全问题。

解决办法：

1）把使用到共享数据的代码块进行同步（使用 synchronized 关键字进行同步）

2）建议在 servlet 类中尽量不要使用成员变量。如果确实要使用成员，必须同步。而且尽量缩小同步代码块的范围。（哪里使用到了成员变量，就同步哪里！！），以避免因为同步而导致并发效率降低。

转发与重定向区别？

1）转发

a）地址栏不会改变

b）转发只能转发到当前 web 应用内的资源

c）可以在转发过程中，可以把数据保存到 request 域对象中

2）重定向

a）地址栏会改变，变成重定向到地址。

b）重定向可以跳转到当前 web 应用，或其他 web 应用，甚至是外部域名网站。

c）不能再重定向的过程，把数据保存到 request 中。

重定向实现原理

重定向会发送两次请求,浏览器认识状态码为 302，会再次向服务器发送一次请求，获取请求头的中 location 的 value 值进行重定向。

JavaWeb 有哪些会话技术

Cookie 会话数据保存在浏览器客户端

服务器创建 Cookie,将 Cookie 内容以响应头方式发送给客户端存放在本地,当下次发送请求时.会将 Cookie 信息以请求方式发送给服务器端

注意:Cookie 信息不能夸浏览器访问

Session 会话保存与服务器端

服务器创建 Session,Session 内容存放服务器端上,以响应头方式将 SessionId 发送给客户端保存，当下次发送请求时,会将 SessionID 以请求头方式发送给服务器端

注意: 浏览器关闭,只是清除了 Sessionid，并没有清除 Session

解决 java 集群的 session 共享的解决方案：

1.客户端 cookie 加密。（一般用于内网中企业级的系统中，要求用户浏览器端的 cookie 不能禁用，禁用的话，该方案会失效）。

2.集群中，各个应用服务器提供了 session 复制的功能，tomcat 和 jboss 都实现了这样的功能。特点 :性能随着服务器增加急剧下降 ,容易引起广播风暴 ;session 数据需要序列化，影响性能。

3.session 的持久化，使用数据库来保存 session。就算服务器宕机也没事儿，数据库中的 session 照样存在。特点：每次请求 session 都要读写数据库，会带来性能开销。使用内存数据库，会提高性能，但是宕机会丢失数据(像支付宝的宕机，有同城灾备、异地灾备)。

4.使用共享存储来保存 session。和数据库类似，就算宕机了也没有事儿。其实就是专门搞一台服务器，全部对 session 落地。特点：频繁的进行序列化和反序列化会影响性能。

5.使用 memcached 来保存 session。本质上是内存数据库的解决方案。特点：存入 memcached 的数据需要序列化，效率极低。

Cookie 的实现原理

1）服务器创建 cookie 对象，把会话数据存储到 cookie 对象中。

new Cookie("name","value");

2）服务器发送 cookie 信息到浏览器 response.addCookie(cookie);举例：set-cookie: name=eric (隐藏发送了一个 set-cookie 名称的响应头)

3）浏览器得到服务器发送的 cookie，然后保存在浏览器端。

4）浏览器在下次访问服务器时，会带着 cookie 信息 举例： cookie: name=eric (隐藏带着一个叫 cookie 名称的请求头)

5）服务器接收到浏览器带来的 cookie 信息 request.getCookies();

Cookie 的应用场景

购物车、显示用户上次访问的时间

Session 的实现原理

1）第一次访问创建 session 对象，给 session 对象分配一个唯一的 ID，叫 JSESSIONID

new HttpSession();

2）把 JSESSIONID 作为 Cookie 的值发送给浏览器保存

Cookie cookie = new Cookie("JSESSIONID", sessionID);
response.addCookie(cookie);

3）第二次访问的时候，浏览器带着 JSESSIONID 的 cookie 访问服务器

4）服务器得到 JSESSIONID，在服务器的内存中搜索是否存放对应编号的 session 对象。

if(找到){
    return map.get(sessionID);
}
Map<String,HttpSession>]

<"s001", s1>
<"s001,"s2>

5）如果找到对应编号的 session 对象，直接返回该对象

6）如果找不到对应编号的 session 对象，创建新的 session 对象，继续走 1 的流程

结论 通过 JSESSION 的 cookie 值在服务器找 session 对象 !!!!!

什么是 token

token 其实就是一个令牌,具有随机性,类似于 sessionId。

在对接一些第三方平台的时候,为了能够保证数据安全性,通常会使用一些令牌进行交互

如何自定义 token

token 生成规则,只要保证 token 生成一个不重复的唯一字符串即可。

使用 jdk 自带的 uuid 生成规则。

什么是 UUID

UUID 含义是通用唯一识别码 (Universally Unique Identifier)，这是一个软件建构的标准，也是被开源软件基金会 (Open Software Foundation, OSF) 的组织应用在分布式计算环境 (Distributed Computing Environment, DCE) 领域的一部分。

UUID 的目的，是让分布式系统中的所有元素，都能有唯一的辨识资讯，而不需要透过中央控制端来做辨识资讯的指定。如此一来，每个人都可以建立不与其它人冲突的 UUID。

在这样的情况下，就不需考虑数据库建立时的名称重复问题。目前最广泛应用的 UUID，即是微软的 Microsoft's Globally Unique Identifiers (GUIDs)，而其他重要的应用，

则有 Linux ext2/ext3 档案系统、LUKS 加密分割区、GNOME、KDE、Mac OS X 等等

UUID 组成

UUID 保证对在同一时空中的所有机器都是唯一的。通常平台会提供生成的 API。按照开放软件基金会(OSF)制定的标准计算，用到了以太网卡地址、纳秒级时间、芯片 ID 码和许多可能的数字

UUID 由以下几部分的组合：

（1）当前日期和时间，UUID 的第一个部分与时间有关，如果你在生成一个 UUID 之后，过几秒又生成一个 UUID，则第一个部分不同，其余相同。

（2）时钟序列。

（3）全局唯一的 IEEE 机器识别号，如果有网卡，从网卡 MAC 地址获得，没有网卡以其他方式获得。

UUID 的唯一缺陷在于生成的结果串会比较长。关于 UUID 这个标准使用最普遍的是微软的 GUID(Globals Unique Identifiers)。在 ColdFusion 中可以用 CreateUUID()函数很简单地生成 UUID，

其格式为：xxxxxxxx-xxxx- xxxx-xxxxxxxxxxxxxxxx(8-4-4-16)，其中每个 x 是 0-9 或 a-f 范围内的一个十六进制的数字。而标准的 UUID 格式为：xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx (8-4-4-4-12);

UUID 代码

```
UUID.randomUUID().toString()
```

什么是 Filter

Filter 是 Servlet 技术中的过滤器,主要做拦截请求作用，一般用于防御 XSS 攻击、权限、登录判断等。

http 部分

什么是 http 协议

http 协议： 对浏览器客户端 和 服务器端 之间数据传输的格式规范

Http 格式的分类

请求行
请求头
请求内容
响应行
响应头
响应内容

https(2443)与 http(80)区别

（1）HTTPS 协议握手阶段比较费时，会使页面的加载时间延长近 50%，增加 10% 到 20% 的耗电；

（2）HTTPS 连接缓存不如 HTTP 高效，会增加数据开销和功耗，甚至已有的安全措施也会因此而受到影响；

（3）SSL 证书需要钱，功能越强大的证书费用越高，个人网站、小网站没有必要一般不会用。

（4）SSL 证书通常需要绑定 IP，不能在同一 IP 上绑定多个域名，IPv4 资源不可能支撑这个消耗。

（5）HTTPS 协议的加密范围也比较有限，在黑客攻击、拒绝服务攻击、服务器劫持等方面几乎起不到什么作用。最关键的，SSL 证书的信用链体系并不安全，特别是在某些国家可以控制 CA 根证书的情况下，中间人攻击一样可行。

https 请求方式

常见的请求方式： GET 、 POST、 HEAD、 TRACE、 PUT、 CONNECT 、 DELETE

常用的请求方式： GET 和 POST

表单提交：

```
<form action="提交地址" method="GET/POST">
<form>
```

GET vs POST 区别

1）GET 方式提交

a）地址栏（URI）会跟上参数数据。以？开头，多个参数之间以&分割。

```
GET /day09/testMethod.html?name=eric&password=123456 HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:35.0) Gecko/20100101 Firefox/35.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-cn,en-us;q=0.8,zh;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://localhost:8080/day09/testMethod.html
Connection: keep-alive
```

b）GET 提交参数数据有限制，不超过 1KB。

c）GET 方式不适合提交敏感密码。

d）注意： 浏览器直接访问的请求，默认提交方式是 GET 方式

2）POST 方式提交

a）参数不会跟着 URI 后面。参数而是跟在请求的实体内容中。没有？开头，多个参数之间以&分割。

```
POST /day09/testMethod.html HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:35.0) Gecko/20100101 Firefox/35.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-cn,en-us;q=0.8,zh;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://localhost:8080/day09/testMethod.html
Connection: keep-alive

name=eric&password=123456
```

b）POST 提交的参数数据没有限制。

c）POST 方式提交敏感数据。

3.2 请求头

```
Accept: text/html,image/*       -- 浏览器接受的数据类型
Accept-Charset: ISO-8859-1      -- 浏览器接受的编码格式
Accept-Encoding: gzip,compress  --浏览器接受的数据压缩格式
Accept-Language: en-us,zh-       --浏览器接受的语言
Host: www.it315.org:80           --（必须的）当前请求访问的目标地址（主机:端口）
If-Modified-Since: Tue, 11 Jul 2000 18:23:51 GMT  --浏览器最后的缓存时间
Referer: http://www.it315.org/index.jsp       -- 当前请求来自于哪里
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)  --浏览器类型
Cookie:name=eric                        -- 浏览器保存的 cookie 信息
Connection: close/Keep-Alive                -- 浏览器跟服务器连接状态。close: 连接关闭
keep-alive：保存连接。
Date: Tue, 11 Jul 2000 18:23:51 GMT       -- 请求发出的时间
```

客户端模拟 http 请求工具

Postmen(谷歌插件)、RestClient

服务器模拟 http 请求工具

httpclient、HttpURLConnection

前端 ajax 请求

```
$.ajax({
                type : 'post',
                dataType : "text",
                url : "http://a.a.com/a/FromUserServlet",
                data : "userName=余胜军&userAge=19",
                success : function(msg) {
                    alert(msg);
                }
            });
```

Spring 概述

Spring 框架，可以解决对象创建以及对象之间依赖关系的一种框架。

且可以和其他框架一起使用；Spring 与 Struts，Spring 与 hibernate

(起到整合（粘合）作用的一个框架)

Spring 提供了一站式解决方案：

1） Spring Core　spring 的核心功能： IOC 容器, 解决对象创建及依赖关系

2） Spring Web　Spring 对 web 模块的支持。

-→ 可以与 struts 整合,让 struts 的 action 创建交给 spring

-→ spring mvc 模式

3） Spring DAO　Spring 对 jdbc 操作的支持 【JdbcTemplate 模板工

具类】

4） Spring ORM　spring 对 orm 的支持：

→ 既可以与 hibernate 整合，【session】

→ 也可以使用 spring 的对 hibernate 操作的封装

5）Spring AOP　切面编程

6）SpringEE　spring 对 javaEE 其他模块的支持

SpringBeanId 重复会怎么样？

会报错。

什么是 SpringIOC

SpringIOC 就是把每个 bean 与 bean 之间的关系交给第三方容器进行管理，这个容器就是 Spring。

什么是 SpringAOP

Aop， aspect object programming　面向切面编程

功能：　让关注点代码与业务代码分离！

关注点，

重复代码就叫做关注点；

切面，

关注点形成的类，就叫切面(类)！

面向切面编程，就是指 对很多功能都有的重复的代码抽取，再在运行的时候网业务方法上动态植入“切面类代码”。

切入点，

执行目标对象方法，动态植入切面代码。

可以通过切入点表达式，指定拦截哪些类的哪些方法；给指定的类在运行的时候植入切面类代码。

应用场景:事物、日志、权限控制

SpringAOP 创建方式

注解方式实现 AOP 编程

步骤：

1）先引入 aop 相关 jar 文件　（aspectj　aop 优秀组件）

2）bean.xml 中引入 aop 名称空间

3）开启 aop 注解

4) 使用注解

@Aspect　指定一个类为切面类

@Pointcut("execution(* com.itmayiedu.service.UserService.add(..))")　指定切入点表达式

@Before("pointCut_()")　前置通知: 目标方法之前执行

@After("pointCut_()") 后置通知：目标方法之后执行（始终执行）

@AfterReturning("pointCut_()") 返回后通知：执行方法结束前执行(异常不执行)

@AfterThrowing("pointCut_()") 异常通知: 出现异常时候执行

@Around("pointCut_()") 环绕通知： 环绕目标方法执行

```
@Component
@Aspect
public class Aop {
     @Before("execution(* com.itmayiedu.service.UserService.add(..))")
     public void begin() {
          System.out.println("前置通知");
     }
     @After("execution(* com.itmayiedu.service.UserService.add(..))")
     public void commit() {
          System.out.println("后置通知");
     }
     @AfterReturning("execution(* com.itmayiedu.service.UserService.add(..))")
     public void afterReturning() {
          System.out.println("运行通知");
     }
     @AfterThrowing("execution(* com.itmayiedu.service.UserService.add(..))")
     public void afterThrowing() {
          System.out.println("异常通知");
     }
     @Around("execution(* com.itmayiedu.service.UserService.add(..))")
     public void around(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
      System.out.println("我是环绕通知-前");
      proceedingJoinPoint.proceed();
      System.out.println("我是环绕通知-后");
     }

}
```

XML 方式实现 AOP 编程

1） 引入 jar 文件 【aop 相关 jar， 4 个】

2） 引入 aop 名称空间

3）aop 配置

* 配置切面类 （重复执行代码形成的类）

* aop 配置

拦截哪些方法 / 拦截到方法后应用通知代码

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      .........
       http://www.springframework.org/schema/aop/spring-aop.xsd">

     <!-- dao 实例 -->
     <bean id="userDao" class="com.itmayiedu.UserDao"></bean>
     <bean id="orderDao" class="com.itmayiedu.OrderDao"></bean>

     <!-- 切面类 -->
     <bean id="aop" class="com.itmayiedu.Aop"></bean>

     <!-- Aop配置 -->
     <aop:config>
          <!-- 定义一个切入点表达式： 拦截哪些方法 -->
          <aop:pointcut expression="execution(* com.itmayiedu.*.*(..))" id="pt"/>
          <!-- 切面 -->
          <aop:aspect ref="aop">
```

```
            <!-- 环绕通知 -->
            <aop:around method="around" pointcut-ref="pt"/>
            <!-- 前置通知： 在目标方法调用前执行 -->
            <aop:before method="begin" pointcut-ref="pt"/>
            <!-- 后置通知： -->
            <aop:after method="after" pointcut-ref="pt"/>
            <!-- 返回后通知 -->
            <aop:after-returning method="afterReturning" pointcut-ref="pt"/>
            <!-- 异常通知 -->
            <aop:after-throwing method="afterThrowing" pointcut-ref="pt"/>

        </aop:aspect>
    </aop:config>
</beans>
```

Spring 是单例还是多例？

Spring 默认是单例

前 HTTP session 内有效

IOC 容器创建对象

创建对象，有几种方式：

1） 调用无参数构造器

2） 带参数构造器

3） 工厂创建对象

工厂类，静态方法创建对象

工厂类，非静态方法创建对象

依赖注入有哪些方式

Spring 中，如何给对象的属性赋值？【DI，依赖注入】

1) 通过构造函数

2) 通过 set 方法给属性注入值

3) p 名称空间

4) 注解

Spring 事物控制

编程式事务控制

自己手动控制事务，就叫做编程式事务控制。

Jdbc 代码：

```
Conn.setAutoCommite(false);  // 设置手动控制事务
```

Hibernate 代码：

```
Session.beginTransaction();     // 开启一个事务
```

【细粒度的事务控制： 可以对指定的方法、指定的方法的某几行添加事务控制】

(比较灵活，但开发起来比较繁琐： 每次都要开启、提交、回滚.)

声明式事务控制

Spring 提供了对事务的管理，这个就叫声明式事务管理。

Spring 声明事物有 xml 方式和注解方式

Spring 提供了对事务控制的实现。用户如果想用 Spring 的声明式事务管理，只需要在配置文件中配置即可； 不想使用时直接移除配置。这个实现了对事务控制的最大程度的解耦。

Spring 声明式事务管理，核心实现就是基于 Aop。

【粗粒度的事务控制： 只能给整个方法应用事务，不可以对方法的某几行应用事务。】(因为 aop 拦截的是方法。)

Spring 声明式事务管理器类：

Jdbc 技术：**DataSourceTransactionManager**

Hibernate 技术：**HibernateTransactionManager**

Spring 事物传播行为

Spring 中事务的定义：

Propagation（key 属性确定代理应该给哪个方法增加事务行为。这样的属性最重要的部份是传播行为。）有以下选项可供使用：

- PROPAGATION_REQUIRED--支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。
- PROPAGATION_SUPPORTS--支持当前事务，如果当前没有事务，就以非事务方式执行。
- PROPAGATION_MANDATORY--支持当前事务，如果当前没有事务，就抛出异常。
- PROPAGATION_REQUIRES_NEW--新建事务，如果当前存在事务，把当前事务挂起。
- PROPAGATION_NOT_SUPPORTED--以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
- PROPAGATION_NEVER--以非事务方式执行，如果当前存在事务，则抛出异常。

Mybatis 与 Hibernate 区别

Mybatis 是轻量级封装,Hibernate 是重量级封装

Mybatis 以 SQL 语句得到对象,hibernate 是以对象得到 SQL 语句

Mybatis#与$区别

优先使用 #{}。因为 ${} 会导致 sql 注入的问题

当实体类中的属性名和表中的字段名不一样 ，怎么办？

第一种， 通过在查询的 SQL 语句中定义字段名的别名，让字段名的别名和实体类的属性名一致。代码如下：

```
<select id="selectOrder" parameterType="Integer" resultType="Order">
   SELECT order_id AS id, order_no AS orderno, order_price AS price
   FROM orders
   WHERE order_id = #{id}
```

```
</select>
```

第二种，是第一种的特殊情况。大多数场景下，数据库字段名和实体类中的属性名差，主要是前者为下划线风格，后者为驼峰风格。在这种情况下，可以直接配置如下，实现自动的下划线转驼峰的功能。

```
<setting name="logImpl" value="LOG4J"/>
    <setting name="mapUnderscoreToCamelCase" value="true" />
</settings>
```

第三种，通过 <resultMap> 来映射字段名和实体类属性名的一一对应的关系。代码如下：

```
<resultMap type="me.gacl.domain.Order" id="OrderResultMap">
    <!-- 用 id 属性来映射主键字段 -->
    <id property="id" column="order_id">
    <!-- 用 result 属性来映射非主键字段，property 为实体类属性名，column 为数据表中的属性 -->
    <result property="orderNo" column ="order_no" />
    <result property="price" column="order_price" />
</resultMap>

<select id="getOrder" parameterType="Integer" resultMap="OrderResultMap">
    SELECT *
    FROM orders
    WHERE order_id = #{id}
</select>
```

XML 映射文件中，除了常见的 select | insert | update | delete 标 签之外，还有哪些标签？

<cache /> 标签，给定命名空间的缓存配置。

<cache-ref /> 标签，其他命名空间缓存配置的引用。

<resultMap /> 标签，是最复杂也是最强大的元素，用来描述如何从数据库结果集中来加载对象。

<parameterMap /> 标签，已废弃！老式风格的参数映射。内联参数是首选，这个元素可能在将来被移除，这里不会记录。

<sql /> 标签，可被其他语句引用的可重用语句块。

<include /> 标签，引用 <sql /> 标签的语句。

<selectKey /> 标签，不支持自增的主键生成策略标签。

<if />

<choose />、<when />、<otherwise />

<trim />、<where />、<set />

<foreach />

<bind />

Mybatis 动态 SQL 是做什么的？都有哪些动态 SQL ？能简述一下动态 SQL 的执行原理吗？

Mybatis 动态 SQL ，可以让我们在 XML 映射文件内，以 XML 标签的形式编写动态 SQL ，完成逻辑判断和动态拼接 SQL 的功能。

Mybatis 提供了 9 种动态 SQL 标签：<if />、<choose />、<when />、<otherwise />、<trim />、<where />、<set />、<foreach />、<bind /> 。

其执行原理为 ,使用 OGNL 的表达式 ,从 SQL 参数对象中计算表达式的值，根据表达式的值动态拼接 SQL ，以此来完成动态 SQL 的功能。

Mapper 接口绑定有几种实现方式,分别是怎么实现的?

接口绑定有三种实现方式：

第一种，通过 XML Mapper 里面写 SQL 来绑定。在这种情况下，要指定 XML 映射文件里面的 "namespace" 必须为接口的全路径名。

第二种，通过注解绑定，就是在接口的方法上面加上 @Select、@Update、@Insert、@Delete 注解，里面包含 SQL 语句来绑定。

第三种，是第二种的特例，也是通过注解绑定，在接口的方法上面加上 @SelectProvider、@UpdateProvider、@InsertProvider、@DeleteProvider 注解，通过 Java 代码，生成对应的动态 SQL 。

Mybatis 的 XML Mapper 文件中 ,不同的 XML 映射文件，id 是否可以重复？

不同的 XML Mapper 文件，如果配置了 "namespace" ，那么 id 可以重复；如果没有配置 "namespace" ，那么 id 不能重复。毕竟"namespace" 不是必须的，只是最佳实践而已。

原因就是 ,namespace + id 是作为 Map<String, MappedStatement> 的 key 使用的。如果没有 "namespace"，就剩下 id ，那么 id 重复会导致数据互相覆盖。如果有了 "namespace" ,自然 id 就可以重复 ,"namespace"不同 ,namespace + id 自然也就不同。

JDBC 编程有哪些不足之处，MyBatis 是如何解决这些问题的？

问题一：SQL 语句写在代码中造成代码不易维护，且代码会比较混乱。

解决方式：将 SQL 语句配置在 Mapper XML 文件中，与 Java 代码分离。

问题二：根据参数不同，拼接不同的 SQL 语句非常麻烦。例如 SQL 语句的 WHERE 条件不一定，可能多也可能少，占位符需要和参数一一对应。

解决方式：MyBatis 提供 <where />、<if /> 等等动态语句所需要的标签，并支持 OGNL 表达式，简化了动态 SQL 拼接的代码，提升了开发效率。

问题三，对结果集解析麻烦，SQL 变化可能导致解析代码变化，且解析前需要遍历。

解决方式：Mybatis 自动将 SQL 执行结果映射成 Java 对象。

问题四，数据库链接创建、释放频繁造成系统资源浪费从而影响系统性能，如果使用数据库链接池可解决此问题。

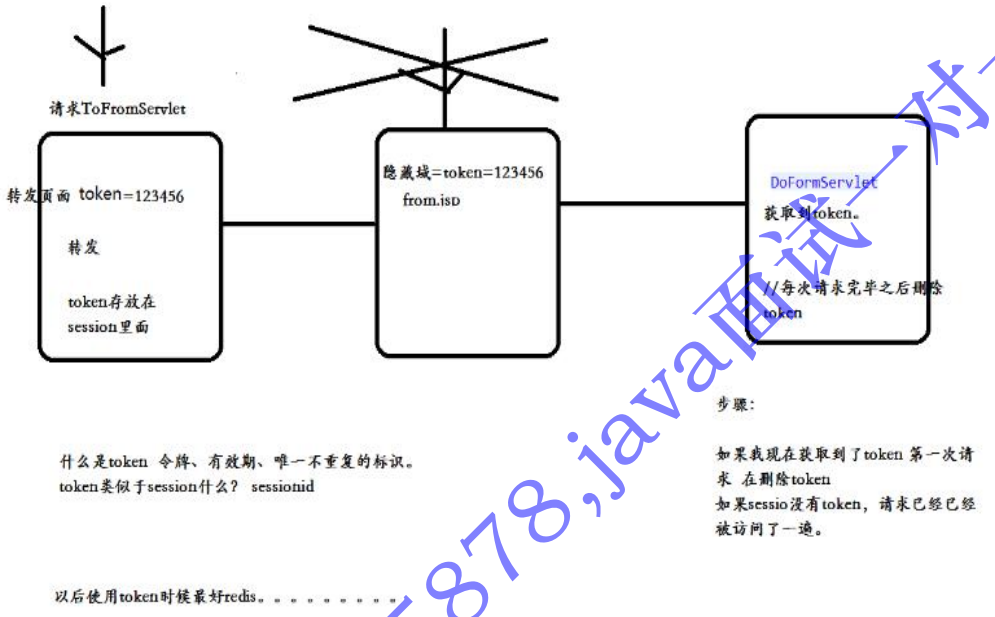
解决方式：在 mybatis-config.xml 中，配置数据链接池，使用连接池管理数据库链接。

安全与防御部分

表单重复提交解决方案(防止 Http 重复提交。)

产生原因

网络延时、重新刷新、点击浏览器的【后退】按钮回退到表单页面后进行再次



提交

前端

点击铓之后，按变灰

标识

2．后瑞

使用 token

1，使用令牌好处唯一性只能有一次请求

web 安全知识点使用令解决模拟请求

我现在把这个生 token 方式知道，还是能被模拟？

模拟程序使用验证码

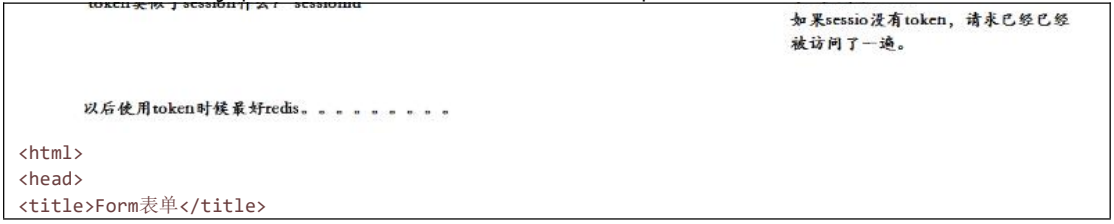
模拟请求识别验证码是非常难

token + 验证码

使用 javascript 解决

既然存在上述所说的表单重复提交问题，那么我们就要想办法解决，比较常用的方法是采用 JavaScript 来防止表单重复提交，具体做法如下：

修改 form.jsp 页面，添加如下的 JavaScript 代码来防止表单重复提交




```
<html>
<head>
<title>Form表单</title>
```

```
<script type="text/javascript">
     var isFlag = false; //表单是否已经提交标识，默认为false

     function submitFlag() {

          if (isFlag == false) {
               isFlag = true;
               return true;
          } else {
               return false;
          }

     }
</script>
</head>

<body>
     <form action="${pageContext.request.contextPath}/DoFormServlet"
          method="post" onsubmit="return submitFlag()">
          用户名：<input type="text" name="userName"> <input type="submit"
               value="提交" id="submit">
     </form>
</body>
</html>
```

除了用这种方式之外，经常见的另一种方式就是表单提交之后，将提交按钮设置为不可用，让用户没有机会点击第二次提交按钮，代码如下：

```
function dosubmit(){
    //获取表单提交按钮
    var btnSubmit = document.getElementById("submit");
    //将表单提交按钮设置为不可用，这样就可以避免用户再次点击提交按钮
    btnSubmit.disabled= "disabled";
     //返回true让表单可以正常提交
    return true;
}
```

6 使用后端提交解决

对于【场景二】和【场景三】导致表单重复提交的问题，既然客户端无法解决，那么就在服务器端解决，在服务器端解决就需要用到 session 了。

具体的做法：在服务器端生成一个唯一的随机标识号，专业术语称为 Token(令牌)，同时在当前用户的 Session 域中保存这个 Token。然后将 Token 发送到客户端的 Form 表单中，在 Form 表单中使用隐藏域来存储这个 Token，表单提交的时候连同这个 Token 一起提交到服务器端，然后在服务器端判断客户端提交上来的 Token 与服务器端生成的 Token 是否一致，如果不一致，那就是重复提交了，此时服务器端就可以不处理重复提交的表单。如果相同则处理表单提交，处理完后清除当前用户的 Session 域中存储的标识号。

在下列情况下，服务器程序将拒绝处理用户提交的表单请求：

存储 Session 域中的 Token(令牌)与表单提交的 Token(令牌)不同。

当前用户的 Session 中不存在 Token(令牌)。

用户提交的表单数据中没有 Token(令牌)。

```
@WebServlet("/ForwardServlet")
public class ForwardServlet extends HttpServlet {
     @Override
     protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
          req.getSession().setAttribute("sesionToken", TokenUtils.getToken());
          req.getRequestDispatcher("form.jsp").forward(req, resp);
     }
```

```
}
```

转发页面:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Form表单</title>

</head>

<body>
    <form action="${pageContext.request.contextPath}/DoFormServlet"
        method="post" onsubmit="return dosubmit()">
        <input type="hidden" name="token" value="${sesionToken}"> 用户名：<input type="text"
            name="userName"> <input type="submit" value="提交" id="submit">
    </form>
</body>
</html>
```

后端Java代码:

```
@WebServlet("/DoFormServlet")
public class DoFormServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        req.setCharacterEncoding("UTF-8");
        boolean flag = isFlag(req);
        if (!flag) {
            resp.getWriter().write("已经提交...");
            System.out.println("数据已经提交了...");
            return;
        }
        String userName = req.getParameter("userName");
        try {
            Thread.sleep(300);
        } catch (Exception e) {
            // TODO: handle exception
        }
        System.out.println("往数据库插入数据...." + userName);
        resp.getWriter().write("success");
    }

    public boolean isFlag(HttpServletRequest request) {
        HttpSession session = request.getSession();
        String sesionToken = (String) session.getAttribute("sesionToken");
        String token = request.getParameter("token");
        if (!(token.equals(sesionToken))) {
            return false;
        }
        session.removeAttribute("sesionToken");
        return true;
    }
}
```


如何防御 XSS 攻击

XSS 攻击使用 Javascript 脚本注入进行攻击

解决办法:就是将请求可能会发送的特殊字符、javascript 标签转换为 html 代码执行

例如在表单中注入:<script>location.href='http://www.itmayiedu.com'</script>

注意:谷歌浏览器 已经防止了 XSS 攻击，为了演示效果，最好使用火狐浏览器

代码: fromToXss.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>
</head>
<body>
    <form action="XssDemo" method="post">
        <input type="text" name="userName"> <input type="submit">
    </form>
</body>
</html>
```

代码: XssDemo

```
import java.io.IOException;
@WebServlet("/XssDemo")
public class XssDemo extends HttpServlet {

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
IOException {
        String userName = req.getParameter("userName");
        req.setAttribute("userName", userName);
        req.getRequestDispatcher("showUserName.jsp").forward(req, resp);
    }


}
```

代码: showUserName.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Insert title here</title>

</head>
<body>userName:${userName}

</body>
</html>
```

7.1 解决方案

使用 Fileter 过滤器过滤器注入标签

FilterDemo

```
import java.io.IOException;
.......
public class FilterDemo implements Filter {
    public FilterDemo() {
        System.out.println("FilterDemo 构造函数被执行...");
    }

    /**
     * 销毁
```

```
     */
    public void destroy() {
        System.out.println("destroy");
    }

    public void doFilter(ServletRequest paramServletRequest, ServletResponse paramServletResponse,
            FilterChain paramFilterChain) throws IOException, ServletException {
        System.out.println("doFilter");
        HttpServletRequest request = (HttpServletRequest) paramServletRequest;
        XssAndSqlHttpServletRequestWrapper xssRequestWrapper = new
XssAndSqlHttpServletRequestWrapper(request);
        paramFilterChain.doFilter(xssRequestWrapper, paramServletResponse);

    }
    /**
     * 初始化
     */
    public void init(FilterConfig paramFilterConfig) throws ServletException {
        System.out.println("init");
    }
}
```

XssAndSqlHttpServletRequestWrapper

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletRequestWrapper;
import org.apache.commons.lang3.StringEscapeUtils;
import org.apache.commons.lang3.StringUtils;

/**
 * 防止XSS攻击
 */
public class XssAndSqlHttpServletRequestWrapper extends HttpServletRequestWrapper {
    HttpServletRequest request;
    public XssAndSqlHttpServletRequestWrapper(HttpServletRequest request) {
        super(request);
        this.request = request;
    }
    @Override
    public String getParameter(String name) {
        String value = request.getParameter(name);
        System.out.println("name:" + name + "," + value);
        if (!StringUtils.isEmpty(value)) {
            // 转换Html
            value = StringEscapeUtils.escapeHtml4(value);
        }
        return value;
    }
}
```


跨域实战解决方案

跨域原因产生：在当前域名请求网站中，默认不允许通过 ajax 请求发送其他域名。（主机域名和端口号有一个不同）

1. jsonp 支持 get 请求，不支持 post 请求。
2. httpClient 内部转发
3. 添加 header 头信息，允许访问（最简单）
4. 使用 nginx 搭建企业 api 接口网关
5. 使用 springzull 接口网关

XMLHttpRequest cannot load 跨域问题解决办法

6. 使用后台 response 添加 header

response.setHeader("Access-Control-Allow-Origin", "*")
Spring4.2 后注解@CrosOrigin(origins="")

使用接口网关

使用 nginx 转发。

配置:

```
server {
       listen       80;
       server_name  www.itmayiedu.com;
       location /A {
             proxy_pass  http://a.a.com:81/A;
              index  index.html index.htm;
       }
          location /B {
             proxy_pass  http://b.b.com:81/B;
              index  index.html index.htm;
       }
    }
```

什么是 SQL 语句注入

Sql 语句如果是通过拼接方式执行的话,传入参数 or 1=1 会发生语句成立，导致数据错误。

应该使用 PreparedStatement 先编译 在执行 通过？号穿参数的方式进行执行。

什么是 NOSQL?

NoSQL 是 Not Only SQL 的缩写，意即"不仅仅是 SQL"的意思，泛指非关系型的数据库。强调 Key-Value Stores 和文档数据库的优点，而不是单纯的反对 RDBMS。

NoSQL 产品是传统关系型数据库的功能阉割版本 ,通过减少用不到或很少用的功能，来大幅度提高产品性能

NoSQL 产品 redis、mongodb Membase、HBase

什么是 Redis?

Redis 是完全开源免费的，遵守 BSD 协议，是一个高性能的 key-value 数据库。

Redis 与其他 key - value 缓存产品有以下三个特点：

Redis 支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。

Redis 不仅仅支持简单的 key-value 类型的数据，同时还提供 list，set，zset，hash 等数据结构的存储。

Redis 支持数据的备份，即 master-slave 模式的数据备份。

Redis 应用场景

主要能够体现 解决数据库的访问压力。

例如:短信验证码时间有效期、session 共享解决方案

Redis 优势

性能极高 – Redis 能读的速度是 110000 次/s,写的速度是 81000 次/s 。

丰富的数据类型 – Redis 支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。

原子 – Redis 的所有操作都是原子性的，同时 Redis 还支持对几个操作全并后的原子性执行。

丰富的特性 – Redis 还支持 publish/subscribe, 通知, key 过期等等特性。

Redis 的基本数据类型

字符串类型(String)

```
redis 127.0.0.1:6379> SET mykey "redis"
OK
redis 127.0.0.1:6379> GET mykey
"redis"
```

在上面的例子中，SET 和 GET 是 redis 中的命令，而 mykey 是键的名称。

Redis 字符串命令用于管理 Redis 中的字符串值。以下是使用 Redis 字符串命令的语法。

redis 127.0.0.1:6379> COMMAND KEY_NAME

Shell

示例

redis 127.0.0.1:6379> SET mykey "redis"

OK

redis 127.0.0.1:6379> GET mykey

"redis"

Shell

在上面的例子中，SET 和 GET 是 redis 中的命令，而 mykey 是键的名称。

Redis 字符串命令

列表类型(List)

Redis 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）

一个列表最多可以包含 232 - 1 个元素 (4294967295, 每个列表超过 40 亿个元素)。

redis 127.0.0.1:6379> LPUSH runoobkey redis

(integer) 1

redis 127.0.0.1:6379> LPUSH runoobkey mongodb

(integer) 2

redis 127.0.0.1:6379> LPUSH runoobkey mysql

(integer) 3

redis 127.0.0.1:6379> LRANGE runoobkey 0 10

1) "mysql"
2) "mongodb"
3) "redis"

Redis 列表命令

下表列出了列表相关的基本命令：

号	命令及描述
	BLPOP key1 [key2] timeout 移出并获取列表的第一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。
	BRPOP key1 [key2] timeout 移出并获取列表的最后一个元素，如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。
	BRPOPLPUSH source destination timeout 从列表中弹出一个值，将弹出的元素插入到另外一个列表中并返回它； 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。
	LINDEX key index 通过索引获取列表中的元素
	LINSERT key BEFORE|AFTER pivot value 在列表的元素前或者后插入元素
	LLEN key 获取列表长度
	LPOP key 移出并获取列表的第一个元素
	LPUSH key value1 [value2] 将一个或多个值插入到列表头部
	LPUSHX key value 将一个值插入到已存在的列表头部
0	LRANGE key start stop 获取列表指定范围内的元素
1	LREM key count value 移除列表元素
2	LSET key index value 通过索引设置列表元素的值
	LTRIM key start stop

3	对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的将被删除。
4	RPOP key 移除并获取列表最后一个元素
5	RPOPLPUSH source destination 移除列表的最后一个元素，并将该元素添加到另一个列表并返回
6	RPUSH key value1 [value2] 在列表中添加一个或多个值
7	RPUSHX key value 为已存在的列表添加值

集合(Set)

Redis 的 Set 是 string 类型的无序集合。集合成员是唯一的，这就意味着集合中不能出现重复的数据。

Redis 中 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 O(1)。

集合中最大的成员数为 232 - 1 (4294967295, 每个集合可存储 40 多亿个成员)。

实例

```
redis 127.0.0.1:6379> SADD runoobkey redis
(integer) 1
redis 127.0.0.1:6379> SADD runoobkey mongodb
(integer) 1
redis 127.0.0.1:6379> SADD runoobkey mysql
(integer) 1
redis 127.0.0.1:6379> SADD runoobkey mysql
(integer) 0
redis 127.0.0.1:6379> SMEMBERS runoobkey

1) "mysql"
2) "mongodb"
3) "redis"
```

什么是 redis 的主从复制

1、redis 的复制功能是支持多个数据库之间的数据同步。一类是主数据库（master）一类是从数据库（slave），主数据库可以进行读写操作，当发生写操作的时候自动将数据同步到从数据库，而从数据库一般是只读的，并接收主数据库同步过来的数据，一个主数据库可以有多个从数据库，而一个从数据库只能有一个主数据库。

2、通过 redis 的复制功能可以很好的实现数据库的读写分离，提高服务器的负载能力。主数据库主要进行写操作，而从数据库负责读操作。

过程：

1：当一个从数据库启动时，会向主数据库发送 sync 命令，

2：主数据库接收到 sync 命令后会开始在后台保存快照（执行 rdb 操作），并将保存期间接收到的命令缓存起来

3：当快照完成后，redis 会将快照文件和所有缓存的命令发送给从数据库。

4：从数据库收到后，会载入快照文件并执行收到的缓存的命令。

修改 redis.conf

修改从 redis 中的 redis.conf 文件

slaveof 192.168.33.130 6379

masterauth 123456--- 主 redis 服务器配置了密码,则需要配置

什么是哨兵机制

Redis 的哨兵(sentinel) 系统用于管理多个 Redis 服务器,该系统执行以下三个任务:

· **监控(Monitoring)**: 哨兵(sentinel) 会不断地检查你的 Master 和 Slave 是否运作正常。

· **提醒(Notification)**:当被监控的某个 Redis 出现问题时, 哨兵(sentinel) 可以通过 API 向管理员或者其他应用程序发送通知。

· **自动故障迁移(Automatic failover)**:当一个 Master 不能正常工作时，哨兵(sentinel) 会开始一次自动故障迁移操作,它会将失效 Master 的其中一个 Slave 升级为新的 Master, 并让失效 Master 的其他 Slave 改为复制新的 Master; 当客户端试图连接失效的 Master 时,集群也会向客户端返回新 Master 的地址,使得集群可以使用 Master 代替失效 Master。

哨兵(sentinel) 是一个分布式系统,你可以在一个架构中运行多个哨兵(sentinel) 进程,这些进程使用流言协议(gossipprotocols)来接收关于 Master 是否下线的信息,并使用投票协议(agreement protocols)来决定是否执行自动故障迁移,以及选择哪个 Slave 作为新的 Master.

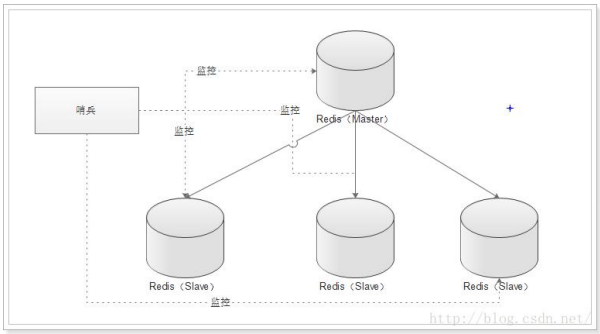
每个哨兵(sentinel) 会向其它哨兵(sentinel)、master、slave 定时发送消息,以确认对方是否”活”着,如果发现对方在指定时间(可配置)内未回应,则暂时认为对方已挂(**所谓的”主观认为宕机” Subjective Down,简称 sdown**).

若”哨兵群”中的多数 sentinel,都报告某一 master 没响应,系统才认为该 master"彻底死亡"(**即:客观上的真正 down 机,Objective Down,简称 odown**),通过一定的 vote 算法,从剩下的 slave 节点中,选一台提升为 master,然后自动修改相关配置.

虽然哨兵(sentinel) 释出为一个单独的可执行文件 redis-sentinel ,但实际上它只是一个运行在特殊模式下的 Redis 服务器，你可以在启动一个普通 Redis 服务器时通过给定 --sentinel 选项来启动哨兵(sentinel).

哨兵(sentinel) 的一些设计思路和 zookeeper 非常类似

单个哨兵(sentinel)



10.2 哨兵模式修改配置

实现步骤:

1.拷贝到 etc 目录

cp sentinel.conf /usr/local/redis/etc

2.修改 sentinel.conf 配置文件

sentinel monitor mymast 192.168.110.133 6379 1 #主节点 名称 IP 端口号 选举次数

3. 修改心跳检测 5000 毫秒

sentinel down-after-milliseconds mymaster 5000

4.sentinel parallel-syncs mymaster 2 --- 做多多少合格节点

5. 启动哨兵模式

./redis-server /usr/local/redis/etc/sentinel.conf --sentinel &

6. 停止哨兵模式

Redis 事物

Redis 事务可以一次执行多个命令， 并且带有以下两个重要的保证：

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

一个事务从开始到执行会经历以下三个阶段：

开始事务。

命令入队。

执行事务。

以下是一个事务的例子， 它先以 MULTI 开始一个事务， 然后将多个命令入队到事务中， 最后由 EXEC 命令触发事务， 一并执行事务中的所有命令：

```
redis 127.0.0.1:6379> MULTI
OK
redis 127.0.0.1:6379> SET book-name "Mastering C++ in 21 days"
QUEUED
redis 127.0.0.1:6379> GET book-name
QUEUED
redis 127.0.0.1:6379> SADD tag "C++" "Programming" "Mastering Series"
QUEUED
redis 127.0.0.1:6379> SMEMBERS tag
QUEUED
redis 127.0.0.1:6379> EXEC
1) OK
2) "Mastering C++ in 21 days"
3) (integer) 3
4) 1) "Mastering Series"
   2) "C++"
   3) "Programming"
```

Redis 持久化

什么是 Redis 持久化,就是将内存数据保存到硬盘。
Redis 持久化存储 (AOF 与 RDB 两种模式)

RDB 默认开启，redis.conf 中的具体配置参数如下；

```
#dbfilename：持久化数据存储在本地的文件
dbfilename dump.rdb
#dir：持久化数据存储在本地的路径，如果是在/redis/redis-3.0.6/src下启动的redis-cli，则数据会存储在当前src目录下
dir ./
##snapshot触发的时机，save
##如下为900秒后，至少有一个变更操作，才会snapshot
##对于此值的设置，需要谨慎，评估系统的变更操作密集程度
##可以通过“save “””来关闭snapshot功能
#save时间，以下分别表示更改了1个key时间隔900s进行持久化存储；更改了10个key300s进行存储；更改10000个key60s进行存储。
save 900 1
save 300 10
save 60 10000
##当snapshot时出现错误无法继续时，是否阻塞客户端“变更操作”，“错误”可能因为磁盘已满/磁盘故障/OS级别异常等
stop-writes-on-bgsave-error yes
##是否启用rdb文件压缩，默认为“yes”，压缩往往意味着“额外的cpu消耗”，同时也意味这较小的文件尺寸以及较短的网络传输时间
rdbcompression yes
```

AOF 持久化

```
##此选项为aof功能的开关，默认为“no”，可以通过“yes”来开启aof功能
##只有在“yes”下，aof重写/文件同步等特性才会生效
appendonly yes

##指定aof文件名称
appendfilename appendonly.aof

##指定aof操作中文件同步策略，有三个合法值：always everysec no,默认为everysec
appendfsync everysec
##在aof-rewrite期间，appendfsync是否暂缓文件同步，"no"表示“不暂缓”，“yes”表示“暂缓”，默认为“no”
no-appendfsync-on-rewrite no

##aof文件rewrite触发的最小文件尺寸(mb,gb),只有大于此aof文件大于此尺寸是才会触发rewrite，默认“64mb”，建议“512mb”
auto-aof-rewrite-min-size 64mb

##相对于“上一次”rewrite，本次rewrite触发时aof文件应该增长的百分比。
##每一次rewrite之后，redis都会记录下此时“新aof”文件的大小(例如A)，那么当aof文件增长到A*(1 + p)之后
##触发下一次rewrite，每一次aof记录的添加，都会检测当前aof文件的尺寸。
auto-aof-rewrite-percentage 100
```

AOF 与 RDB 区别

Rdb 存储：二进制存储，不是实时，存储 10 个值以上，开始持久化机制，优点体积小
Aof 存储：实时存储，日志文件存储，文件大。因为实时存储，所以经常用于灾难备份。
Redis 宕机后，redis 值会失效吗？

不会，默认开启 Rdb 存储。注：Rdb 存储方式以 key 值达到一定次数才做持久化机制，Rdb 断开会自动备份。但是 kill 进程，断电除外。最好采用 Aof 进行持久化，因为是实时的

Redis 发布订阅

Redis 如何做集群？

什么是 nginx

nginx 是一款高性能的 http 服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器。官方测试 nginx 能够支支撑 5 万并发链接，并且 cpu、内存等资源消耗却非常低，运行非常稳定，所以现在很多知名的公司都在使用 nginx。

Nginx :可以根据客户端的 ip 进行负载均衡，在 upstream 里设置 ip_path，负载均衡五种配置：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；2）指定权重，指定轮询几率。权重越大，轮询几率越大，用于后端服务器性能不均的情况。3）ip 绑定 ip_path，每个请求按访问 ip 的哈希结果分配，这样每个客户固定访问一个服务器，可以解决 session 问题。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。5）url_hash 按访问的 url 结果来分配请求，使每个 url 定位到同一个后端服务器。后端服务器为缓存时比较有效。

什么是 Nginx

Nginx 是一个高级的轻量级的 web 服务器,由俄罗斯科学家开发的 ,具有如下优点:

1.占用内存少，并发量强，支持多种并发连接，效率高.

2.能够作为负载均衡服务器和(内部直接支持 Rails 和 PHP)代理服务器。Nginx 用 C 编写开销和 CPU 占有小.

3.安装启动简单，配置简洁，bug 少，一般几个月不需要重新启动且不会宕机，稳定性和安全性好.

Nginx 的作用

反向代理、负载均衡、配置主备 tomcat、动静分离

Nginx 应用场景

做 HTTP 服务器、反向代理服务器、静态资源服务器

什么是反向代理

代替真实服务器接收网络请求，然后将请求转发到真实服务器

反向代理的作用

隐藏真实服务器，使真实服务器只能通过内网访问，保护了真实服务器不被攻击。配置负载均衡，减轻单台真实服务器的压力。配置主备服务器，保持服务稳定运行。

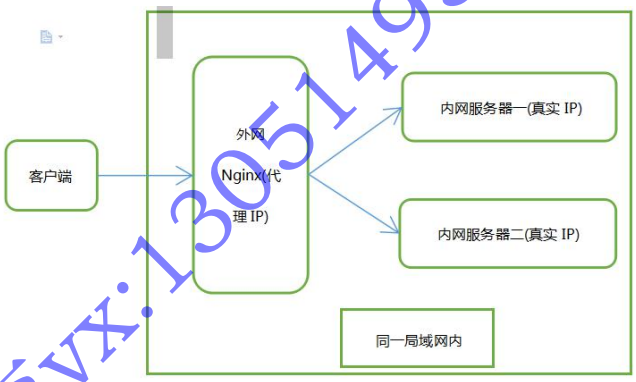
Nginx 如何配置反向代理

首先到 DNS 服务器做域名解析，如果是局域网在 hosts 文件中配置 IP 和域名对应关系。编辑 nginx 的 nginx.conf 文件，配置 server_name 为指向 nginx 服务器的域名，location 拦截请求，如果是访问 nginx 本地资源则配置 root，如果是反向代理到真实服务器则配置 proxy_pass 为服务器地址

说说常用 Nginx 的相关配置

upstream　负载均衡配置
server [IP] [weight] [backup]配置 tomcat 集群
proxy_connect_timeout、proxy_read_timeout、proxy_send_timeout　连接时间、真实服务器响应时间、返回结果时间
location匹配用户请求的 url
root　配置本地资源路径
proxy_pass　配置真实服务器地址

请画图展示反向代理流程



LVS 与 Nginx 区别

LVS 是四层反向代理，基于 TCP 和 UDP 协议，可用于管理 Nginx 集群，抗负载能力强。Nginx 是七层反向代理，基于 HTTP 协议，用于管理真实服务器集群。

location 的作用

匹配用户请求 url，根据不同请求转发到不同的服务器。

Nginx 中如何配置负载均衡

在 upstream 中配置多个 server，在 location 的 proxy_pass 配置为

http://+upstream 名称

四层负载均衡与七层负载均衡区别

四层负载均衡基于 TCP 和 UDP 协议，通过 IP+端口号接受请求并转发到服务器。七层负载均衡基于 HTTP 协议，通过 url 或主机名接收请求并转发到服务器。

四层负载均衡有那些实现方案

LVS、F5

负载均衡有那些算法

轮询算法:按照时间顺序分配到不同的服务器 ,当其中一台服务器宕机则被自动剔除，切换到正常的服务器。

权重算法:按照分配给服务器的权重比例来分发到不同服务器，权重比例越高，则访问几率越大。

IP 绑定(ip_hash):根据访问的 IP 的哈希结果来判定，使同一个 IP 访问一台固定的后端服务器，同时解决动态页面的 session 问题.

服务器集群后，会产生了那些问题

分布式锁
分布式全局 ID
分布式 Session 一致性问题
分布式事务
分布式任务调度
分布式日志收集
分布式配置中心

什么是动态负载均衡

一般情况下，使用 nginx 搭建服务器集群，每次修改 nginx.conf 配置文件都需要重启 nginx 服务器。动态负载均衡就是修改 nginx.conf 配置文件后不必重启 nginx 而使配置生效。

Nginx 如何实现动态负载均衡

搭建 Nginx+Consul+Upsycn 环境。Nginx 实现服务的反向代理和负载均衡。Consul 是一个开源的注册中心和服务发现的框架，通过 HTTP API 来发现服务，注册服务。同时支持故障发现，K/V 存储，多数据中心，Raft 算法等多中高可用特性。Consul 在 Nginx 动态负载均衡作用是

通过 Http api 注册和发现服务.Upsycn 是新浪微博的开源框架，在 Nginx 动态负载均衡的作用是 Consul 的后端的 server 列表，即获取 Nginx 的上游服务器(Upstream server ）信息,并动态更新 Nginx 的路由信息.

什么是 Http 协议

超文本传输协议

Http 协议组成部分

Http 协议是基于 TCP 协议封装成超文本传输协议，包括请求(request）和响应(response),http 协议请求（request）分为请求参数（request params）和方法类型(request method）、请求头（request hearder）、请求体(request body) ，

响应(response)分为 响应状态(response state)、响应头(response header)、响应体(response body）等.

如何实现双机主从热备

Nginx+Tomcat：在 upstream 中配置多台服务器，从服务器后加 backup

Keepalived+Nginx：在多台 nginx 服务器上安装 keepalived，将主服务器的 state 设置为 MASTER，从服务器设置为 BACKUP，主服务器的优先级要高于从服务器

项目发布如何不影响到正常用户访问，实现 7*24 小时访问

可以两台机子互为热备，平时各自负责各自的服务。在做上线更新的时候，关闭一台服务器的 tomcat 后，nginx 自动把流量切换到另外一台服务的后备机子上，从而实现无痛更新，保持服务的持续性，提高服务的可靠性，从而保证服务器 7*24 小时运行。

项目如何发生故障宕机了，如何处理。

使用 lvs+keepalived+Nginx 做主从热备，lvs 管理 nginx 集群，nginx 管理服务器集群，在服务器宕机的情况下 keepalived 启动健康检测，多次重启无果可以短信通知运维人员及时维护。

nginx 应用场景

1、http 服务器。Nginx 是一个 http 服务可以独立提供 http 服务。可以做网页静态服务器。

2、虚拟主机。可以实现在一台服务器虚拟出多个网站。例如个人网站使用的虚拟主机。

3、反向代理，负载均衡。当网站的访问量达到一定程度后，单台服务器不能满足用户的请求时，需要用多台服务器集群可以使用 nginx 做反向代理。并且多台服务器可以平均分担负载，不会因为某台服务器负载高宕机而某台服务器闲置的情况。

Nginx 与 Ribbon 的区别

Nginx 是反向代理同时可以实现负载均衡，nginx 拦截客户端请求采用负载均衡策略根据 upstream 配置进行转发，相当于请求通过 nginx 服务器进行转发。Ribbon 是客户端负载均衡，从注册中心读取目标服务器信息，然后客户端采用轮询策略对服务直接访问，全程在客户端操作。

Nginx(类似 nginx 反向代理、lvs、 (中国人) F5(硬件)、Haproxy)

1.http 服务器(反向代理服务器，不暴露真实 IP 地址)、
2.虚拟主机
3. 负裁均衡、
4.集群、动静分离

优点：nginx 非常好，性能非常好，占内存小、轻量级服务器、抗住同时 5 万并发连接。分布式、微服务、高并发、高可用、消息中间、

使用 nginx 实现反向代理、集群、负载均衡的算法或策略（轮询，权重，ip 绑定：只让一个 ip 访问，解决会话共享问题，不常用 ），

conf/nginx.conf:

```
upstream backserver {
        （在此配 ip_hash 表示 ip 绑定）
        server 192.168.0.14 weight:2; //权重 2：1
        server 192.168.0.15 weight:1; //14，15 两个服务器轮询
}
   server{
   Listen:80;
   server_name:www.itmayidu.com
     Location / (/表示拦截所有请求/a 拦截/a 请求){
root: html(配置后 proxy_pass     http://127.0.0.1:8080(要转发的地址));
index index.html( ) }
}
```

解决跨域问题，使用 nginx 搭建企业级 api 接口网关 nginx 解决防 ddos、nginx 安 全
linux 离可用、会话共享、高并发解决方案、动静分离。

80 端口是浏览器默认端口，nigix 的静态资源放在 html 文件下，用于动静分离。服务器集群怎么实现，使用 nginx 实现负载均衡，Windows linux 配置完全相同。负载均衡（默认轮询）：提高网站吞吐量 ，每台 nginx 都需要安装 keepalived

面试官安全架构

nginx 反向代理(不暴露真实 IP 地址)
使用 Https 防止抓包分析 Htp 请求(抓包工具？？？)
搭建企业黑名单和白名单系统(防盗链)模拟请求(csrf) 、XSS、sql 注入、
ddos (流量攻 击) nginx 解决 csrf 业务攻击
ddos 流量攻击--让别人无法访问 ICDN

集群（目的：解决但台服务器压力，）会产生什么情况：

1. 分布式 job 幂等性（重复）问题（使用 XXLjob 分布式任务调度平台）
2. 会话共享问题（eg:两个 tomcat 的 session 不一样）
3. 分布式生成全局 id（生成订单 id，时间戳，或者订单号怎么解决幂等性问题，解决：提前生成好存放在 redis 中。）

跨域：

当俩个协议，域名，端口号不一致认为 js 或者 ajax 跨域（eg:a 的应用的 js

访问 b 的后端地址）

解决办法：

①jsonp 但是只支持 get 请求，不支持 post 请求

②httpclient 进行内部转发

③添加 header 头请求头允许访问，最简单

Response.setHeader(“Access-Controller-Allow-Origin”,”http://...”(* 为 全部访问，不安全))

Response.setHeader(“Access-Controller-Allow-Credentials”,”true”.(如 有 cookie 写)

Spring4.2 的注解整合

@crossorigin(origins=”http://...”,allowCredentials=”true”(可不写))

④使用 nginx 搭建企业接口网关(多配几个 localtion/..)

⑤使用 springzull 接口网关

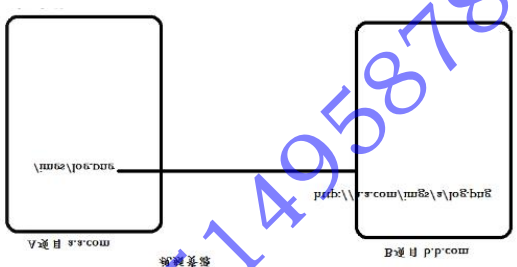
防盗链机制

反盗链机制

1Java 代码控制请求来源资源判断 Referer

2 使用 ngx 反向代理解决防盗链

黑名单和白名单-黑禁止访问白名单运行



1. A 项目中写一个过滤器
2. 过滤器获取请求头的来源字段
3. 判断请求头中来源字段

```
20
21    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
22            throws IOException, ServletException {
23        System.out.println("doFilter...");
24
25        HttpServletRequest req = (HttpServletRequest) request;
26        HttpServletResponse res = (HttpServletResponse) response;
27        String referer = req.getHeader("referer");
28        // 请求服务名称 http://a.a.com
29        String serverName = req.getServerName();
30        System.out.println("referer:" + referer + ",serverName:" + serverName);
31        if (referer == null || !(referer.contains(serverName))) {
32            //
33            req.getRequestDispatcher("imgs/error.png").forward(req, res);
34            return ;
35        }
36
37        //放行
38        chain.doFilter(req, res);
39    }
40
```

Servers | Console | Progress

Tomcat v8.0 Server at localhost [Apache Tomcat] C:\Program Files\Java\jre1.8.0_101\bin\javaw.exe (2017年12月28日 下午11:05:40)

```
doFilter...
referer:http://a.a.com/a/,serverName:a.a.com
doFilter...
referer:http://a.a.com/a/,serverName:a.a.com
doFilter...
referer:http://a.a.com/a/,serverName:a.a.com
```

31% 73.3K/s 7.1K/s

nginx 配置防盗链

```
location ~ .*\.(jpg|jpeg|JPG|png|gif|icon)$ {
       valid_referers blocked http://www.itmayiedu.com www.itmayiedu.com;
       if ($invalid_referer) {
           return 403;
       }
          }
```

Ribbon 底层实现原理

Ribbon 使用 discoveryClient 从注册中心读取目标服务信息 ,对同一接口请求进行计数，使用%取余算法获取目标服务集群索引，返回获取到的目标服务信息。

nginx 优缺点

占内存小，可以实现高并发连接、处理响应快。
可以实现 http 服务器、虚拟主机、反向代理、负载均衡。
nginx 配置简单
可以不暴露真实服务器 IP 地址
nginx.conf 介绍
nginx.conf 文件的结构
nginx 的配置由特定的标识符(指令符)分为多个不同的模块。指令符分为简单指令和块指令。

配置静态访问

Web server 很重要一部分工作就是提供静态页面的访问，例如 images, html page。nginx 可以通过不同的配置，根据 request 请求，从本地的目录提供不同的文件返回给客户端。
打开安装目录下的 nginx.conf 文件，默认配置文件已经在 http 指令块中创建了一个空的 server 块，在 nginx-1.8.0 中的 http 块中已经创建了一个默认的 server 块。内容如下：

```
server {
        listen       80;
        server_name  localhost;
        location / {
            root   html;
            index  index.html index.htm;
        }
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }
}
```

nginx 实现反向代理

反向代理（Reverse Proxy）方式是指以代理服务器来接受 internet 上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给 internet 上请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器。

启动一个 Tomcat 127.0.0.1:8080

使用 nginx 反向代理 8080.itmayiedu.com 直接跳转到 127.0.0.1:8080

Host 文件新增

```
127.0.0.1 8080.itmayiedu.com
127.0.0.1 b8081.itmayiedu.com
```

nginx.conf 配置

```
    server {
        listen       80;
        server_name  8080.itmayiedu.com;
        location / {
              proxy_pass  http://127.0.0.1:8080;
                index  index.html index.htm;
        }
    }
     server {
        listen       80;
        server_name  b8081.itmayiedu.com;
        location / {
              proxy_pass  http://127.0.0.1:8081;
                index  index.html index.htm;
        }
    }
```

nginx 实现负载均衡

什么是负载均衡

负载均衡 建立在现有网络结构之上，它提供了一种廉价有效透明的方法扩展网络设备和服务器的带宽、增加吞吐量、加强网络数据处理能力、提高网络的灵活性和可用性。

负载均衡，英文名称为 Load Balance，其意思就是分摊到多个操作单元上进行执行，例如 Web 服务器、FTP 服务器、企业关键应用服务器和其它关键任务服务器等，从而共同完成工作任务。

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;
}
```

配置代码

```
upstream backserver {
    server 127.0.0.1:8080;
    server 127.0.0.1:8081;
   }
```

```
    server {
        listen       80;
        server_name  www.itmayiedu.com;
        location / {
               proxy_pass  http://backserver;
                 index  index.html index.htm;
        }
    }
```

宕机轮训配置规则

```
    server {
        listen       80;
        server_name  www.itmayiedu.com;
        location / {
               proxy_pass  http://backserver;
                 index  index.html index.htm;
                 proxy_connect_timeout 1;
             proxy_send_timeout 1;
             proxy_read_timeout 1;
        }

          }
```

负载均衡服务器有哪些？

LVS、Ngnix、Tengine（taobao 开发的 Nginx 升级版）、HAProxy（高可用、负载均衡）、Keepalived（故障转移，备机，linux 环境下的组件）

nginx 解决网站跨域问题

配置:

```
server {
        listen       80;
        server_name  www.itmayiedu.com;
        location /A {
               proxy_pass  http://a.a.com:81/A;
                 index  index.html index.htm;
        }
           location /B {
               proxy_pass  http://b.b.com:81/B;
                 index  index.html index.htm;
        }
    }
```

nginx 配置防盗链

```
location ~ .*\.(jpg|jpeg|JPG|png|gif|icon)$ {
       valid_referers blocked http://www.itmayiedu.com www.itmayiedu.com;
       if ($invalid_referer) {
           return 403;
       }
          }
```

设置 Nginx、Nginx Plus 的连接数在一个真实用户请求的合理范围内。比如，你可以设置每个客户端 IP 连接/store 不可以超过 10 个。

集群情况下 Session 共享解决方案

nginx 或者 haproxy 做的负载均衡）

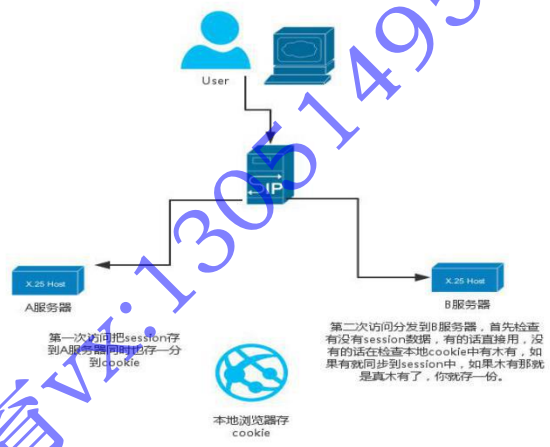
用 Nginx 做的负载均衡可以添加 ip_hash 这个配置，

用 haproxy 做的负载均衡可以用 balance source 这个配置。

从而使同一个 ip 的请求发到同一台服务器。

利用数据库同步 session

利用 cookie 同步 session 数据原理图如下



缺点:安全性差、http 请求都需要带参数增加了带宽消耗

使用 Session 集群存放 Redis

使用 spring-session 框架，底层实现原理是重写 httpsession

引入 maven 依赖

```
    <!--spring boot 与redis应用基本环境配置 -->
      <dependency>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-redis</artifactId>
      </dependency>
      <!--spring session 与redis应用基本环境配置,需要开启redis后才可以使用，不然启动Spring boot会报错
-->
      <dependency>
          <groupId>org.springframework.session</groupId>
```

```
            <artifactId>spring-session-data-redis</artifactId>
        </dependency>
```

创建 SessionConfig

```
import org.springframework.beans.factory.annotation.Value;
。。。。。。。
//这个类用配置redis服务器的连接
//maxInactiveIntervalInSeconds为SpringSession的过期时间（单位：秒）
@EnableRedisHttpSession(maxInactiveIntervalInSeconds = 1800)
public class SessionConfig {

      // 冒号后的值为没有配置文件时，制动装载的默认值
      @Value("${redis.hostname:localhost}")
      String HostName;
      @Value("${redis.port:6379}")
      int Port;

      @Bean
      public JedisConnectionFactory connectionFactory() {
            JedisConnectionFactory connection = new JedisConnectionFactory();
            connection.setPort(Port);
            connection.setHostName(HostName);
            return connection;
      }
}
```

初始化 Session

```
//初始化Session配置
public class SessionInitializer extends AbstractHttpSessionApplicationInitializer{
    public SessionInitializer() {
        super(SessionConfig.class);
    }
}
```

控制器层代码

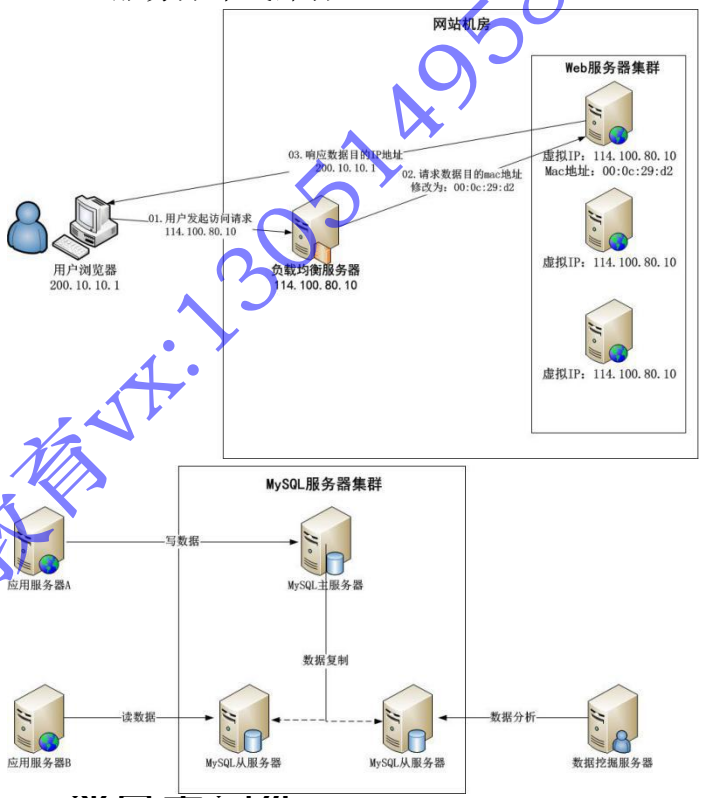
```
import javax.servlet.http.HttpServletRequest;
。。。。。。。
@RestController
public class SessionController {

      @Value("${server.port}")
      private String PORT;

      public static void main(String[] args) {
            SpringApplication.run(SessionController.class, args);
      }

      @RequestMapping("/index")
      public String index() {
            return "index:" + PORT;
      }

      @RequestMapping("/setSession")
      public String setSession(HttpServletRequest request, String sessionKey, String sessionValue) {
            HttpSession session = request.getSession(true);
            session.setAttribute(sessionKey, sessionValue);
            return "success,port:" + PORT;
      }


      @RequestMapping("/getSession")
      public String getSession(HttpServletRequest request, String sessionKey) {
            HttpSession session =null;
            try {
```

```
            session = request.getSession(false);
          } catch (Exception e) {
            e.printStackTrace();
          }
          String value=null;
          if(session!=null){
               value = (String) session.getAttribute(sessionKey);
          }
          return "sessionValue:" + value + ",port:" + PORT;
     }

}
```

高并发解决方案

业务数据库 -》数据水平分割(分区分表分库)、读写分离、SQL 优化
业务应用 -》逻辑代码优化(算法优化)、公共数据缓存
应用服务器 -》反向静态代理、配置优化、负载均衡(apache 分发，多 tomcat 实例)
系统环境 -》JVM 调优
页面优化 -》减少页面连接数、页面尺寸瘦身

1、动态资源和静态资源分离；
2、CDN；
3、负载均衡；
4、分布式缓存；
5、数据库读写分离或数据切分（垂直或水平）；
6、服务分布式部署。



消息中间件

发送者将消息发送给消息服务器，消息服务器将消感存放在若干队列中，在合适的时候再将消息转发给接收者。

这种模式下，发送和接收是异步的，发送者无需等待; 二者的生命周期未必相同: 发送消息的时候接收者不一定运行 ,接收消息的时候
发送者也不一定运行;一对多通信: 对于一个消息可以有多个接收者。

消息中间件 (ActiveMQ,RabbitMQ,RocketMQ,ZeroMQ,Kafka)

JMS（消息中间件）：发送消息，客户端与服务端进行异步通讯方式，与传统通讯方式比，异步无需等待。如果生产发送几万个消息，消息中间件会宕机吗？不会，mq 本身就有解决高并发能力

如何让避免消息丢失

消息模型（消息中间件有哪些通讯方式）

1. 点对点 （生产者，消息队列，消费者）
2. 发布订阅 （生产者，主题（topic），消费者），用的不多，有可能会导致重复消费

保证消息中间件重试机制后，消费者不会被重复消费，怎么保证消息中间件幂等性问题？核心通过全局 id 去解决

MQ 协议：

消息中间件持久化：可以保证 jms 如果宕机情况，可以从硬盘中恢复。

安装 ActiveMQ

```
public class ProducerTest001 {

	public static void main(String[] args) throws JMSException {
		// 获取 mq 连接工程
		ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(ActiveMQConnection.DEFAULT_USER,
				ActiveMQConnection.DEFAULT_PASSWORD, "tcp://127.0.0.1:61616");
		// 创建连接
		Connection createConnection = connectionFactory.createConnection();
		// 启动连接
		createConnection.start();
		// 创建会话工厂
		Session session = createConnection.createSession(Boolean.FALSE, Session.AUTO_ACKNOWLEDGE);
		// 创建队列
		Destination destination = session.createQueue("itmayiedu_queue");
		MessageProducer producer = session.createProducer(destination);
		// 不持久化
		producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
		for (int i = 1; i <= 5; i++) {
			System.out.println("我是生产者: " + i);
			sendMsg(session, producer, "我是生产者: " + i);

		}
		System.out.println("生产者 发送消息完毕!!!");
	}

	public static void sendMsg(Session session, MessageProducer producer, String i) throws JMSException {
		TextMessage textMessage = session.createTextMessage("hello activemq " + i);
		producer.send(textMessage);
	}
}
```

Springboot

```
@Configuration
public class QueueConfig {
	@Value("${queue}")
	private String queue;

	@Bean
	public Queue logQueue() {
		return new ActiveMQQueue(queue);
	}

	@Bean
	public JmsTemplate jmsTemplate(ActiveMQConnectionFactory activeMQConnectionFactory, Queue queue) {
		JmsTemplate jmsTemplate = new JmsTemplate();
		jmsTemplate.setDeliveryMode(2);// 进行持久化配置 1 表示非持久化，2 表示持久化</span>
		jmsTemplate.setConnectionFactory(activeMQConnectionFactory);
		jmsTemplate.setDefaultDestination(queue); // 此处可不设置默认，在发送消息时也可设置队列
		jmsTemplate.setSessionAcknowledgeMode(4);// 客户端签收模式</span>
		return jmsTemplate;
	}

	// 定义一个消息监听器连接工厂，这里定义的是点对点模式的监听器连接工厂
	@Bean(name = "jmsQueueListener")
	public DefaultJmsListenerContainerFactory jmsQueueListenerContainerFactory(
			ActiveMQConnectionFactory activeMQConnectionFactory) {
		DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory();
		factory.setConnectionFactory(activeMQConnectionFactory);
		// 设置连接数
		factory.setConcurrency("1-10");
		// 重连间隔时间
		factory.setRecoveryInterval(1000L);
		factory.setSessionAcknowledgeMode(4);
		return factory;
	}

}
```

```
spring:
  activemq:
    broker-url: tcp://127.0.0.1:61616
    user: admin
    password: admin
queue: springboot2018-02-04-queue
server:
  port: 8080
```

```
@SpringBootApplication
@Component
@EnableScheduling
public class Producer {

	@Autowired
	private JmsMessagingTemplate jmsMessagingTemplate;
	@Autowired
	private Queue queue;

	@Scheduled(fixedDelay = 3000)
	public void send() {
		String reuslt = System.currentTimeMillis() + "---测试消息";
		System.out.println("reuslt:" + reuslt);
		jmsMessagingTemplate.convertAndSend(queue, reuslt);
	}

	public static void main(String[] args) {
		SpringApplication.run(Producer.class, args);
	}

}
```

消费端

```
@SpringBootApplication
@Component
public class Consumer {
        private int count = 0;

        @JmsListener(destination = "${queue}")
        public void receive(TextMessage textMessage,Session session) {
                try {
                        String text = textMessage.getText();

                        System.out.println("消费端接受到生长者消息: " + text + "
第几次获取消息 count:" + (++count));
                        int i = 1 / 0;
                        //伪代码耗时 15 秒
//                      Thread.sleep(15000l);
                        String jmsMessageID = textMessage.getJMSMessageID();
                        //网络延迟情况下，第二次消费过来，应该使用全局 ID 该
消息是否被消费。
//                      if(jmsMessageID==缓存里面){
//                              textMessage.acknowledge();// 避免第三次重试。
//                      }
                        // 消费成功
                        //jmsMessageID==存放在缓存裡面


                } catch (Exception e) {
                        // 在 catch 日志记录消息报文，可以采用补偿机制 （使
用人工补偿），使用定时 JOB 健康检查脏数据。
//                      session.recover();// 重试。
                }
        }

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

使用 MQ 注意事项：

ActiveMQ（这个重试，别的不一定）消费者发生异常，会默认自动重试（eg.接口调用超时）。注意事项：如果消费者代码需要发布解决版本解决问题，不要重试，使用 try,catch,catch 里面记录消息报文，可以采用人工补偿机制，使用定时 job 健康检查脏数据。消费者集群不会产生重复消费，生产者集群要考虑消费者没有及时签收，mq 自动重试机制，造成重复消费，解决消息中间件幂等性问题：使用全局 id 区分消息 textMessage.getMessageID();Mq 幂等性问题：网络发生延迟产生重试机制才会产生幂等性问题。

activeMQ 底层实现原理：

在讨论具体方式的时候，我们先看看使用 activemq 需要启动服务的主要过程。

按照 JMS 的规范，我们首先需要获得一个 JMS connection factory.，通过这个 connection factory 来创建 connection.在这个基础之上我们再创建 session, destination, producer 和 consumer。因此主要的几个步骤如下：

1. 获得 JMS connection factory. 通过我们提供特定环境的连接信息来构造 factory。
2. 利用 factory 构造 JMS connection
3. 启动 connection
4. 通过 connection 创建 JMS session.
5. 指定 JMS destination.
6. 创建 JMS producer 或者创建 JMS message 并提供 destination.
7. 创建 JMS consumer 或注册 JMS message listener.
8. 发送和接收 JMS message.
9. 关闭所有 JMS 资源，包括 connection, session, producer, consumer 等。

Java Message Service：是 Java 平台上有关面向消息中间件的技术规范。

执行流程：

通过生产者配置文件配置 Spring Caching 连接工厂和 JmsTemplate 的类型（Queue，Topic），发布消息时通过调用 jmsTemplate 的 send(“bos_sms”，new MessageCreator())方法(第一个参数与消费者配置文件中的 监听器 一致，new MessageCreator()是一个接口，通过 mapMessage.setString(“randomCode”, randomCode)添加信息)，当消息发布成功之后，消费者会通过消费者配置文件中对应的的监听器（Queue，Topic）监听到消息队列中

有新的消息，则对应的()消费者方法（smsConsumer）执行，该方法需要实现 MessageListener 接口

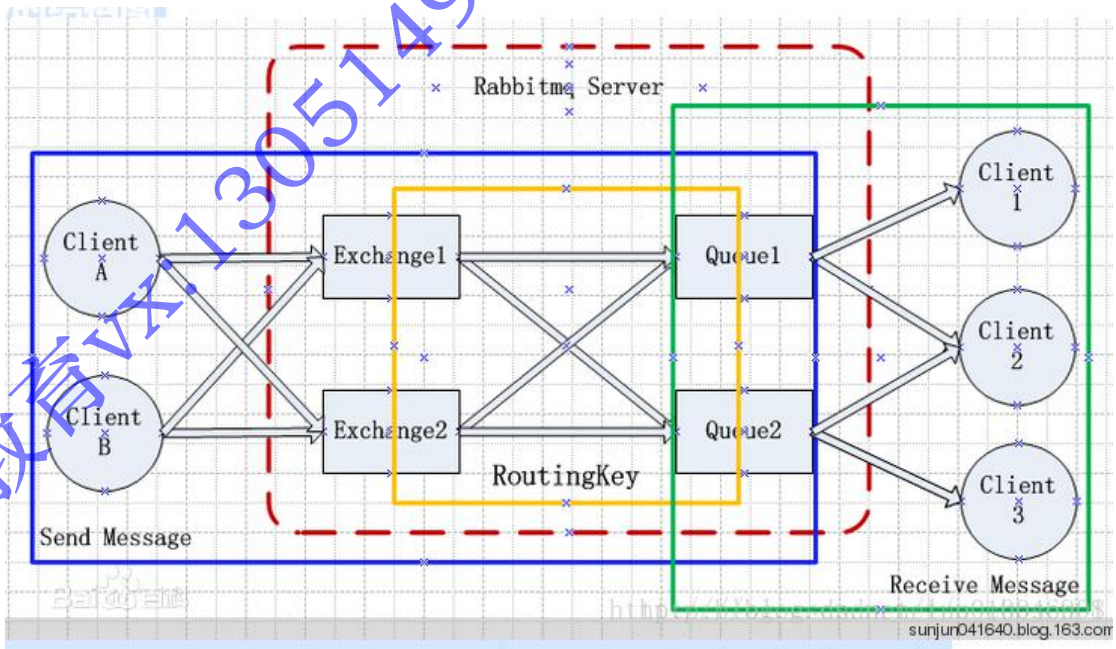
RabbitMQ 是什么

RabbitMQ 是一个遵循 AMQP 协议的消息中间件，它从生产者接受消息并传递给消费者，在这和过程中，根据路由规则就行路由、缓存和持久化。

消息队列中间件的应用场景（不只是 RabbitMQ）

异步处理：在注册服务的时候，如果同步串行化的方式处理，让存储数据、邮件通知等挨着完成，延迟较大，采用消息队列，可以将邮件服务分离开来，将邮件任务之间放入消息队列中，之间返回，减少了延迟，提高了用户体验
应用解耦：电商里面，在订单与库存系统的中间添加一个消息队列服务器，在用户下单后，订单系统将数据先进行持久化处理，然后将消息写入消息队列，直接返回订单创建成功，然后库存系统使用拉/推的方式，获取订单信息再进行库存操作
流量削锋：秒杀活动中，一般会因为流量过大，导致流量暴增，应用挂掉。为解决这个问题，一般需要在应用前端加入消息队列。
服务器在接收到用户请求后，首先写入消息队列。这时如果消息队列中消息数量超过最大数量，则直接拒绝用户请求或返回跳转到错误页面；
秒杀业务根据秒杀规则读取消息队列中的请求信息，进行后续处理
日志处理：Kalfka 消息中间件

RabbitMQ 的结构图



几个概念说明：

Broker：简单来说就是消息队列服务器实体。
Exchange：消息交换机，它指定消息按什么规则，路由到哪个队列。

Queue：消息队列载体，每个消息都会被投入到一个或多个队列。
Binding：绑定，它的作用就是把 exchange 和 queue 按照路由规则绑定起来。
Routing Key：路由关键字，exchange 根据这个关键字进行消息投递。
vhost：虚拟主机，一个 broker 里可以开设多个 vhost，用作不同用户的权限分离。
Producer：消息生产者，就是投递消息的程序。
Consumer：消息消费者，就是接受消息的程序。
Channel：消息通道，在客户端的每个连接里，可建立多个 channel，每个 channel 代表一个会话任务。
消息队列的使用过程

1、客户端连接到消息队列服务器，打开一个 channel。
2、客户端声明一个 exchange，并设置相关属性。
3、客户端声明一个 queue，并设置相关属性。
4、客户端使用 routing key，在 exchange 和 queue 之间建立好绑定关系。
5、客户端投递消息到 exchange。
6、exchange 接收到消息后，就根据消息的 key 和已经设由 binding，进行消息路里，将消息投递到一个或多个队列里

ps:通过 durable 参数来进行 exchang、queue、消息持久化

RabbitMQ 交换机

RabbitMQ 包含四种不同的交换机类型：

Direct exchange：直连交换机，转发消息到 routigKey 指定的队列，如果消息的 routigKey 和 binding 的 routigKey 直接匹配的话，消息将会路由到该队列
Fanout exchange：扇形交换机，转发消息到所有绑定队列（速度最快），不管消息的 routigKey 息和 binding 的参数表头部信息和值是什么，消息将会路由到所有的队列
Topic exchange：主题交换机，按规则转发消息（最灵活），如果消息的 routigKey 和 binding 的 routigKey 符合通配符匹配的话，消息将会路由到该队列
Headers exchange：首部交换机 ，如果消息的头部信息和 binding 的参数表中匹配的话，消息将会路由到该队列。

使用消息中间注意事项

消费者代码不要抛出异常，否则 activqmq 默认有重试机制。
如果代码发生异常，需要发布版本才可以解决的问题，不要使用重试机制，采用日志记录方式，定时 Job 进行补偿。
如果不需要发布版本解决的问题，可以采用重试机制进行补偿。

JMS 可靠消息

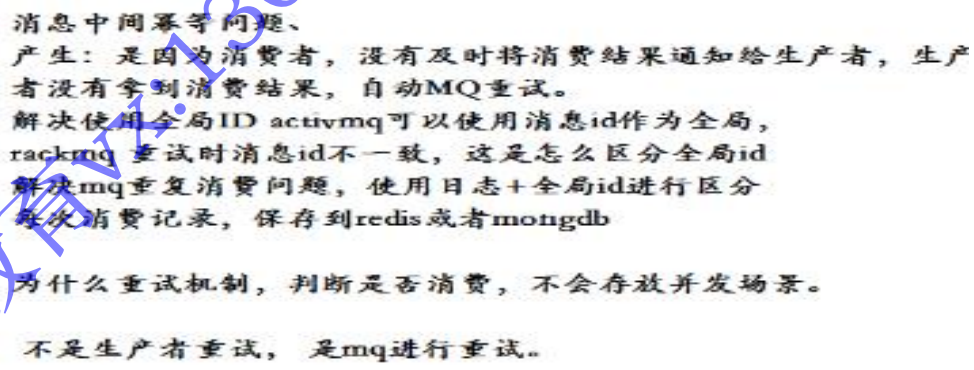
1. 自动签收
2 事物消息
1. 生产者完成发送消息完成后, 必须提交给队列。

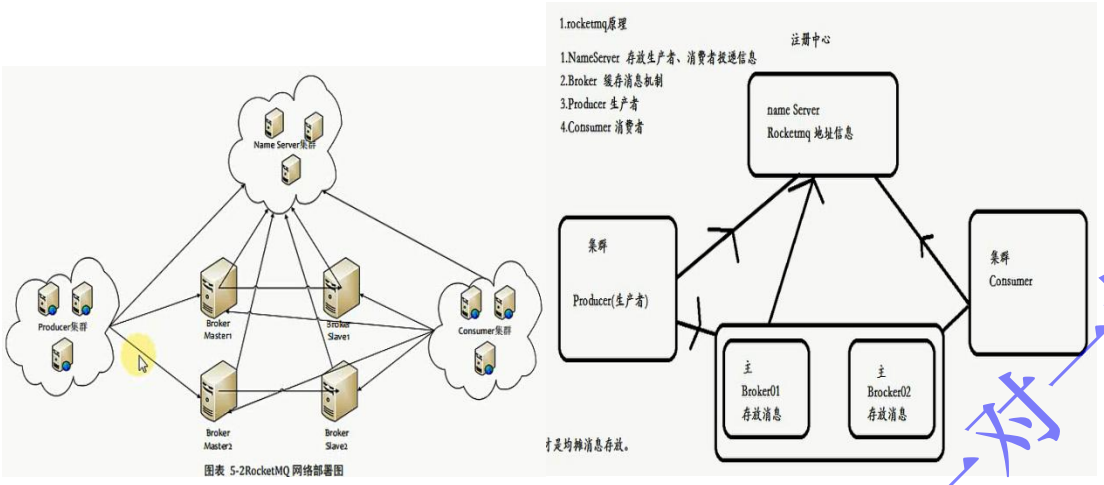
2 消费者,获取事物消息,如采消费没有提交事物,默认表示没有进行消费。一默认自动重试机制。
3. 手动签收
消费者,溪有手动签收消息,默认表示没有进行消费

1 什么是消息中间件?客户端与服务器进行异步通讯。消息中间点对点发布订阅
2 如采生产发送几万个消息？消息中间件会宕机了吗?
不会,mq 本身就有解决高并发能力
3 保试 jms 可靠消息
保证消息中间重试机制后,消费者不会被重复消费。
怎么保证消息中间幂等性!核心通过全 ID 去进行
消息中间件，保证消息幂等性？
消费者，没有及时签收签收情况，MQ 自动重试机制，造成重复消费，解决 MQ 幂等性
使用全局 ID
消息 ID
使用业务逻辑 ID

消费者如果保证消息幂等性，不被重复消费。

产生原因:网络延迟传输中，会造成进行 MQ 重试中，在重试过程中，可能会造成重复消费。
解决办法:
1.使用全局 MessageID 判断消费方使用同一个，解决幂等性。
2.使用 JMS 可靠消息机制（1.自动签收。2.事务消息）
ActiveMQ 点对点通讯，发布订阅，本身不支持集群，不支持分布式，消息堆积没有 RockerMQ 强
RocketMQ 分布式消息中间件，集群，消息效率特别高





图表 5-2RocketMQ 网络部署图

```java
Consumer.java  Producer.java
                                                    蚂蚁课堂 (mayikt.com)
14
15 public class Consumer {
16     static private Map<String, String> LogMap = new HashMap<>();
17
18     public static void main(String[] args) throws MQClientException {
19
20         DefaultMQPushConsumer consumer = new DefaultMQPushConsumer("rmq-group");
21         consumer.setNamesrvAddr("192.168.110.200:9876;192.168.110.197:9876");
22         consumer.setInstanceName("consumer");
23         consumer.subscribe("itmayiedu_topic", "tagA");
24         consumer.registerMessageListener(new MessageListenerConcurrently() {
25
26             @Override
27             public ConsumeConcurrentlyStatus consumeMessage(List<MessageExt> msgs, ConsumeConcurrentlyContext c
28                 String keys =null;
29                 String msgId=null;
30                 for (MessageExt messageExt : msgs) {
31                     try {
32                         keys = messageExt.getKeys();
33                         msgId = messageExt.getMsgId();
34                         String body = new String(messageExt.getBody());
35                         if (LogMap.containsKey(keys)) {
36                             System.out.println("msgId:" + msgId + ",body:" + body + ",已经重复消费了.");
37                             return Cons                                    S;
38                         }
39                         // 正常业务代码
40                         System.out.prin                                    w String(messageExt.getBody()));
41                     } catch (Exception e) {
42                     } finally {
43                         LogMap.put(keys, msgId);
44                     }
45
46                 }
47                 //消费状态 1.消费成功 2.消费失败
48                 return ConsumeConcurrentlyStatus.CONSUME_SUCCESS;
49
50             }
51         });
52         consumer.start();
53         System.out.println("Consumer Started.");
54     }
55 }
```


同步

A 调用 B 接口时,B 接口有廷迟,会产生什么场?A-直午得,等枴 B 响应给我。Http 超时时闯。2.问步接口中,如果网絡廷迟,可能会产生重复提交
接口重复提交如何解决? token(令牌)+图像验证

A 调用 B,如果 B 没有及时响应。A 项默从有 3 次重试补偿机,将该信息存近在日志表(补偿表),在使用定时 jpb 每天晚上健康检查数据,可以自己手动补偿,不是实时。B 责任:处理幂等问题

缺点:有可能会阻塞,超时,数据不一致

Java(JMM)内存模型
多线程线程安全
多线程之间通讯
ConcurrentHashMap 原理分析
Vector 与 ArrayList 线程安全源码分析 Volatile 可见性
ThreadLock 原理剖析
Atomiclntege 原子类
JDK15 以后常用并发包
创建线程池四种方式
线程池原理分析
线程数合理配置
Java 常用锁机制
重入锁、自旋锁、重入锁
负载均衡采用取模算法
Java 实现定时任务有几种?
Redis,eureka,zookeeper 都可以做注册中心

Thread
TimeTask
线程池,可定时线程
Quartz 定时任务调度框架
Springboot 内置定时任务调度

分布式 job 怎么解决幂等性问题?

1. 使用分布式锁,(zookeeper,redis 都可以实现)保证只有一台服务器执行 job
2. 使用配置文件,配置文件开关加一个 start=true 或者 start=false,如为 true,执行 job,如果为 false,不执行
3. 使用数据库唯一标识,缺点:效率低。(执行一个任务给数据库插入一个唯一 id,插入第二次的时候报错)

传统任务调度缺点:1.没有补偿机制 2.不支持集群 3.不支持路由策略(类似于 nginx 负载均衡一个道理)4.统计 5.管理平台 6.报警邮箱,监控状态

分布式任务调度平台 xxljob

Zookeeper 是分布式协调工具,而不单单是注册中心,zookeeper 应用场景?

1. 分布式锁
2. 负载均衡
3. 命名服务 dubbo
4. 分布式通知/协调 wtcher 事件通知

5. 发布订阅 wtcher？通知
6. 集群环境，选举

项目中使用分布式锁？

案例：需要生成订单号？

使用 UUID，时间戳+业务 id（也有可能会重复）

单点登陆注意：生成订单号唯一，使用同步代码块或着 lock 机制

分布式集群环境：

1. 大公司，订单号提前生成好存放在 redis 中，（好几个 tomcat）
2. 使用分布式锁：①使用数据库实现分布锁（tomcat01 给数据库能插入数据就拿到锁，别的 tomcat 拿不到就等待，等到 tomcat01 用完了，数据库在释放锁，但是缺点释放不了锁就会导致死锁。

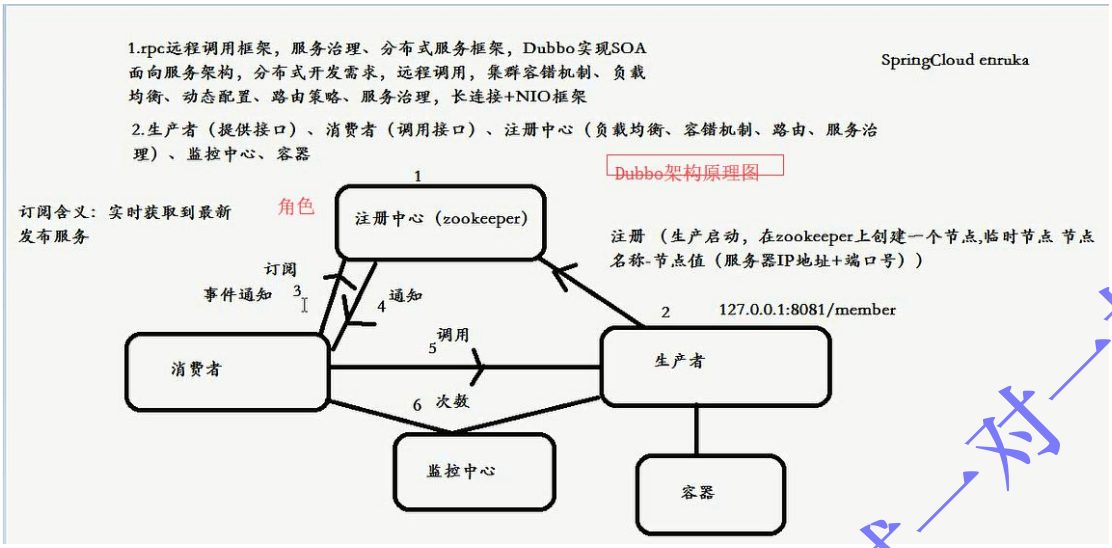
②使用缓存实现分布式锁

③使用 zookeeper 实现分布式锁

分布式锁原理：

首先多个服务器在 zookeeper 上创建同一个临时节点，只要谁创建临时节点成功，谁就拿到锁，其他服务没有在 zookeeper 上创建节点成功，应该等待。什么时候唤醒，使用事件通知获取节点被删除，重新进入到获取锁资源。（临时节点特征：会话连接失效后，值自动删除掉）

什 么 是 JMS ？



JMS 是 java 的消息服务，JMS 的客户端之间可以通过 JMS 服务进行异步的消息传输。

什么是消息模型

- Point-to-Point(P2P) --- 点对点
- Publish/Subscribe(Pub/Sub)--- 发布订阅

即点对点和发布订阅模型

MQ 产品的分类

RabbitMQ

是使用 Erlang 编写的一个开源的消息队列，本身支持很多的协议：AMQP，XMPP, SMTP, STOMP，也正是如此，使的它变的非常重量级，更适合于企业级的开发。同时实现了一个经纪人(Broker)构架，这意味着消息在发送给客户端时先在中心队列排队。对路由(Routing)，负载均衡(Load balance)或者数据持久化都有很好的支持。

Redis

是一个 Key-Value 的 NoSQL 数据库，开发维护很活跃，虽然它是一个 Key-Value 数据库存储系统，但它本身支持 MQ 功能，所以完全可以当做一个轻量级的队列服务来使用。对于 RabbitMQ 和 Redis 的入队和出队操作，各执行 100 万次，每 10 万次记录一次执行时间。测试数据分为 128Bytes、512Bytes、1K 和 10K 四个不同大小的数据。实验表明：入队时，当数据比较小时 Redis 的性能要高于 RabbitMQ，而如果数据大小超过了 10K，Redis 则慢的无法忍受；出队时，无论数据大小，Redis 都表现出非常好的性能，而 RabbitMQ 的出队性能则远低于 Redis。

	入队	出队

	128B	512B	1K	10K	128B	512B	1K	10K
Redis	16088	15961	17094	25	15955	20449	18098	9355
RabbitMQ	10627	9916	9370	2366	3219	3174	2982	1588

ZeroMQ

号称最快的消息队列系统，尤其针对大吞吐量的需求场景。ZMQ 能够实现 RabbitMQ 不擅长的高级/复杂的队列，但是开发人员需要自己组合多种技术框架，技术上的复杂度是对这 MQ 能够应用成功的挑战。ZeroMQ 具有一个独特的非中间件的模式，你不需要安装和运行一个消息服务器或中间件，因为你的应用程序将扮演了这个服务角色。你只需要简单的引用 ZeroMQ 程序库，可以使用 NuGet 安装，然后你就可以愉快的在应用程序之间发送消息了。但是 ZeroMQ 仅提供非持久性的队列，也就是说如果 down 机，数据将会丢失。其中，Twitter 的 Storm 中使用 ZeroMQ 作为数据流的传输。

ActiveMQ

是 Apache 下的一个子项目。 类似于 ZeroMQ，它能够以代理人和点对点的技术实现队列。同时类似于 RabbitMQ，它少量代码就可以高效地实现高级应用场景。RabbitMQ、ZeroMQ、ActiveMQ 均支持常用的多种语言客户端 C++、Java、.Net、Python、 Php、 Ruby 等。

Jafka/Kafka

Kafka 是 Apache 下的一个子项目，是一个高性能跨语言分布式 Publish/Subscribe 消息队列系统，而 Jafka 是在 Kafka 之上孵化而来的，即 Kafka 的一个升级版。具有以下特性：快速持久化，可以在 O(1)的系统开销下进行消息持久化；高吞吐，在一台普通的服务器上既可以达到 10W/s 的吞吐速率；完全的分布式系统，Broker、Producer、Consumer 都原生自动支持分布式，自动实现复杂均衡；支持 Hadoop 数据并行加载，对于像 Hadoop 的一样的日志数据和离线分析系统，但又要求实时处理的限制，这是一个可行的解决方案。Kafka 通过 Hadoop 的并行加载机制来统一了在线和离线的消息处理，这一点也是本课题所研究系统所看重的。Apache Kafka 相对于 ActiveMQ 是一个非常轻量级的消息系统，除了性能非常好之外，还是一个工作良好的分布式系统。

其他一些队列列表 HornetQ、Apache Qpid、Sparrow、Starling、Kestrel、Beanstalkd、Amazon SQS 就不再一一分析。

MQ 怎么保证消息幂等问题

发送端 MQ-client 将消息发送给服务端 MQ-server
服务端 MQ-server 将消息落地
服务端 MQ-server 回 ACK(表示确认) 2.如果 3 丢失 发送端在超时后，又会发送一遍，此时重发是 MQ-client 发起的，消息处理的是 MQ-server 为了避免 2

重复落地，对每条 MQ 消息系统内部需要生成一个 inner-msg-id,作为去重和幂等的依据，这个内部消息 ID 的特点是

在分布式环境中，MQ 通讯产生网络延迟，重试补偿中，会造成 MQ 重复消费。

解决办法:

使用日志+msg-id 保存报文信息，作为去重和幂等的依据。

消费端代码抛出异常，不需要重试补偿，使用日志记录报文，下次发版本解决。

MQ 有哪些协议

Stomp、XMPP

Stomp 协议，英文全名 Streaming Text Orientated Message Protocol，中文名称为 '流文本定向消息协议'。是一种以纯文本为载体的协议（以文本为载体的意思是它的消息格式规范中没有类似 XMPP 协议那样的 xml 格式要求，你可以将它看作'半结构化数据'）。目前 Stomp 协议有两个版本：V1.1 和 V1.2。

一个标准的 Stomp 协议包括以下部分：命令/信息关键字、头信息、文本内容。如下图所示：

消息队列异步通讯与同步通讯区别

同步通讯是客户端直接将请求发往服务器，等待服务器处理完请求并返回响应信息后才会继续向下执行。消息队列独立于客户端和服务器端，单独架设消息队列服务器，对于不必立即获取响应和处理过程复杂的请求，客户端可以将请求发往消息队列然后立即返回，指定的消费者处理请求，这样客户端不必持续等待。

JMS 消息通讯模型有那些

消息队列：点对点，发布订阅

消息中间应用场景

发送邮件或短信的服务、秒杀、运行过程复杂耗时的服务。

发布订阅与点对点通讯的区别

发布订阅是生产者发布一个主题，所有订阅该主题的消费者都会参与消费，消息会被重复消费。点对点通讯是一个消息只能有一个消费者消费，需要保证数据的幂等性。

如何保证 JMS 可靠消息

ActiveMQ 采用消息签收机制保证数据的可靠性，消息签收有三种方式：自动签收、手动签收、事务，默认自动签收。如果是带事务的消息，事务执行完毕后自动签收，事务回滚则重新发送。

ActiveMQ 服务端宕机了，消息会丢失吗？

生产者可以通过 setDeliveryMode 方法设置消息模式，当设置未非持久化时服务器宕机后消息将销毁，重启服务器后无法继续消费。当设置为持久化时服务器宕机后消息将保存到服务器中，重启后消费者还可以继续消费未处理完毕的消息。

RabbitMQ 与其他 MQ 有什么不同？

RabbitMQ 是用 erlang 语言实现，安装 RabbitMQ 之前需要先安装 erlang 环境。RabbitMQ 只支持 AMQP 协议，用于对稳定性要求比较高的企业。

谈谈 RabbitMQ 五种列队形式

点对点模式：生产者生成的消息由一个消费者消费；

工作模式：在消费者集群的情况下，可以根据消费者服务器的性能分配消息，即性能好的服务器多消费，性能次的少消费。

发布订阅模式：在生产者和队列之间插入一个交换机，由交换机转发到与该交换机绑定的队列；

路由模式：路由模式是基于发布订阅模式，只是在生产者向交换机生产消息时板顶一个 routingkey，交换机向绑定同一个 routingkey 的队列转发消息；

通配符模式：是对路由模式的补充，使用通配符进行 routingkey 匹配，通配符有#和*，#代表匹配多个，*代表匹配一个。

RabbitMQ 四中交换机类型

Fanout：以这种方式连接到交换机的队列都可以获得交换机的转发；

Direct：生产者绑定 direct 类型的交换机，在向交换机发送消息时绑定 routingkey，交换机会将这条消息发送到相同 routingkey 的队列。

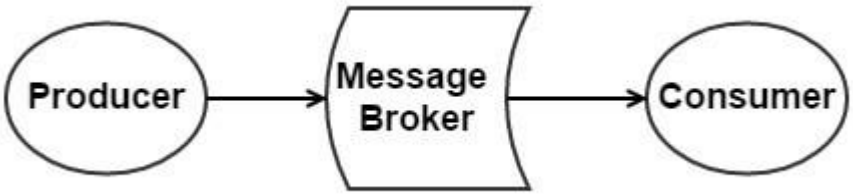
Topic：和 Direct 相似，可以在 routingKey 中使用通配符，#代表多个匹配，*代表单个匹配，routingkey 使用"."作为分隔符。

Headers：类似 Direct，使用多个消息代替路由键建立路由规则，通过消息头匹配来转发消息。

RabbitMQQMAP 协议原理

在 RabbitMQ 中，生产者将消息发送到交换机，交换机将消息根据路由策略将消息发送到绑定的消息队列，消费者通过消息队列获取并消费消息。

消息队列由哪些角色组成？

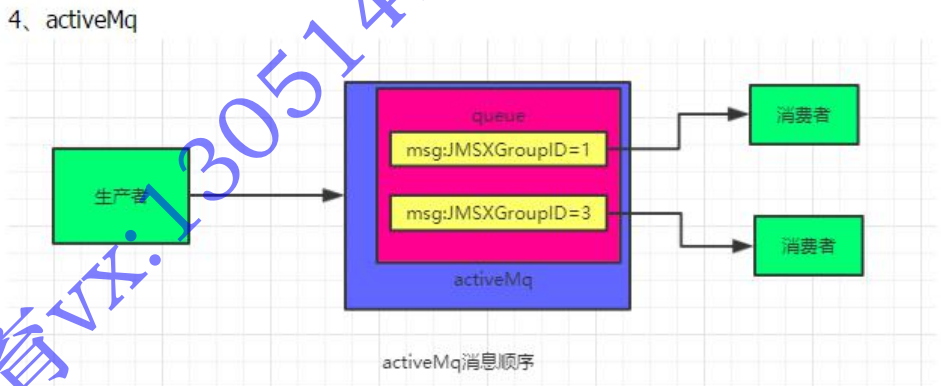


生产者（Producer）：负责产生消息。
消费者（Consumer）：负责消费消息
消息代理（Message Broker）：负责存储消息和转发消息两件事情。其中，转发消息分为推送和拉取两种方式。
拉取（Pull），是指 Consumer 主动从 Message Broker 获取消息
推送（Push），是指 Message Broker 主动将 Consumer 感兴趣的消息推送给 Consumer 。

消息队列有哪些使用场景？

应用解耦
异步处理
流量削峰
消息通讯
日志处理

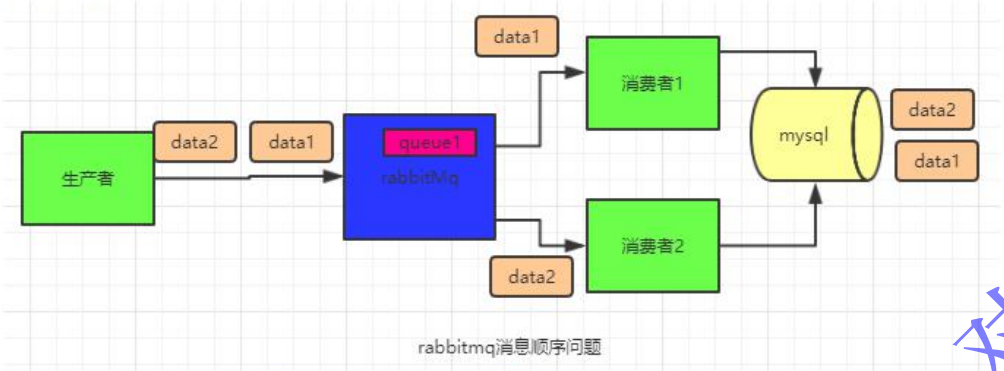
如何保证消息的顺序性?



如图，activeMq 里面有 messageGroups 属性，可以指定 JMSXGroupID，消费者会消费指定的 JMSXGroupID。即保证了顺序性，又解决负载均衡的问题。

1、rabbitMq

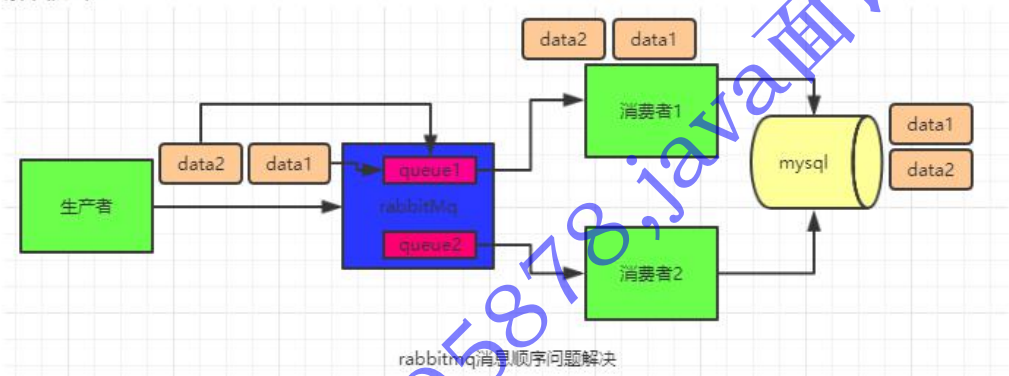
问题分析：



rabbitmq消息顺序问题

如图，data1 和 data2 是有顺序的，必须 data1 先执行，data2 后执行；这两个数据被不同的消费者消费到了，可能 data2 先执行，data1 后执行，这样原来的顺序就错乱了。

解决方案：



rabbitmq消息顺序问题解决

如图，在 MQ 里面创建多个 queue，同一规则的数据（对唯一标识进行 hash），有顺序的放入 MQ 的 queue 里面，消费者只取一个 queue 里面获取数据消费，这样执行的顺序是有序的。或者还是只有一个 queue 但是对应一个消费者，然后这个消费者内部用内存队列做排队，然后分发给底层不同的 worker 来处理。

为什么要使用消息队列？

解耦和异步处理，以及在高并发场景下平滑短时间内大量的服务请求。

如何保证消息的可靠性传输(如何处理消息丢失的问题)?

生产者弄丢了数据

生产者将数据发送到 RabbitMQ 的时候，可能数据就在半路给搞丢了，因为网络啥的问题，都有可能。

此时可以选择用 RabbitMQ 提供的事务功能，就是生产者发送数据之前开启 RabbitMQ 事务（channel.txSelect），然后发送消息，如果消息没有成功被 RabbitMQ 接收到，那么生产者会收到异常报错，此时就可以回滚事务（channel.txRollback），然后重试发送消息；如果收到了消息，那么可以提交事务（channel.txCommit）。这种方式会降低吞吐量，影响性能。

还有一种方式是开启 confirm 模式，在生产者那里设置开启 confirm 模式之后，你每次写的消息都会分配一个唯一的 id，然后如果写入了 RabbitMQ 中，

RabbitMQ 会给你回传一个 ack 消息，告诉你说这个消息 ok 了。如果 RabbitMQ 没能处理这个消息，会回调你一个 nack 接口，告诉你这个消息接收失败，你可以重试。而且你可以结合这个机制自己在内存里维护每个消息 id 的状态，如果超过一定时间还没接收到这个消息的回调，那么你可以重发。

事务机制和 cnofirm 机制最大的不同在于，事务机制是同步的，你提交一个事务之后会阻塞在那儿，但是 confirm 机制是异步的，你发送个消息之后就可以发送下一个消息，然后那个消息 RabbitMQ 接收了之后会异步回调你一个接口通知你这个消息接收到了。 所以一般在生产者这块避免数据丢失，都是用 confirm 机制的。

RabbitMQ 弄丢了数据

就是 RabbitMQ 自己弄丢了数据，这个你必须开启 RabbitMQ 的持久化，消息写入之后会持久化到磁盘，哪怕是 RabbitMQ 自己挂了，恢复之后会自动读取之前存储的数据，一般数据不会丢。

除非极其罕见的是，RabbitMQ 还没持久化，自己就挂了，可能导致少量数据会丢失的，但是这个概率较小。

设置持久化有两个步骤，第一个是创建 queue 的时候将其设置为持久化的，这样就可以保证 RabbitMQ 持久化 queue 的元数据，但是不会持久化 queue 里的数据；

第二个是发送消息的时候将消息的 deliveryMode 设置为 2，就是将消息设置为持久化的，此时 RabbitMQ 就会将消息持久化到磁盘上去。

必须要同时设置这两个持久化才行，RabbitMQ 哪怕是挂了，再次重启，也会从磁盘上重启恢复 queue，恢复这个 queue 里的数据。

而且持久化可以跟生产者那边的 confirm 机制配合起来，只有消息被持久化到磁盘之后，才会通知生产者 ack 了，所以哪怕是在持久化到磁盘之前，RabbitMQ 挂了，数据丢了，生产者收不到 ack，你也是可以自己重发的。

给 RabbitMQ 开启了持久化机制后，也还有一种可能，就是这个消息写到了 RabbitMQ 中，但是还没来得及持久化到磁盘上，结果不巧，此时 RabbitMQ 挂了，就会导致内存里的一点点数据会丢失。

消费端弄丢了数据

消费的时候，刚消费到，还没处理，结果进程挂了，比如重启了，RabbitMQ 认为你都消费了，这数据就丢了。

这个时候得用 RabbitMQ 提供的 ack 机制，简单来说，就是你关闭 RabbitMQ 自动 ack，可以通过一个 api 来调用就行，然后每次你自己代码里确保处理完的时候，再程序里 ack 一把。

这样的话，如果你还没处理完，不就没有 ack？那 RabbitMQ 就认为你还没处理完，这个时候 RabbitMQ 会把这个消费分配给别的 consumer 去处理，消息是不会丢的。

ActiveMQ 处理失败时的消息重发机制

1. 处理失败 指的是 MessageListener 的 onMessage 方法里抛出 RuntimeException。

2. Message 头里有两个相关字段：Redelivered 默认为 false，redeliveryCounter 默认为 0。

3. 消息先由 broker 发送给 consumer，consumer 调用 listener，如果处理失败，本地 redeliveryCounter++，给 broker 一个特定应答，broker 端的 message 里 redeliveryCounter++，延迟一点时间继续调用 默认 1s。超过 6 次，则给 broker 另一个特定应答，broker 就直接发送消息到 DLQ。

4. 如果失败 2 次，consumer 重启，则 broker 再推过来的消息里，redeliveryCounter=2，本地只能再重试 4 次即会进入 DLQ。

5. 重试的特定应答发送到 broker，broker 即会在内存将消息的 redelivered 设置为 true，redeliveryCounter++，但是这两个字段都没有持久化，即没有修改存储中的消息记录。所以 broker 重启时这两个字段会被重置为默认值。

消息队列有什么优缺点？

任何中间件的引入，带来优点的时候，也同时会带来缺点。

优点，在上述的「消息队列有哪些使用场景？」问题中，我们已经看到了。

缺点，主要是如下三点：

系统可用性降低。

系统引入的外部依赖越多，越容易挂掉。本来你就是 A 系统调用 BCD 三个系统的接口就好了，本来 ABCD 四个系统好好的，没啥问题，你偏加个 MQ 进来，万一 MQ 挂了咋整，MQ 一挂，整套系统崩溃的，你不就完了？所以，消息队列一定要做好高可用。

系统复杂度提高。

主要需要多考虑，1）消息怎么不重复消息。2）消息怎么保证不丢失。3）需要消息顺序的业务场景，怎么处理。

一致性问题。

A 系统处理完了直接返回成功了，人都以为你这个请求就成功了。但是问题是，要是 B、C。D 三个系统那里，B、D 两个系统写库成功了，结果 C 系统写库失败了，咋整？你这数据就不一致了。

当然，这不仅仅是 MQ 的问题，引入 RPC 之后本身就存在这样的问题。如果我们在使用 MQ 时，一定要达到数据的最终一致性。即，C 系统最终执行完成。

消息队列有几种消费语义？

一共有 3 种，分别如下：

消息至多被消费一次（At most once）：消息可能会丢失，但绝不重传。

消息至少被消费一次（At least once）：消息可以重传，但绝不丢失。

消息仅被消费一次（Exactly once）：每一条消息只被传递一次。

消息队列有几种投递方式？分别有什么优缺点

在「消息队列由哪些角色组成？」中，我们已经提到消息队列有 push 推送和 pull 拉取两种投递方式。

一种模型的某些场景下的优点，在另一些场景就可能是缺点。无论是 push 还是 pull ，都存在各种的利弊。

push

优点，就是及时性。

缺点，就是受限于消费者的消费能力，可能造成消息的堆积，Broker 会不断给消费者发送不能处理的消息。

pull

优点，就是主动权掌握在消费方，可以根据自己的消息速度进行消息拉取。

缺点，就是消费方不知道什么时候可以获取的最新的消息，会有消息延迟和忙等。

1. 如何确保消息正确地发送至 RabbitMQ？

RabbitMQ 使用发送方确认模式，确保消息正确地发送到 RabbitMQ。

发送方确认模式：将信道设置成 confirm 模式（发送方确认模式），则所有在信道上发布的消息都会被指派一个唯一的 ID。一旦消息被投递到目的队列后，或者消息被写入磁盘后（可持久化的消息），信道会发送一个确认给生产者（包含消息唯一 ID）。如果 RabbitMQ 发生内部错误从而导致消息丢失，会发送一条 nack（not acknowledged，未确认）消息。

发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者应用程序，生产者应用程序的回调方法就会被触发来处理确认消息。

2. 如何确保消息接收方消费了消息？

接收方消息确认机制：消费者接收每一条消息后都必须进行确认（消息接收和消息确认是两个不同操作）。只有消费者确认了消息，RabbitMQ 才能安全地把消息从队列中删除。

这里并没有用到超时机制，RabbitMQ 仅通过 Consumer 的连接中断来确认是否需要重新发送消息。也就是说，只要连接不中断，RabbitMQ 给了 Consumer 足够长的时间来处理消息。

下面罗列几种特殊情况：

如果消费者接收到消息，在确认之前断开了连接或取消订阅，RabbitMQ 会认为消息没有被分发，然后重新分发给下一个订阅的消费者。（可能存在消息重复消费的隐患，需要根据 bizId 去重）

如果消费者接收到消息却没有确认消息，连接也未断开，则 RabbitMQ 认为该消费者繁忙，将不会给该消费者分发更多的消息。

3. 如何避免消息重复投递或重复消费？

在消息生产时，MQ 内部针对每条生产者发送的消息生成一个 inner-msg-id，作为去重和幂等的依据（消息投递失败并重传），避免重复的消息进入队列；在消息消费时，要求消息体中必须要有一个 bizId（对于同一业务全局唯一，如支付 ID、订单 ID、帖子 ID 等）作为去重和幂等的依据，避免同一条消息被重复消费。

4. 消息基于什么传输？

由于 TCP 连接的创建和销毁开销较大，且并发数受系统资源限制，会造成性能瓶颈。RabbitMQ 使用信道的方式来传输数据。信道是建立在真实的 TCP 连接内的虚拟连接，且每条 TCP 连接上的信道数量没有限制。

5. 消息如何分发？

若该队列至少有一个消费者订阅，消息将以循环（round-robin）的方式发送给消费者。每条消息只会分发给一个订阅的消费者（前提是消费者能够正常处理消息并进行确认）。

6. 消息怎么路由？

从概念上来说，消息路由必须有三部分：交换器、路由、绑定。生产者把消息发布到交换器上；绑定决定了消息如何从路由器路由到特定的队列；消息最终到达队列，并被消费者接收。

消息发布到交换器时，消息将拥有一个路由键（routing key），在消息创建时设定。

通过队列路由键，可以把队列绑定到交换器上。

消息到达交换器后，RabbitMQ 会将消息的路由键与队列的路由键进行匹配（针对不同的交换器有不同的路由规则）。如果能够匹配到队列，则消息会投递到相应队列中；如果不能匹配到任何队列，消息将进入“黑洞”。

常用的交换器主要分为一下三种：

direct：如果路由键完全匹配，消息就被投递到相应的队列

fanout：如果交换器收到消息，将会广播到所有绑定的队列上

topic：可以使来自不同源头的消息能够到达同一个队列。使用 topic 交换器时，可以使用通配符，比如：“*”匹配特定位置的任意文本，“.”把路由键分为了几部分，“#”匹配所有规则等。特别注意：发往 topic 交换器的消息不能随意的设置选择键（routing_key），必须是由"."隔开的一系列的标识符组成。

7. 如何确保消息不丢失？

消息持久化的前提是：将交换器/队列的 durable 属性设置为 true，表示交换器/队列是持久交换器/队列，在服务器崩溃或重启之后不需要重新创建交换器/队列（交换器/队列会自动创建）。

如果消息想要从 Rabbit 崩溃中恢复，那么消息必须：

在消息发布前，通过把它的“投递模式”选项设置为 2（持久）来把消息标记成持久化

将消息发送到持久交换器

消息到达持久队列

RabbitMQ 确保持久性消息能从服务器重启中恢复的方式是，将它们写入磁盘上的一个持久化日志文件，当发布一条持久性消息到持久交换器上时，Rabbit 会在消息提交到日志文件后才发送响应（如果消息路由到了非持久队列，它会自动从持久化日志中移除）。一旦消费者从持久队列中消费了一条持久化消息，RabbitMQ 会在持久化日志中把这条消息标记为等待垃圾收集。如果持久化消息在被消费之前 RabbitMQ 重启，那么 Rabbit 会自动重建交换器和队列（以及绑定），并重播持久化日志文件中的消息到合适的队列或者交换器上。

8. 使用 RabbitMQ 有什么好处？

应用解耦（系统拆分）

异步处理（预约挂号业务处理成功后，异步发送短信、推送消息、日志记录等）

消息分发

流量削峰

消息缓冲

.....

9. 其他

RabbitMQ 是 消息投递服务，在应用程序和服务器之间扮演路由器的角色，

而应用程序或服务器可以发送和接收包裹。其通信方式是一种“发后即忘（fire-and-forget）”的单向方式。

其中消息包含两部分内容：有效载荷（payload）和标签（label）。

使用 RabbitMQ 有什么好处？

应用解耦（系统拆分）

异步处理（预约挂号业务处理成功后，异步发送短信、推送消息、日志记录等）

消息分发

流量削峰

消息缓冲

消息基于什么传输？

由于 TCP 连接的创建和销毁开销较大，且并发数受系统资源限制，会造成性能瓶颈。RabbitMQ 使用信道的方式来传输数据。信道是建立在真实的 TCP 连接内的虚拟连接，且每条 TCP 连接上的信道数量没有限制。

消息怎么路由？

从概念上来说，消息路由必须有三部分：交换器、路由、绑定。生产者把消息发布到交换器上；绑定决定了消息如何从路由器路由到特定的队列；消息最终到达队列，并被消费者接收。常用的交换器主要分为一下三种：

direct：如果路由键完全匹配，消息就被投递到相应的队列

fanout：如果交换器收到消息，将会广播到所有绑定的队列上

topic：可以使来自不同源头的消息能够到达同一个队列。 使用 topic 交换器时，可以使用通配符，比如：“*”匹配特定位置的任意文本，“.”把路由键分为了几部分，“#”匹配所有规则等。特别注意：发往 topic 交换器的消息不能随意的设置选择键（routing_key），必须是由"."隔开的一系列的标识符组成。

如何做到信息的可靠性？确保消息正确地发送至 RabbitMQ？确保消息接受方消费了消息？休息不丢失不重复？

采用事务或发送方确认模式保证消息发送到 rabbitmq。 发送方确认模式：将信道设置成 confirm 模式（发送方确认模式），则所有在信道上发布的消息都会被指派一个唯一的 ID。一旦消息被投递到目的队列后，或者消息被写入磁盘后（可持久化的消息），信道会发送一个确认给生产者（包含消息唯一 ID）。如果 RabbitMQ 发生内部错误从而导致消息丢失，会发送一条 nack（not acknowledged，未确认）消息。发送方确认模式是异步的，生产者应用程序在等待确认的同时，可以继续发送消息。当确认消息到达生产者应用程序，生产者应用程序的回调方法就会被触发来处理确认消息。

接收方消息确认机制来确保消息被消费：消费者接收每一条消息后都必须进行确认（消息接收和消息确认是两个不同操作）。只有消费者确认了消息，RabbitMQ 才能安全地把消息从队列中删除。RabbitMQ 仅通过 Consumer 的连接中断来确认是否需要重新发送消息。也就是说，只要连接不中断，RabbitMQ 给了 Consumer 足够长的时间来处理消息。消费者接收到消息，在确认之前断开

了连接或取消订阅，RabbitMQ 会认为消息没有被分发，然后重新分发给下一个订阅的消费者。（可能存在消息重复消费的隐患，需要根据 bizId 去重）

在消息生产时，MQ 内部针对每条生产者发送的消息生成一个 inner-msg-id，作为去重和幂等的依据（消息投递失败并重传），避免重复的消息进入队列；在消息消费时，要求消息体中必须要有一个 bizId（对于同一业务全局唯一，如支付 ID、订单 ID、帖子 ID 等）作为去重和幂等的依据，避免同一条消息被重复消费。

ActiveMQ 服务器宕机怎么办？

这得从 ActiveMQ 的储存机制说起。在通常的情况下，非持久化消息是存储在内存中的，持久化消息是存储在文件中的，它们的最大限制在配置文件的<systemUsage>节点中配置。但是，在非持久化消息堆积到一定程度，内存告急的时候，ActiveMQ 会将内存中的非持久化消息写入临时文件中，以腾出内存。虽然都保存到了文件里，但它和持久化消息的区别是，重启后持久化消息会从文件中恢复，非持久化的临时文件会直接删除。

那如果文件增大到达了配置中的最大限制的时候会发生什么？我做了以下实验：

设置 2G 左右的持久化文件限制，大量生产持久化消息直到文件达到最大限制，此时生产者阻塞，但消费者可正常连接并消费消息，等消息消费掉一部分，文件删除又腾出空间之后，生产者又可继续发送消息，服务自动恢复正常。

设置 2G 左右的临时文件限制，大量生产非持久化消息并写入临时文件，在达到最大限制时，生产者阻塞，消费者可正常连接但不能消费消息，或者原本慢速消费的消费者，消费突然停止。整个系统可连接，但是无法提供服务，就这样挂了。

具体原因不详，解决方案：尽量不要用非持久化消息，非要用的话，将临时文件限制尽可能的调大。

丢消息怎么办？

这得从 java 的 java.net.SocketException 异常说起。简单点说就是当网络发送方发送一堆数据，然后调用 close 关闭连接之后。这些发送的数据都在接收者的缓存里，接收者如果调用 read 方法仍旧能从缓存中读取这些数据，尽管对方已经关闭了连接。但是当接收者尝试发送数据时，由于此时连接已关闭，所以会发生异常，这个很好理解。不过需要注意的是，当发生 SocketException 后，原本缓存区中数据也作废了，此时接收者再次调用 read 方法去读取缓存中的数据，就会报 Software caused connection abort: recv failed 错误。

通过抓包得知，ActiveMQ 会每隔 10 秒发送一个心跳包，这个心跳包是服务器发送给客户端的，用来判断客户端死没死。如果你看过上面第一条，就会知道非持久化消息堆积到一定程度会写到文件里，这个写的过程会阻塞所有动作，而且会持续 20 到 30 秒，并且随着内存的增大而增大。当客户端发完消息调用 connection.close()时，会期待服务器对于关闭连接的回答，如果超过 15 秒没回答就直接调用 socket 层的 close 关闭 tcp 连接了。这时客户端发出的消息其实还在服务器的缓存里等待处理，不过由于服务器心跳包的设置，导致发生了 java.net.SocketException 异常，把缓存里的数据作废了，没处理的消息全部丢失。

解决方案：用持久化消息，或者非持久化消息及时处理不要堆积，或者启动事务，启动事务后，commit()方法会负责任的等待服务器的返回，也就不会关闭连接导致消息丢失了。

持久化消息非常慢。

默认的情况下，非持久化的消息是异步发送的，持久化的消息是同步发送的，遇到慢一点的硬盘，发送消息的速度是无法忍受的。但是在开启事务的情况下，消息都是异步发送的，效率会有 2 个数量级的提升。所以在发送持久化消息时，请务必开启事务模式。其实发送非持久化消息时也建议开启事务，因为根本不会影响性能。

消息的不均匀消费

有时在发送一些消息之后，开启 2 个消费者去处理消息。会发现一个消费者处理了所有的消息，另一个消费者根本没收到消息。原因在于 ActiveMQ 的 prefetch 机制。当消费者去获取消息时，不会一条一条去获取，而是一次性获取一批，默认是 1000 条。这些预获取的消息，在还没确认消费之前，在管理控制台还是可以看见这些消息的，但是不会再分配给其他消费者，此时这些消息的状态应该算作“已分配未消费”，如果消息最后被消费，则会在服务器端被删除，如果消费者崩溃，则这些消息会被重新分配给新的消费者。但是如果消费者既不消费确认，又不崩溃，那这些消息就永远躺在消费者的缓存区里无法处理。更通常的情况是，消费这些消息非常耗时，你开了 10 个消费者去处理，结果发现只有一台机器吭哧吭哧处理，另外 9 台啥事不干。

解决方案：将 prefetch 设为 1，每次处理 1 条消息，处理完再去取，这样也慢不了多少。

死信队列。

如果你想在消息处理失败后，不被服务器删除，还能被其他消费者处理或重试，可以关闭 AUTO_ACKNOWLEDGE，将 ack 交由程序自己处理。那如果使用了 AUTO_ACKNOWLEDGE，消息是什么时候被确认的，还有没有阻止消息确认的方法？有！

消费消息有 2 种方法，一种是调用 consumer.receive()方法，该方法将阻塞直到获得并返回一条消息。这种情况下，消息返回给方法调用者之后就自动被确认了。另一种方法是采用 listener 回调函数，在有消息到达时，会调用 listener 接口的 onMessage 方法。在这种情况下，在 onMessage 方法执行完毕后，消息才会被确认，此时只要在方法中抛出异常，该消息就不会被确认。那么问题来了，如果一条消息不能被处理，会被退回服务器重新分配，如果只有一个消费者，该消息又会重新被获取，重新抛异常。就算有多个消费者，往往在一个服务器上不能处理的消息，在另外的服务器上依然不能被处理。难道就这么退回--获取--报错死循环了吗？

在重试 6 次后，ActiveMQ 认为这条消息是“有毒”的，将会把消息丢到死信队列里。如果你的消息不见了，去 ActiveMQ.DLQ 里找找，说不定就躺在那里。

ActiveMQ 中的消息重发时间间隔和重发次数吗？

ActiveMQ：是 Apache 出品，最流行的，能力强劲的开源消息总线。是一个

完全支持 JMS1.1 和 J2EE 1.4 规范的 JMS Provider 实现。JMS（Java 消息服务）：是一个 Java 平台中关于面向消息中间件（MOM）的 API，用于在两个应用程序之间，或分布式系统中发送消息，进行异步通信。

首先，我们得大概了解下，在哪些情况下，ActiveMQ 服务器会将消息重发给消费者，这里为简单起见，假定采用的消息发送模式为队列（即消息发送者和消息接收者）。

① 如果消息接收者在处理完一条消息的处理过程后没有对 MOM 进行应答，则该消息将由 MOM 重发.

② 如果我们队某个队列设置了预读参数（consumer.prefetchSize），如果消息接收者在处理第一条消息时（没向 MOM 发送消息接收确认）就宕机了，则预读数量的所有消息都将被重发!

③ 如果 Session 是事务的，则只要消息接收者有一条消息没有确认，或发送消息期间 MOM 或客户端某一方突然宕机了，则该事务范围中的所有消息 MOM 都将重发。

④ 说到这里，大家可能会有疑问，ActiveMQ 消息服务器怎么知道消费者客户端到底是消息正在处理中还没来得急对消息进行应答还是已经处理完成了没有应答或是宕机了根本没机会应答呢？其实在所有的客户端机器上，内存中都运行着一套客户端的 ActiveMQ 环境，该环境负责缓存发来的消息，负责维持着和 ActiveMQ 服务器的消息通讯，负责失效转移（fail-over）等，所有的判断和处理都是由这套客户端环境来完成的。

我们可以来对 ActiveMQ 的重发策略(Redelivery Policy)来进行自定义配置，其中的配置参数主要有以下几个：

可用的属性

属性 默认值 说明

l collisionAvoidanceFactor 默认值 0.15，设置防止冲突范围的正负百分比，只有启用 useCollisionAvoidance 参数时才生效。

l maximumRedeliveries 默认值 6，最大重传次数，达到最大重连次数后抛出异常。为-1 时不限制次数，为 0 时表示不进行重传。

l maximumRedeliveryDelay 默认值 -1，最大传送延迟，只在 useExponentialBackOff 为 true 时有效（V5.5），假设首次重连间隔为 10ms，倍数为 2，那么第二次重连时间间隔为 20ms，第三次重连时间间隔为 40ms，当重连时间间隔大的最大重连时间间隔时，以后每次重连时间间隔都为最大重连时间间隔。

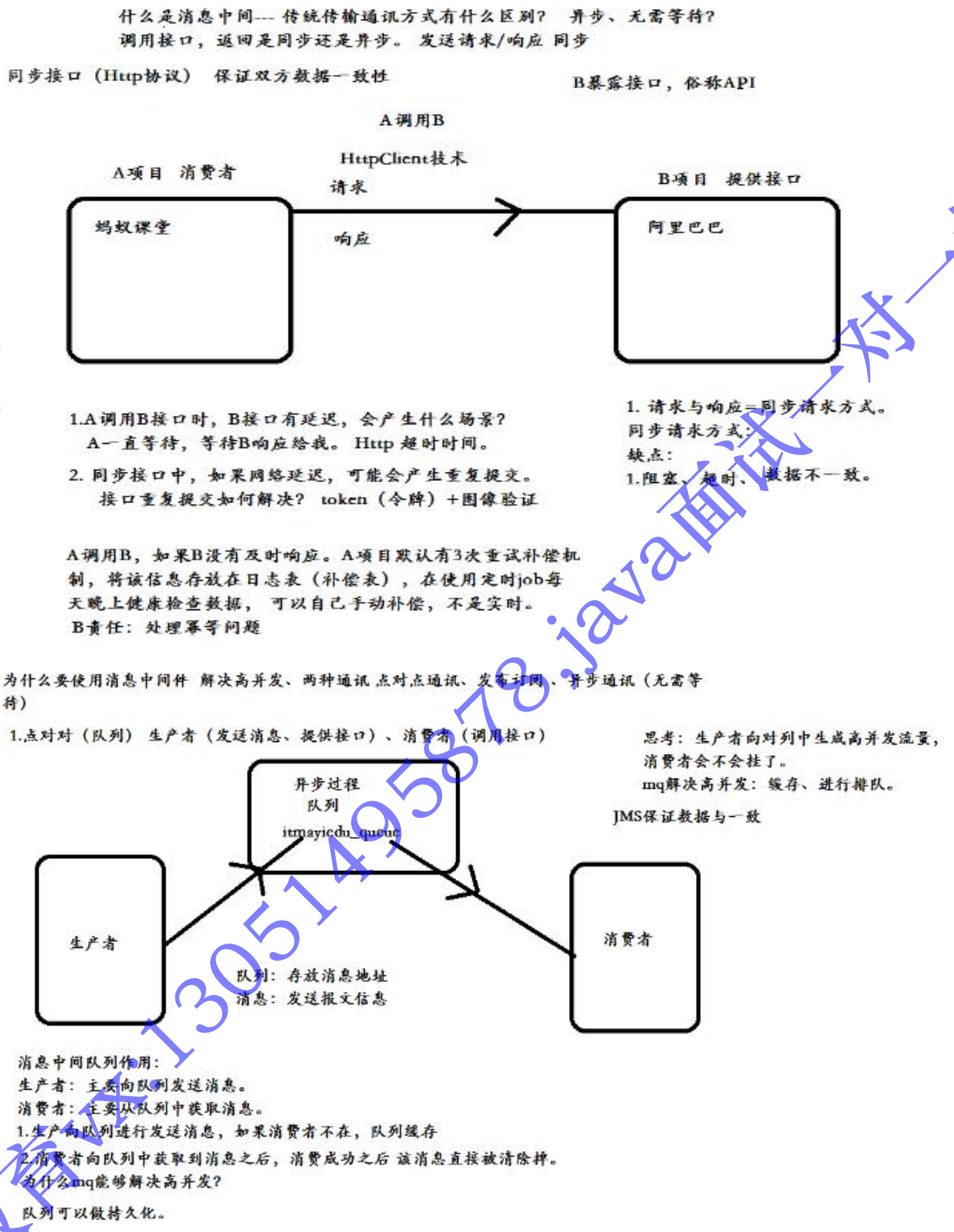
l initialRedeliveryDelay 默认值 1000L，初始重发延迟时间

l redeliveryDelay 默认值 1000L，重发延迟时间，当 initialRedeliveryDelay=0 时生效（v5.4）

l useCollisionAvoidance 默认值 false，启用防止冲突功能，因为消息接收时是可以使用多线程并发处理的，应该是为了重发的安全性，避开所有并发线程都在同一个时间点进行消息接收处理。所有线程在同一个时间点处理时会发生什么问题呢？应该没有问题，只是为了平衡 broker 处理性能，不会有时很忙，有时很空闲。

l useExponentialBackOff 默认值 false，启用指数倍数递增的方式增加延迟时间。

l backOffMultiplier 默认值 5，重连时间间隔递增倍数，只有值大于 1 和启用 useExponentialBackOff 参数时才生效。



JMS java 发送消息客户端与服务器进行异步通讯方式

1. 点对点通讯方式 （生产者，消息队列，消费者）（一对一）
2. 发布订阅（生产者，主题，消费者）

什么是消息中间件

发送者将消息发送给消息服务器，消息服务器将消感存放在若干队列中，在合适的时候再将消息转发给接收者。

这种模式下，发送和接收是异步的，发送者无需等待; 二者的生命周期未必相同: 发送消息的时候接收者不一定运行 ,接收消息的时

候 发送者也不一定运行;一对多通信: 对于一个消息可以有多个接收者。

什么是 JMS？

JMS 是 java 的消息服务，JMS 的客户端之间可以通过 JMS 服务进行异步的消息传输。

Kafka、ActiveMQ、RabbitMQ、RocketMQ 有什么优缺点？

特性	ActiveMQ	RabbitMQ	RocketMQ	Kafka
单机吞吐量	万级，比 RocketMQ、Kafka 低一个数量级	同 ActiveMQ	10 万级，支撑高吞吐	10 万级，高吞吐，一般配合大数据类的系统来进行实时数据计算、日志采集等场景
topic 数量对吞吐量的影响			topic 可以达到几百/几千的级别，吞吐量会有较小幅度的下降，这是 RocketMQ 的一大优势，在同等机器下，可以支撑大量的 topic	topic 从几十到几百个时候，吞吐量会大幅度下降，在同等机器下，Kafka 尽量保证 topic 数量不要过多，如果要支撑大规模的 topic，需要增加更多的机器资源
时效性	ms 级	微秒级，这是 RabbitMQ 的一大特点，延迟最低	ms 级	延迟在 ms 级以内
可用性	高，基于主从架构实现高可用	同 ActiveMQ	非常高，分布式架构	非常高，分布式，一个数据多个副本，少数机器宕机，不会丢失数据，不会导致不可用
消息可靠性	有较低的概率丢失数据		经过参数优化配置，可以做到 0 丢失	同 RocketMQ
功能支持	MQ 领域的功能极其完备	基于 erlang 开发，并发能力很强，性能极好，延时很低	MQ 功能较为完善，还是分布式的，扩展性好	功能较为简单，主要支持简单的 MQ 功能，在大数据领域的实时计算以及日志采集被大规模使用

总结

所以中小型公司，技术实力较为一般，技术挑战不是特别高，用 RabbitMQ 是不错的选择

大型公司，基础架构研发实力较强，用 RocketMQ 是很好的选择。

当然，中小型公司使用 RocketMQ 也是没什么问题的选择，特别是以 Java 为主语言的公司。

如果是大数据领域的实时计算、日志采集等场景，用 Kafka 是业内标准的，绝对没问题，社区活跃度很高，绝对不会黄，何况几乎是全世界这个领域的事实性规范。

另外，目前国内也是有非常多的公司，将 Kafka 应用在业务系统中，例如唯品会、陆金所、美团等等。

目前，芳芳的团队使用 RocketMQ 作为消息队列，因为有 RocketMQ 5 年左右使用经验，并且目前线上环境是使用阿里云，适合我们团队。

微服务与分布式

SpringBoot

Spring Boot 的主要优点：

为所有 Spring 开发者更快的入门

开箱即用，提供各种默认配置来简化项目配置

内嵌式容器简化 Web 项目

没有冗余代码生成和 XML 配置的要求

本章主要目标完成 Spring Boot 基础项目的构建，并且实现一个简单的 Http

请求处理，通过这个例子对 Spring Boot 有一个初步的了解，并体验其结构简单、开发快速的特性。

@RestController

在上加上 RestController 表示修饰该 Controller 所有的方法返回 JSON 格式，直接可以编写

Restful 接口

@EnableAutoConfiguration

注解:作用在于让 Spring Boot 根据应用所声明的依赖来对 Spring 框架进行自动配置

这个注解告诉 Spring Boot 根据添加的 jar 依赖猜测你想如何配置 Spring。由于 spring-boot-starter-web 添加了 Tomcat 和 Spring MVC，所以 auto-configuration 将假定你正在开发一个 web 应用并相应地对 Spring 进行设置。

SpringApplication.run(HelloController.class, args);

标识为启动类

SpringBoot 快速开发框架

什么是 SpringBoot

SpringBoot 是快速开发的 Spring 框架，能够快速整合主流框架，简化 xml 配置，采用全注解化，内置 Http 服务器（如 tomcat、jetty 等），通过 java 部署运行。

为什么要用 SpringBoot

快速开发，快速整合，配置简化、内嵌服务容器

SpringBoot 启动方式

主类 @SpringBootApplication 注解或添加 @ComponentScan 和 @EnableAutoConfiguration 注解，使用@SpringBootApplication 时注意自动扫描当前包

SpringBoot 与 SpringMVC 区别

SpringMVC 是 SpringBoot 的 Web 开发框架

SpringBoot 与 SpringCloud 区别

SpringBoot 是快速开发的 Spring 框架，SpringCloud 是完整的微服务框架，SpringCloud 依赖于 SpringBoot。

SpringBoot 中用那些注解

@EnableAutoConfiguration 作用

自动扫描并添加 jar 包依赖

@SpringBootApplication 原理
是一个组合注解，相当于@EnableAutoConfiguration 和@ComponentScan

SpringBoot 热部署使用什么？

devtools

热部署原理是什么？

热部署的实现原理主要依赖 java 的类加载机制，在实现方式可以概括为在容器启动的时候起一条后台线程，定时的检测类文件的时间戳变化，如果类的时间戳变掉了，则重新加载整个应用的 class 文件，同时重启服务，重新部署。

热部署原理与热加载区别是什么

热加载是在运行时重新加载 class 文件，不会重启服务。

你们项目中异常是如何处理

在 web 项目中，使用全局捕获异常返回统一错误信息。

SpringBoot 如何实现异步执行

在启动类添加@EnableAsync 表示开启对异步任务的支持，在异步服务上添加@Async

SpringBoot 多数据源拆分的思路

先在 properties 配置文件中配置两个数据源，创建分包 mapper，使用@ConfigurationProperties 读取 properties 中的配置，使用@MapperScan 注册到对应的 mapper 包中

SpringBoot 多数据源事务如何管理

第一种方式是在 service 层的 @TransactionManager 中使用 transactionManager 指定 DataSourceConfig 中配置的事务
第二种是使用 jta-atomikos 实现分布式事务管理

SpringBoot 如何实现打包

进入项目目录在控制台输入 mvn clean package，clean 是清空已存在的项目包，package 进行打包

SpringBoot 性能如何优化

如果项目比较大，类比较多，不使用@SpringBootApplication，采用@Compoment 指定扫包范围
在项目启动时设置 JVM 初始内存和最大内存相同
将 springboot 内置服务器由 tomcat 设置为 undertow

SpringBoot 执行流程

使用 SpringApplication.run() 启动，在该方法所在类添加

@SpringBootApplication 注解，该注解由@EnableAutoConfiguration 和@ComponentScan 等注解组成，@EnableAutoConfiguration 自动加载 SpringBoot 配置和依赖包，默认使用@ComponentScan 扫描当前包及子包中的所有类，将有 spring 注解的类交给 spring 容器管理

SpringBoot 底层实现原理

使用 maven 父子包依赖关系加载相关 jar 包，使用 java 操作 Spring 的初始化过程生成 class 文件，然后用 java 创建 tomcat 服务器加载这些 class 文件

SpringBoot 装配 Bean 的原理

通过@EnableAutoConfiguration 自动获取配置类信息，使用反射实例化为 spring 类，然后加载到 spring 容器

Spring Boot 是什么？

Spring Boot 是 Spring 的子项目，正如其名字，提供 Spring 的引导(Boot)的功能。

通过 Spring Boot ，我们开发者可以快速配置 Spring 项目，引入各种 Spring MVC、Spring Transaction、Spring AOP、MyBatis 等等框架，而无需不断重复编写繁重的 Spring 配置，降低了 Spring 的使用成本。

Spring Boot 提供了哪些核心功能？

1、独立运行 Spring 项目

Spring Boot 可以以 jar 包形式独立运行，运行一个 Spring Boot 项目只需要通过 java -jar xx.jar 来运行。

2、内嵌 Servlet 容器

Spring Boot 可以选择内嵌 Tomcat、Jetty 或者 Undertow，这样我们无须以 war 包形式部署项目。

3、提供 Starter 简化 Maven 配置

Spring 提供了一系列的 starter pom 来简化 Maven 的依赖加载。例如，当你使用了 spring-boot-starter-web ,会自动加入如下依赖 :带内嵌的 Tomcat，带来参数校验的依赖，带来 json 数据的转换器，带来 Springmvc 框架

4、自动配置 Spring Bean

Spring Boot 检测到特定类的存在，就会针对这个应用做一定的配置，进行自动配置 Bean ，这样会极大地减少我们要使用的配置。

当然，Spring Boot 只考虑大多数的开发场景，并不是所有的场景，若在实际开发中我们需要配置 Bean ,而 Spring Boot 没有提供支持，则可以自定义自动配置进行解决。

5、准生产的应用监控

Spring Boot 提供基于 HTTP、JMX、SSH 对运行时的项目进行监控。

6、无代码生成和 XML 配置

Spring Boot 没有引入任何形式的代码生成，它是使用的 Spring 4.0 的条件 @Condition 注解以实现根据条件进行配置。同时使用了 Maven /Gradle 的

依赖传递解析机制来实现 Spring 应用里面的自动配置。

Spring Boot 的优点

1、使【编码】变简单。

2、使【配置】变简单。
3、使【部署】变简单。
4、使【监控】变简单。

运行 Spring Boot 有哪几种方式？

1、打包成 Fat Jar ，直接使用 java -jar 运行。目前主流的做法，推荐。
2、在 IDEA 或 Eclipse 中，直接运行应用的 Spring Boot 启动类的 #main(String[] args) 启动。适用于开发调试场景。
3、如果是 Web 项目，可以打包成 War 包，使用外部 Tomcat 或 Jetty 等容器

如果更改内嵌 Tomcat 的端口？

方式一，修改 application.properties 配置文件的 server.port 属性。

```
server.port=9090
```

方式二，通过启动命令增加 server.port 参数进行修改。

```
java -jar xxx.jar --server.port=9090
```

Spring Boot 有哪几种读取配置的方式？

Spring Boot 目前支持 2 种读取配置：
@Value 注解，读取配置到属性。最最最常用。
另外，支持和 @PropertySource 注解一起使用，指定使用的配置文件。
@ConfigurationProperties 注解，读取配置到类上。
另外，支持和 @PropertySource 注解一起使用，指定使用的配置文件。

如何集成 Spring Boot 和 Spring MVC ？

引入 spring-boot-starter-web 的依赖。
实现 WebMvcConfigurer 接口，可添加自定义的 Spring MVC 配置。

Spring Boot 支持哪些日志框架？

Spring Boot 支持的日志框架有：
Logback
Log4j2
Log4j
Java Util Logging

Spring Cloud 核心功能是什么？

毫无疑问，Spring Cloud 可以说是目前微服务架构的最好的选择，涵盖了基本我们需要的所有组件，所以也被称为全家桶。Spring Cloud 主要提供了如下核心的功能：

Distributed/versioned configuration 分布式/版本化的配置管理
Service registration and discovery 服务注册与服务发现
Routing 路由
Service-to-service calls 端到端的调用
Load balancing 负载均衡
Circuit Breakers 断路器
Global locks 全局锁
Leadership election and cluster state 选举与集群状态管理
Distributed messaging 分布式消息

Spring Cloud 和 Spring Boot 的区别和关系？

Spring Boot 专注于快速方便的开发单个个体微服务。

Spring Cloud 是关注全局的微服务协调整理治理框架以及一整套的落地解决方案，它将 Spring Boot 开发的一个个单体微服务整合并管理起来，为各个微服务之间提供：配置管理，服务发现，断路器，路由，微代理，事件总线等的集成服务。

Spring Boot 可以离开 Spring Cloud 独立使用，但是 Spring Cloud 离不开 Spring Boot ，属于依赖的关系。

总结：

Spring Boot ，专注于快速，方便的开发单个微服务个体。
Spring Cloud ，关注全局的服务治理框架。

Feign 和 Ribbon 的区别？

Ribbon 和 Feign 都是使用于调用用其余服务的，不过方式不同。
启动类用的注解不同。
Ribbon 使用的是 @RibbonClient 。
Feign 使用的是 @EnableFeignClients 。
服务的指定位置不同。
Ribbon 是在 @RibbonClient 注解上设置。
Feign 则是在定义声明方法的接口中用 @FeignClient 注解上设置。
调使用方式不同。

Ribbon 需要自己构建 Http 请求，模拟 Http 请求而后用 RestTemplate 发送给其余服务，步骤相当繁琐。

Feign 采使用接口的方式，将需要调使用的其余服务的方法定义成声明方法就可，不需要自己构建 Http 请求。不过要注意的是声明方法的注解、方法签名要和提供服务的方法完全一致。

Hystrix 隔离策略？

Hystrix 有两种隔离策略：
线程池隔离
信号量隔离

聊聊 Hystrix 缓存机制？

Hystrix 提供缓存功能，作用是：
减少重复的请求数。
在同一个用户请求的上下文中，相同依赖服务的返回数据始终保持一致。

什么是 Hystrix 断路器？

Hystrix 断路器通过 HystrixCircuitBreaker 实现。
HystrixCircuitBreaker 有三种状态 ：
CLOSED ：关闭
OPEN ：打开
HALF_OPEN ：半开
其中，断路器处于 OPEN 状态时，链路处于非健康状态，命令执行时，直接调用回退逻辑，跳过正常逻辑。

什么是 Hystrix 服务降级？

在 Hystrix 断路器熔断时，可以调用一个降级方法，返回相应的结果。当然，降级方法需要配置和编码，如果胖友不需要，也可以不写，也就是不会有服务降级的功能。

什么是 RPC 远程调用？

RPC 的全称是 Remote Procedure Call 是一种进程间通信方式。
它允许程序调用另一个地址空间（通常是共享网络的另一台机器上）的过程或函数，而不用程序员显式编码这个远程调用的细节。即无论是调用本地接口/服务的还是远程的接口/服务，本质上编写的调用代码基本相同。
比如两台服务器 A，B，一个应用部署在 A 服务器上，想要调用 B 服务器上应用提供的函数或者方法，由于不在一个内存空间，不能直接调用，这时候需要通过就可以应用 RPC 框架的实现来解决

什么是 SOA？与 SOAP 区别是什么？

SOA 是一种面向服务架构，是将相同业务逻辑抽取出来组成单独服务。
SOAP 是 WebService 面向服务协议， 采用 xml,因为比较中，现在不是特别流行。

什么是微服务架构

微服务架构师一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相
协调、互相配合没用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的同学机制互相沟通(通畅采用 Http+restful API)，每个服务都围绕着具体业务进行构建，并且能够被独立的部署到生成环境、类生存环境等。另外，应尽量避免同一的、集中式服务管理机制。

微服务与 SOA 区别

SOA 实现	微服务架构实现
企业级，自顶向下开展实施	团队级，自定向上开展实施
服务由多个子系统组成	一个系统被拆分成多个服务
集成式服务(esb、ws、soap)	集成方式简单(http、rest、json)

RPC 远程调用有哪些框架？

SpringCloud、Dubbo、Dubbox、Hessian、HttpClient、thrift 等。

什么是 SpringCloud

SpringCloud 是微服务的一种解决方案，依赖 SpringBoot 实现。包含注册中心(eureka)、客户端负载均衡(Ribbon)、网关(zull)、分布式锁、分布式会话等。

为什么要使用 SpringCloud

SpringCloud 是一套非常完整的微服务解决方案，俗称"微服务全家桶"，几乎内置了微服务所使用的各种技术，可以不必集成第三方依赖。

SpringCloud 服务注册发现原理

每个 SpringCloud 服务器启动后向注册中心注册本服务器信息，如服务别名、服务器 IP、端口号等，其他服务进行请求时先根据服务别名从注册中心获取到目标服务器 IP 和端口号，并将获取到的信息缓存到本地，然后通过本地使用 HttpClient 等技术进行远程调用。

SpringCloud 支持那些注册中心

Eureka、Consul、Zookeeper

@LoadBalanced 注解的作用

开启客户端负载均衡。

SpringCloud 有几种调用接口方式

使用 Feign 和 RestTemplate

DiscoveryClient 的作用

可以从注册中心中根据服务别名获取注册的服务器信息。

现在 Eureka 闭源了，可以通过什么注册中心替代 Eureka 呢？

Consul 或 Zookeeper

谈谈你对微服务服务治理的思想

Eureka 如何实现高可用

启动多台 Eureka 服务器，然后作为 SpringCloud 服务互相注册，客户端从 Eureka 集群获取信息时，按照注册的 Eureka 顺序对第一个 Eureka 进行访问。

谈谈服务雪崩效应

雪崩效应是在大型互联网项目中，当某个服务发生宕机时，调用这个服务的其他服务也会发生宕机，大型项目的微服务之间的调用是互通的，这样就会将服务的不可用逐步扩大到各个其他服务中，从而使整个项目的服务宕机崩溃.发生雪崩效应的原因有以下几点

1.单个服务的代码存在 bug. 2 请求访问量激增导致服务发生崩溃(如大型商城的枪红包，秒杀功能). 3.服务器的硬件故障也会导致部分服务不可用.

在微服务中，如何保护服务?

一般使用使用 Hystrix 框架，实现服务隔离来避免出现服务的雪崩效应，从而达到保护服务的效果。当微服务中 ,高并发的数据库访问量导致服务线程阻塞，使单个服务宕机，服务的不可用会蔓延到其他服务，引起整体服务灾难性后果，使用服务降级能有效为不同的服务分配资源,一旦服务不可用则返回友好提示，不占用其他服务资源，从而避免单个服务崩溃引发整体服务的不可用.

服务雪崩效应产生的原因

因为 Tomcat 默认情况下只有一个线程池来维护客户端发送的所有的请求，这时候某一接口在某一时刻被大量访问就会占据 tomcat 线程池中的所有线程，其他请求处于等待状态，无法连接到服务接口。

谈谈 Hystrix 服务保护的原理

通过服务降级、服务熔断、服务隔离为高并发服务提供保护。

谈谈服务降级、熔断、服务隔离

服务降级：当客户端请求服务器端的时候，防止客户端一直等待，不会处理业务逻辑代码，直接返回一个友好的提示给客户端。

服务熔断是在服务降级的基础上更直接的一种保护方式，当在一个统计时间范围内的请求失败数量达到设定值（requestVolumeThreshold）或当前的请求错误率达到设定的错误率阈值（errorThresholdPercentage）时开启断路，之后的请求直接走 fallback 方法，在设定时间（sleepWindowInMilliseconds）后尝试恢复。

服务隔离就是 Hystrix 为隔离的服务开启一个独立的线程池，这样在高并发的情况下不会影响其他服务。服务隔离有线程池和信号量两种实现方式，一般使用线程池方式。

服务降级底层是如何实现的？

Hystrix 实现服务降级的功能是通过重写 HystrixCommand 中的 getFallback()方法，当 Hystrix 的 run 方法或 construct 执行发生错误时转而执行 getFallback()方法。

分布式配置中心有那些框架？

Apollo(阿波罗)、zookeeper、springcloud config。

分布式配置中心的作用？

动态变更项目配置信息而不必重新部署项目。

SpringCloud Config 可以实现实时刷新吗？

springcloud config 实时刷新采用 SpringCloud Bus 消息总线。

什么是网关?

网关相当于一个网络服务架构的入口，所有网络请求必须通过网关转发到具体的服务。

网关的作用是什么

统一管理微服务请求，权限控制、负载均衡、路由转发、监控、安全控制黑名单和白名单等

网关与过滤器有什么区别

网关是对所有服务的请求进行分析过滤，过滤器是对单个服务而言。

常用网关框架有那些？

Nginx、Zuul、Gateway

Zuul 与 Nginx 有什么区别？

Zuul 是 java 语言实现的，主要为 java 服务提供网关服务，尤其在微服务架构中可以更加灵活的对网关进行操作。Nginx 是使用 C 语言实现，性能高于 Zuul，但是实现自定义操作需要熟悉 lua 语言，对程序员要求较高，可以使用 Nginx 做 Zuul 集群。

既然 Nginx 可以实现网关？为什么还需要使用 Zuul 框架

Zuul 是 SpringCloud 集成的网关，使用 Java 语言编写，可以对 SpringCloud 架构提供更灵活的服务。

ZuulFilter 常用有那些方法

Run()：过滤器的具体业务逻辑
shouldFilter()：判断过滤器是否有效
filterOrder()：过滤器执行顺序
filterType()：过滤器拦截位置

如何实现动态 Zuul 网关路由转发

通过 path 配置拦截请求，通过 ServiceId 到配置中心获取转发的服务列表，Zuul 内部使用 Ribbon 实现本地负载均衡和转发。

Zuul 网关如何搭建集群

使用 Nginx 的 upstream 设置 Zuul 服务集群，通过 location 拦截请求并转发到 upstream，默认使用轮询机制对 Zuul 集群发送请求。

Http 协议同步接口调用失败了怎么做？

采用消息补偿机制重新发送请求

高并发解决方案

高性能 Nginx+Lua

什么是 DNS 解析域名

DNS 域名解析就是讲域名转化为不需要显示端口（二级域名的端口一般为 80）的 IP 地址，域名解析的一般先去本地环境的 host 文件读取配置，解析成对应的 IP 地址，根据 IP 地址访问对应的服务器。若 host 文件未配置，则会去网络运营商获取对应的 IP 地址和域名.

你用过那些外网映射工具

花生壳,natapp,ngrok。

动态网站与静态网站区别

在浏览器中打开一个网站，点击鼠标右键查看源码，多次请求后如果源码不产生变化就是静态网站，变化就是动态网站。

动态页面静态化的作用

便于搜索引擎抓取和排名

什么是动静分离架构模式

静态页面与动态页面分开不同系统访问的架构设计方法，静态页面与动态页面以不同域名区分。

如何搭建动静分离

以 nginx 服务器作为静态资源服务器，静态资源和动态资源访问分开配置，静态资源在 location 中使用本地文件路径配置方式，动态资源使用 proxy_pass 配置到后台服务器。

如：

```
###静态资源访问
server {
   listen          80;
   server_name    static.itmayiedu.com;
   location /static/imgs {
```

```
            root F:/;
            index    index.html index.htm;
        }
    }
###动态资源访问
    server {
        listen          80;
        server_name    www.itmayiedu.com;

        location / {
            proxy_pass http://127.0.0.1:8080;
            index    index.html index.htm;
        }
    }
```

动静分离与前后分离区别

动静分离是将静态资源和动态资源存放在不同服务器中，前后分离是将前端和后台分离，前端通过 api 调用后台接口

如何控制浏览器静态资源缓存

静态资源存在缓存的原因是项目上线时，浏览器缓存中的静态资源导致与服务器将淘汰资源的代码发生冲突(或者是页面访问频繁访问同一资源，导致一些浏览器如 IE(本人开发亲身经历过)返回默认的响应结果，与实际响应结果不符合)，

一般的服务器是强制 F5 进行刷新或者是清除缓存，最有效的解决方法就是在请求资源后面加上变量(如时间戳,随机数)

Http 状态码 304 的作用

表示浏览器存在静态资源缓存就不从服务器获取静态资源

高并发服务限流特级

高并发服务限流特技有哪些算法?

传统计算器算法，滑动窗口计数器算法,令牌桶算法和漏桶算法。

传统计数器限流算法有什么弊端?

传统计数器限流方式不支持高并发 ,存在线程安全问题.若大量访问请求集中在计数器最后时刻，计数器极易发生临界问题，访问的请求无法完成.

什么是滑动窗口计数器?

滑动窗口计数器是一种服务限流的算法,相对于计数器方法的实现 ,滑动窗口实现会更加平滑，并自动消除毛刺。其原理是当有访问进来时，会判断若干个单位来的请求是否超过

设置的阀值，并对当前时间片的请求数+1.

令牌桶算法的原理?

向一个存放固定容量令牌的同，以固定速率往桶里添加令牌，当桶已经装满时，新增的令牌会被丢弃或者拒绝,当一个固定数目的数据包到达时，会在

桶中删除同等数量的令牌，数据包会发到网络上，当这个固定数目超过桶中的令牌数，不会删除桶中的令牌数目，则该数据包会被限流(丢弃或者存入缓冲区等待)

漏桶算法的原理?

向一个存放固定容量的桶，以任意速率滴入水滴(请求)，以固定速率滴出水滴，当滴入水滴量超过桶中设置固定容量，则会发生溢出，溢出的水滴的请求是无法访问的，直接走

服务限流降级，桶中的容量不发生任何变化。

令牌桶与漏桶算法的区别?

令牌桶和漏桶算法的区别是令牌桶会根据请求的令牌数与桶中的令牌数做对比，倘若桶中令牌数小于请求令牌数则多余的令牌数的请求被拒绝。漏桶算法则是向桶中添加请求，当

请求数大于桶中容量发生溢出，溢出的请求直接被拒绝访问。主要区别是漏桶算法是强行限制数据的传输速率，而令牌桶在能够限制数据的平均传输速率外，还允许某种程度的突发传输，使用于抢红包等高并发的场景。

Web 前端有哪些优化方案

1.网站框架实行动静分离，动态资源和和静态资源部署到不同的服务器上面，使用 nginx 访问本地静态资源，通过 nginx 代理转发到 tomcate 访问动态资源

2.在访问静态资源时在 Url 后缀加上时间戳，防止访问资源的与浏览器本地缓存资源存在冲突.

3.页面减少 HTTP 请求，合并静态资源(如 js 或者 css ）并进行压缩。

4.使用 CDN 内容分发，缓存静态资源，让用户访问最近的服务器，减少宽带之间的传输.

什么是 CDN 内容分发?

CDN 内容转发是指在用户和服务器之间加上一个缓存机制，一组分布在不同的静态资源服务器，储存静态资源，动态获取用户 IP，并让用户访问最近的静态资源服务器,防止出现网络延迟阻塞，

提高访问速度。CND 服务器部署在网络运营商的机房，用户请求首先会访问 CND 服务器，如果 CND 服务器中没有缓存会自动把创建缓存，当用户再次访问时，直接获取缓存资源，并返回给前端，提高静态

资源的访问速度。

CDN 内容加速原理?

1)用户向浏览器提供要访问的域名.

2)浏览器从本地 host 文件中解析域名,如何 host 文件没有做任何配置，则浏览器调用域名解析库对域名进行解析,析函数库一般得到的是该域名对应的 CNAME 记录,从中获取真正的 IP 地址,此过程中,根据地理位置信息解析对应的 IP 地址，使得用户能就近访问静态资源服务器。

3)此次解析的 CDN 服务器的 IP 地址，浏览器根据这个地址，向缓存服务器发出请求。

4)缓存服务器根究浏览器访问的域名 ,通过缓存内部获得此域名的真实 IP 地址，再有缓存服务器提交该 IP 地址的访问请求。

5)缓存服务器从服务器的 ip 地址获取内容备用,并将数据返回给用户，完成服务过程.

6)客户端将服务器返回的数据展示给前端

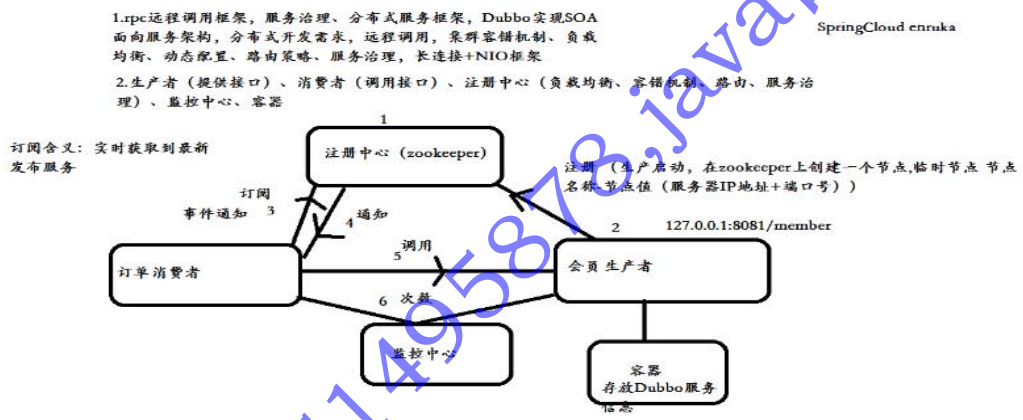
什么是 Dubbo？

Duubbo 是一个 RPC 远程调用框架， 分布式服务治理框架

什么是 Dubbo 服务治理？

服务与服务之间会有很多个 Url、依赖关系、负载均衡、容错、自动注册服务。

Dubbo 原理



Dubbo 有几种配置方式？

XML 配置

注解配置

属性配置

Java API 配置

Dubbo 调用是同步的吗？

默认情况下，调用是同步的方式。

Dubbo 可以对调用结果进行缓存吗?

Dubbo 通过 CacheFilter 过滤器，提供结果缓存的功能，且既可以适用于 Consumer 也可以适用于 Provider 。

通过结果缓存，用于加速热门数据的访问速度，Dubbo 提供声明式缓存，以减少用户加缓存的工作量。

Dubbo 目前提供三种实现：

lru ：基于最近最少使用原则删除多余缓存，保持最热的数据被缓存。

threadlocal ：当前线程缓存，比如一个页面渲染，用到很多 portal，每个 portal 都要去查用户信息，通过线程缓存，可以减少这种多余访问。

jcache ：与 JSR107 集成，可以桥接各种缓存实现。

注册中心挂了还可以通信吗？

可以。对于正在运行的 Consumer 调用 Provider 是不需要经过注册中心，所以不受影响。并且，Consumer 进程中，内存已经缓存了 Provider 列表。

那么，此时 Provider 如果下线呢？如果 Provider 是正常关闭，它会主动且直接对和其处于连接中的 Consumer 们，发送一条“我要关闭”了的消息。那么，Consumer 们就不会调用该 Provider ，而调用其它的 Provider 。

另外，因为 Consumer 也会持久化 Provider 列表到本地文件。所以，此处如果 Consumer 重启，依然能够通过本地缓存的文件，获得到 Provider 列表。

再另外，一般情况下，注册中心是一个集群，如果一个节点挂了，Dubbo Consumer 和 Provider 将自动切换到集群的另外一个节点上。

Dubbo Provider 异步关闭时，如何从注册中心下线？

① Zookeeper 注册中心的情况下

服务提供者，注册到 Zookeeper 上时，创建的是 EPHEMERAL 临时节点。所以在服务提供者异常关闭时，等待 Zookeeper 会话超时，那么该临时节点就会自动删除。

② Redis 注册中心的情况下

使用 Redis 作为注册中心，是有点小众的选择，我们就不在本文详细说了。感兴趣的胖友，可以看看《精尽 Dubbo 源码分析 —— 注册中心（三）之 Redis》一文。总的来说，实现上，还是蛮有趣的。因为，需要通知到消费者，服务列表发生变化，所以就无法使用 Redis Key 自动过期。所以... 还是看文章吧。哈哈哈哈。

Dubbo Consumer 只能调用从注册中心获取的 Provider 么？

不是，Consumer 可以强制直连 Provider 。

在开发及测试环境下，经常需要绕过注册中心，只测试指定服务提供者，这时候可能需要点对点直连，点对点直连方式，将以服务接口为单位，忽略注册中心的提供者列表，A 接口配置点对点，不影响 B 接口从注册中心获取列表。

Dubbo 支持哪些通信协议？

对应【protocol 远程调用层】。

Dubbo 目前支持如下 9 种通信协议：

【重要】dubbo:// ，默认协议。参见《Dubbo 用户指南 —— dubbo://》。

【重要】rest:// ，贡献自 Dubbox ，目前最合适的 HTTP Restful API 协议。参见《Dubbo 用户指南 —— rest://》。

rmi:// ，参见《Dubbo 用户指南 —— rmi://》。

webservice:// ，参见《Dubbo 用户指南 —— webservice://》。

hessian:// ，参见《Dubbo 用户指南 —— hessian://》。

thrift:// ，参见《Dubbo 用户指南 —— thrift://》。
memcached:// ，参见《Dubbo 用户指南 —— memcached://》。
redis:// ，参见《Dubbo 用户指南 —— redis://》。
http:// ，参见《Dubbo 用户指南 —— http://》。注意，这个和我们理解的 HTTP 协议有差异，而是 Spring 的 HttpInvoker 实现。

Dubbo 使用什么通信框架？

对应【transport 网络传输层】。
在通信框架的选择上，强大的技术社区有非常多的选择，如下列表：
Netty3
Netty4
Mina
Grizzly

Dubbo 支持哪些序列化方式？

对应【serialize 数据序列化层】。
Dubbo 目前支付如下 7 种序列化方式：
【重要】Hessian2 ：基于 Hessian 实现的序列化拓展。dubbo:// 协议的默认序列化方案。
Hessian 除了是 Web 服务，也提供了其序列化实现，因此 Dubbo 基于它实现了序列化拓展。
另外，Dubbo 维护了自己的 hessian-lite ，对 Hessian 2 的 序列化 部分的精简、改进、BugFix 。
Dubbo ：Dubbo 自己实现的序列化拓展。
具体可参见《精尽 Dubbo 源码分析 —— 序列化（二）之 Dubbo 实现》。
Kryo ：基于 Kryo 实现的序列化拓展。
具体可参见《Dubbo 用户指南 —— Kryo 序列化》
FST ：基于 FST 实现的序列化拓展。
具体可参见《Dubbo 用户指南 —— FST 序列化》
JSON ：基于 Fastjson 实现的序列化拓展。
NativeJava ：基于 Java 原生的序列化拓展。
CompactedJava ：在 NativeJava 的基础上，实现了对 ClassDescriptor 的处理。

Dubbo 有哪些负载均衡策略？

调用策略。
Random LoadBalance
随机，按权重设置随机概率。
RoundRobin LoadBalance
轮询，按公约后的权重设置轮询比率。
LeastActive LoadBalance
最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。
使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。
这个就是自动感知一下，如果某个机器性能越差，那么接收的请求越少，越不活跃，此时就会给不活跃的性能差的机器更少的请求。
ConsistentHash LoadBalance

一致性 Hash，相同参数的请求总是发到同一提供者。

当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。

Dubbo 服务如何监控和管理？

一旦使用 Dubbo 做了服务化后，必须必须必须要做的服务治理，也就是说，要做服务的管理与监控。当然，还有服务的降级和限流。这块，放在下面的面试题，在详细解析。

Dubbo 管理平台 + 监控平台

dubbo-monitor 监控平台，基于 Dubbo 的【monitor 监控层】，实现相应的监控数据的收集到监控平台。

dubbo-admin 管理平台，基于注册中心，可以获取到服务相关的信息。

Dubbo 服务如何做降级？

比如说服务 A 调用服务 B，结果服务 B 挂掉了。服务 A 再重试几次调用服务 B，还是不行，那么直接降级，走一个备用的逻辑，给用户返回响应。

在 Dubbo 中，实现服务降级的功能，一共有两大种方式。

① Dubbo 原生自带的服务降级功能

当然，这个功能，并不能实现现代微服务的熔断器的功能。所以一般情况下，不太推荐这种方式，而是采用第二种方式。

② 引入支持服务降级的组件

目前开源社区常用的有两种组件支持服务降级的功能，分别是：

Alibaba Sentinel

Netflix Hystrix

因为目前 Hystrix 已经停止维护，并且和 Dubbo 的集成度不是特别高，需要做二次开发，所以推荐使用 Sentinel 。

Dubbo 支持哪些注册中心？

Dubbo 支持多种主流注册中心，如下：

Redis ，参见《用户指南 —— Redis 注册中心》。

Multicast 注册中心，参见《用户指南 —— Multicast 注册中心》。

Simple 注册中心，参见《用户指南 —— Simple 注册中心》。

Dubbo 需要 Web 容器吗？

这个问题，仔细回答，需要思考 Web 容器的定义。然而实际上，真正想问的是，Dubbo 服务启动是否需要启动类似 Tomcat、Jetty 等服务器。

这个答案可以是，也可以是不是。为什么呢？根据协议的不同，Provider 会启动不同的服务器。

在使用 dubbo:// 协议时，答案是否，因为 Provider 启动 Netty、Mina 等 NIO Server 。

在使用 rest:// 协议时，答案是是，Provider 启动 Tomcat、Jetty 等 HTTP 服务器，或者也可以使用 Netty 封装的 HTTP 服务器。

在使用 hessian:// 协议时，答案是是，Provider 启动 Jetty、Tomcat 等 HTTP 服务器。

Dubbo 有哪些协议？

默认用的 dubbo 协议、Http、RMI、Hessian

dubbo 支持的序列化协议

dubbo 实际基于不同的通信协议 ,支持 hessian、java 二进制序列化、json、SOAP 文本序列化多种序列化协议。

但是 hessian 是其默认的序列化协议。

Dubbo 整个架构流程

分为四大模块

生产者、消费者、注册中心、监控中心

生产者：提供服务

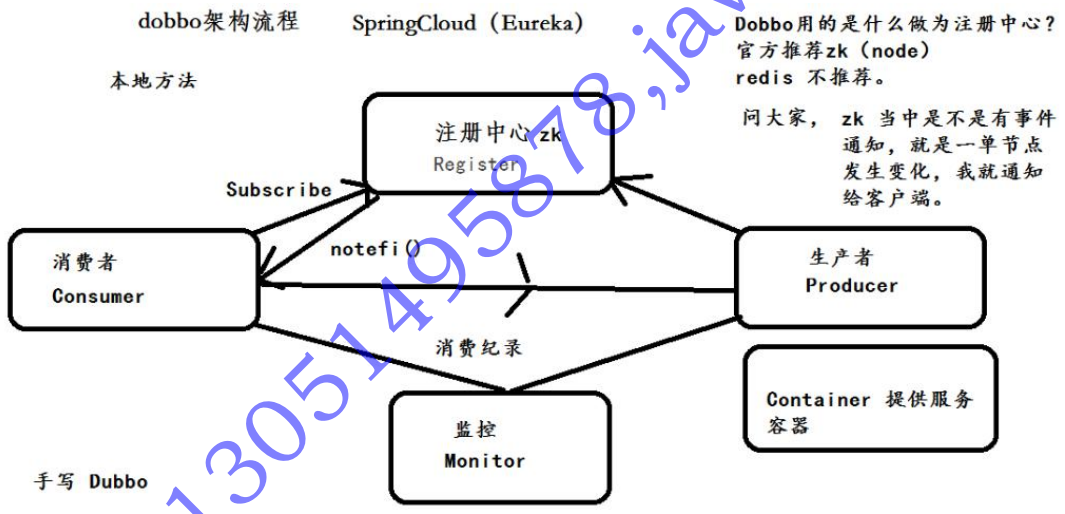
消费者: 调用服务

注册中心:注册信息(redis、zk)

监控中心:调用次数、关系依赖等。

首先生产者将服务注册到注册中心（zk），使用 zk 持久节点进行存储，消费订阅 zk 节点，一旦有节点变更，zk 通过事件通知传递给消费者，消费可以调用生产者服务。

服务与服务之间进行调用，都会在监控中心中，存储一个记录。



Dubbox 与 Dubbo 区别？

Dubox 使用 http 协议+rest 风格传入 json 或者 xml 格式进行远程调用。

Dubbo 使用 Dubbo 协议。

注册中心挂了可以继续通信吗?

可以，因为刚开始初始化的时候，消费者会将提供者的地址等信息拉取到本地缓存，所以注册中心挂了可以继续通信。

SpringCloud 与 Dubbo 区别？

相同点:

dubbo 与 springcloud 都可以实现 RPC 远程调用。

dubbo 与 springcloud 都可以使用分布式、微服务场景下。

区别:
dubbo 有比较强的背景,在国内有一定影响力。
dubbo 使用 zk 或 redis 作为作为注册中心
springcloud 使用 eureka 作为注册中心
dubbo 支持多种协议，默认使用 dubbo 协议。
Springcloud 只能支持 http 协议。
Springcloud 是一套完整的微服务解决方案。
Dubbo 目前已经停止更新,SpringCloud 更新速度快。

什么是 Zookeeper

Zookeeper 是一个工具，可以实现集群中的分布式协调服务。
所谓的分布式协调服务，就是在集群的节点中进行可靠的消息传递，来协调集群的工作。
Zookeeper 之所以能够实现分布式协调服务，靠的就是它能够保证分布式数据一致性。
所谓的分布式数据一致性，指的就是可以在集群中保证数据传递的一致性。
Zookeeper 能够提供的分布式协调服务包括：数据发布订阅、负载均衡、命名服务、分布式协调/通知、集群管理、分布式锁、分布式队列等功能

分布式协调工具

什么是 ZooKeeper

ZooKeeper 是 Java 语言编写的开源框架，用以协调分布式的一个工具。

ZooKeeper 存储结构与特性

类似于树形结构，同一层节点名称不能重复。节点类型分为临时节点与持久节点
Zookeeper 以节点方式进行存储，类似于 xml 树状结构；
节点又分为节点名称（全路径不能重复）和 节点值
节点类型有持久节点（持久化在硬盘上）和临时节点（会话与临时节点同死同生）
b、节点功能：每个节点都有通知功能，当这个节点增删改的时候都会有事件通知
Zookeeper 主要有一下特性
a、一致性：数据按照顺序分批入库；
b、原子性：事务要么成功要么失败，不会局部化；
c、单一视图：客户端连接集群中任何一个 zk 节点，数据都是一致的；
d、可靠性：每次对 zk 的操作状态都会保存在服务端；
e、实时性：客户端可以读取到 zk 服务器的 新数据；

ZooKeeper 中临时节点与持久节点区别

持久节点是持久化在硬盘上，会话断开后节点也能查到；
临时节点与会话保持连接，会话在节点在，会话断开，节点也会删除；

ZooKeeper 应用场景

服务注册与发现的中心
B、利用临时节点特性解决分布式锁
C、分布式配置中心
D、基于哨兵机制实现选举策略
E、实现本地负载均衡
F、基于节点事件通知特性可做消息中间件
G、分布式事务

什么场景下会导致 ZooKeeper 发生延迟通知

watch 事件延迟：节点被修改后，会有事件通知发往观察者，直到接收到 watch 事件 ,观察者才会知道节点被修改了 ;当管擦着接到 watch 事件的那一刻，该节点又被其他修改者修改了，而 近的 watch 事件还没有通知到观察者，就会造成延迟通知。

分布式锁有那些实现方案

a.基于 setNx (SET IF NOT EXISTS)实现分布式锁（麻烦，需要考虑死锁及释放问题）
b、redission 实现分布式锁
c、zookeeper 实现分布式锁（基于临时节点，实现简单，效率高，失效时间容易控制）

ZooKeeper 实现分布式锁的原理

多个 jvm 在同一个 zookeeper 上创建同一个节点（临时节点），哪个 jvm 能创建成功，就表示它拿到了锁，剩下的 jvm 保持对这个节点的监听，一旦发现这个节点被删除了，那么剩下的 jvm 就重新再创建这个节点，谁能创建成功谁能拿到锁，依次循环下去。

ZooKeeper 实现分布式锁与 Redis 实现分布式锁区别

Zookeeper 通过创建临时节点和利用监听事件实现分布式锁 ,Redis 使用 setnx 命令创建相同的 key，因为 Redis 的 key 保证唯一，先创建的先获取锁。
不断的去尝试，去获取锁，比较耗性能
Zookeeper 实现分布式锁，即使获取不到锁，创建对锁的监听即可，不需要不断去尝试获取 锁，性能开销小
Redis 实现分布式锁，如果客户端获取到锁的时候遇到 bug 或挂了，还需要等到超时时间过了以后才能重新获取锁
Zookeeper 实现分布式锁，创建的是临时节点，客户端挂了，节点自然删除，也就达到了自动释放锁的效果

使用 Zookeeper 实现服务 Master 选举原理

多个服务器在启动时候，会在 Zookeeper 上创建相同的临时节点，谁如果能够创建成功，谁就为主。如果主服务器宕机，其他备用节点获取监听信息，

重新创建节点，选出主服务器。

ZooKeeper 集群选举原理

每台 Zookeeper 服务器启动时会发起投票，每次投票后，服务器统计投票信息，如果有机器获取半数以上的投票数则 leader 产生。

分布式 Session 一致性问题

a、使用 Nginx 反向代理，即 IP 绑定，同一个 ip 只能在同一个机器上访问
b、使用数据库，但性能不高
c、tomcat 内置了对 session 同步的支持，但可能会产生延迟
d、使用 Spring-Session 框架，相当于把 session 放到 redis 中
e、使用 token 令牌代替 session

分布式锁解决方案

分布式事务解决方案

分布式任务调度平台解决方案

分布式事务解决方案

分布式日志收集解决方案

分布式全局 id 生成方案

分布式服务追踪与调用链系统解

分布式消息系统与中间件

消息中间件产生的背景

传统的 Web 项目采用 http 协议基于请求和响应传输信息，请求发出后必须等待服务器端响应，如果服务器端不会及时响应客户端会一直等待。

Zookeeper 特点

Zookeeper 工作在集群中，对集群提供分布式协调服务，它提供的分布式协调服务具有如下的特点：

顺序一致性

从同一个客户端发起的事务请求，最终将会严格按照其发起顺序被应用到 zookeeper 中

原子性

所有事物请求的处理结果在整个集群中所有机器上的应用情况是一致的，

即，要么整个集群中所有机器都成功应用了某一事务，要么都没有应用，一定不会出现集群中部分机器应用了改事务，另外一部分没有应用的情况。

单一视图

无论客户端连接的是哪个 zookeeper 服务器，其看到的服务端数据模型都是一致的。

可靠性

一旦服务端成功的应用了一个事务，并完成对客户端的响应，那么该事务所引起的服务端状态变更将会一直保留下来，除非有另一个事务又对其进行了改变。

实时性

zookeeper 并不是一种强一致性，只能保证顺序一致性和最终一致性，只能称为达到了伪实时性。

zookeeper 的数据模型

zookeepei 中可以保存数据，正是利用 zookeeper 可以保存数据这一特点，我们的集群通过在 zookeeper 里存取数据来进行消息的传递。

zookeeper 中保存数据的结构非常类似于文件系统。都是由节点组成的树形结构。不同的是文件系统是由文件夹和文件来组成的树，而 zookeeper 中是由 ZNODE 来组成的树。

每一个 ZNODE 里都可以存放一段数据，ZNODE 下还可以挂载零个或多个子 ZNODE 节点，从而组成一个树形结构。

Zookeeper 应用场景

数据发布订阅
负载均衡
命名服务
分布式协调
集群管理
配置管理
分布式队列
分布式锁

什么是分布式锁

简单的理解就是：分布式锁是一个在很多环境中非常有用的原语，它是不同的系统或是同一个系统的不同主机之间互斥操作共享资源的有效方法。

Zookeeper 实现分布式锁

分布式锁使用 zk，在 zk 上创建一个临时节点，使用临时节点作为锁，因为节点不允许重复。

如果能创建节点成功，生成订单号，如果创建节点失败，就等待。临时节点 zk 关闭，释放锁，其他节点就可以重新生成订单号。

Redis 分布式锁思考

一般的锁只能针对单机下同一进程的多个线程，或单机的多个进程。多机情

况下，对同一个资源访问，需要对每台机器的访问进程或线程加锁，这便是分布式锁。分布式锁可以利用多机的共享缓存（例如 redis）实现。redis 的命令文档[1]，实现及分析参考文档[2]。

利用 redis 的 get、setnx、getset、del 四个命令可以实现基于 redis 的分布式锁：

get key：表示从 redis 中读取一个 key 的 value，如果 key 没有对应的 value，返回 nil，如果存储的值不是字符串类型，返回错误

setnx key value：表示往 redis 中存储一个键值对，但只有当 key 不存在时才成功，返回 1；否则失败，返回 0，不改变 key 的 value

getset key：将给定 key 的值设为 value ，并返回 key 的旧值(old value)。当旧值不存在时返回 nil，当旧值不为字符串类型，返回错误

del key：表示删除 key，当 key 不存在时不做操作，返回删除的 key 数量

关于加锁思考，循环中：

0、setnx 的 value 是当前机器时间+预估运算时间作为锁的失效时间。这是为了防止获得锁的线程挂掉而无法释放锁而导致死锁。

0.1、返回 1，证明已经获得锁，返回啦

0.2、返回 0，获得锁失败，需要检查锁超时时间

1、get 获取到锁，利用失效时间判断锁是否失效。

1.1、取锁超时时间的时刻可能锁被删除释放，此时并没有拿到锁，应该重新循环加锁逻辑。

2、取锁超时时间成功

2.1、锁没有超时，休眠一下，重新循环加锁

2.2、锁超时，但此时不能直接释放锁删除。因为此时可能多个线程都读到该锁超时，如果直接删除锁，所有线程都可能删除上一个删除锁又新上的锁，会有多个线程进入临界区，产生竞争状态。

3、此时采用乐观锁的思想，用 getset 再次获取锁的旧超时时间。

3.1、如果此时获得锁旧超时时间成功

3.1.1、等于上一次获得的锁超时时间，证明两次操作过程中没有别人动过这个锁，此时已经获得锁

3.1.2、不等于上一次获得的锁超时时间，说明有人先动过锁，获取锁失败。虽然修改了别人的过期时间，但因为冲突的线程相差时间极短，所以修改后的过期时间并无大碍。此处依赖所有机器的时间一致。

3.2、如果此时获得锁旧超时时间失败，证明当前线程是第一个在锁失效后又加上锁的线程，所以也获得锁

4、其他情况都没有获得锁，循环 setnx 吧

关于解锁的思考：

在锁的时候，如果锁住了，回传超时时间，作为解锁时候的凭证，解锁时传入锁的键值和凭证。我思考的解锁时候有两种写法：

1、解锁前 get 一下键值的 value，判断是不是和自己的凭证一样。但这样存在一些问题：

get 时返回 nil 的可能，此时表示有别的线程拿到锁并用完释放

get 返回非 nil，但是不等于自身凭证。由于有 getset 那一步，当两个竞争线程都在这个过程中时，存在持有锁的线程凭证不等于 value，而是 value 是稍慢那一步线程设置的 value。

2、解锁前用凭证判断锁是否已经超时，如果没有超时，直接删除；如果超时，等着锁自动过期就好，免得误删别人的锁。但这种写法同样存在问题，由于

线程调度的不确定性，判断到删除之间可能过去很久，并不是绝对意义上的正确解锁。

```
public class RedisLock {

    private static final Logger logger = LoggerFactory.getLogger(RedisLock.class);

    //显然jedis还需要自己配置来初始化
    private Jedis jedis = new Jedis();

    //默认锁住15秒，尽力规避锁时间太短导致的错误释放
    private static final long DEFAULT_LOCK_TIME = 15 * 1000;

    //尝试锁住一个lock，设置尝试锁住的次数和超时时间（毫秒）,默认最短15秒
    //成功时返回这把锁的key，解锁时需要凭借锁的lock和key
    //失败时返回空字符串
    public String lock(String lock, int retryCount, long timeout) {
        Preconditions.checkArgument(retryCount > 0 && timeout > 0, "retry count <= 0 or timeout <= 0 !");
        Preconditions.checkArgument(retryCount < Integer.MAX_VALUE && timeout < Long.MAX_VALUE -
DEFAULT_LOCK_TIME,
                "retry count is too big or timeout is too big!");
        String $lock = Preconditions.checkNotNull(lock) + "_redis_lock";
        long $timeout = timeout + DEFAULT_LOCK_TIME;
        String ret = null;
        //重试一定次数，还是拿不到，就放弃
        try {
            long i, status;
            for (i = 0, status = 0; status == 0 && i < retryCount; ++i) {
                //尝试加锁，并设置超时时间为当前机器时间+超时时间
                if ((status = jedis.setnx($lock, ret = Long.toString(System.currentTimeMillis() +
$timeout))) == 0) {
                    //获取锁失败，查看锁是否超时
                    String time = jedis.get($lock);
                    //在加锁和检查之间，锁被删除了，尝试重新加锁
                    if (time == null) {
                        continue;
                    }
                    //锁的超时时间戳小于当前时间，证明锁已经超时
                    if (Long.parseLong(time) < System.currentTimeMillis()) {
                        String oldTime = jedis.getSet($lock, Long.toString(System.currentTimeMillis()
+ $timeout));
                        if (oldTime == null || oldTime.equals(time)) {
                            //拿到锁了，跳出循环
                            break;
                        }
                    }
                    try {
                        TimeUnit.MILLISECONDS.sleep(1L);
                    } catch (InterruptedException e) {
                        logger.error("lock key:{} sleep failed!", lock);
                    }
                }
            }
            if (i == retryCount && status == 0) {
                logger.info("lock key:{} failed!", lock);
                return "";
            }
            //给锁加上过期时间
            jedis.pexpire($lock, $timeout);
            logger.info("lock key:{} succsee!", lock);
            return ret;
        } catch (Exception e) {
            logger.error("redis lock key:{} failed! cached exception: ", lock, e);
            return "";
        }
    }

    //释放lock的锁，需要传入lock和key
```

```
    //尽力确保删除属于自己的锁，但是不保证做得到
    public void releaseLock(String lock, String key) {
        String $lock = Preconditions.checkNotNull(lock) + "_redis_lock";
        Preconditions.checkNotNull(key);
        try {
            long timeout = Long.parseLong(key);
            //锁还没有超时，锁还属于自己可以直接删除
            //但由于线程运行的不确定性，其实不能完全保证删除时锁还属于自己
            //真正执行删除操作时，距离上语句判断可能过了很久
            if (timeout <= System.currentTimeMillis()) {
                jedis.del($lock);
                logger.info("release lock:{} with key:{} success!", lock, key);
            } else {
                logger.info("lock:{} with key:{} timeout! wait to expire", lock, key);
            }
        } catch (Exception e) {
            logger.error("redis release {}  with key:{} failed! cached exception: ", lock, key, e);
        }
    }
}
```

Zookeeper 与 Redis 实现分布式锁的区别

基于缓存实现分布式锁

锁没有失效事件,容易死锁
非阻塞式
不可重入

基于 Zookeeper 实现分布式锁

实现相对简单
可靠性高
性能较好

Java 实现定时任务有哪些方式

Thread

```
public class Demo01 {
     static long count = 0;
     public static void main(String[] args) {
          Runnable runnable = new Runnable() {
               @Override
               public void run() {
                    while (true) {
                         try {
                              Thread.sleep(1000);
                              count++;
                              System.out.println(count);
                         } catch (Exception e) {
                              // TODO: handle exception
                         }
                    }
               }
          };
          Thread thread = new Thread(runnable);
          thread.start();
     }
}
```

TimerTask

```
/**
 * 使用TimerTask类实现定时任务
*/
public class Demo02 {
     static long count = 0;

     public static void main(String[] args) {
          TimerTask timerTask = new TimerTask() {

               @Override
               public void run() {
                    count++;
                    System.out.println(count);
               }
          };
          Timer timer = new Timer();
          // 天数
          long delay = 0;
          // 秒数
          long period = 1000;
          timer.scheduleAtFixedRate(timerTask, delay, period);
     }

}
```

ScheduledExecutorService

使用 ScheduledExecutorService 是从 Java JavaSE5 的 java.util.concurrent 里，做为并发工具类被引进的，这是最理想的定时任务实现方式。

```
public class Demo003 {
     public static void main(String[] args) {
          Runnable runnable = new Runnable() {
               public void run() {
                    // task to run goes here
                    System.out.println("Hello !!");
               }
          };
          ScheduledExecutorService service = Executors.newSingleThreadScheduledExecutor();
          // 第二个参数为首次执行的延时时间，第三个参数为定时执行的间隔时间
          service.scheduleAtFixedRate(runnable, 1, 1, TimeUnit.SECONDS);
     }
}
```

Quartz

创建一个 quartz_demo 项目

引入 maven 依赖

```
     <dependencies>
          <!-- quartz -->
          <dependency>
               <groupId>org.quartz-scheduler</groupId>
               <artifactId>quartz</artifactId>
```

```
            <version>2.2.1</version>
        </dependency>
        <dependency>
            <groupId>org.quartz-scheduler</groupId>
            <artifactId>quartz-jobs</artifactId>
            <version>2.2.1</version>
        </dependency>
    </dependencies>
```

任务调度类

```
public class MyJob implements Job {

    public void execute(JobExecutionContext context) throws JobExecutionException {
        System.out.println("quartz MyJob date:" + new Date().getTime());
    }

}
```

启动类

```
  //1.创建Scheduler的工厂
    SchedulerFactory sf = new StdSchedulerFactory();
    //2.从工厂中获取调度器实例
    Scheduler scheduler = sf.getScheduler();


    //3.创建JobDetail
    JobDetail jb = JobBuilder.newJob(MyJob.class)
           .withDescription("this is a ram job") //job的描述
           .withIdentity("ramJob", "ramGroup") //job 的name和group
           .build();

    //任务运行的时间，SimpleSchedle类型触发器有效
    long time=  System.currentTimeMillis() + 3*1000L; //3秒后启动任务
    Date statTime = new Date(time);

    //4.创建Trigger
       //使用SimpleScheduleBuilder或者CronScheduleBuilder
    Trigger t = TriggerBuilder.newTrigger()
              .withDescription("")
              .withIdentity("ramTrigger", "ramTriggerGroup")
              //.withSchedule(SimpleScheduleBuilder.simpleSchedule())
              .startAt(statTime)  //默认当前时间启动
              .withSchedule(CronScheduleBuilder.cronSchedule("0/2 * * * * ?")) //两秒执行一次
              .build();

    //5.注册任务和定时器
    scheduler.scheduleJob(jb, t);

    //6.启动 调度器
    scheduler.start();
```


分布式情况下定时任务会出现哪些问题？

分布式集群的情况下，怎么保证定时任务不被重复执行

分布式定时任务解决方案

①使用 zookeeper 实现分布式锁 缺点(需要创建临时节点、和事件通知不易于扩展)

②使用配置文件做一个开关 缺点发布后，需要重启

③数据库唯一约束，缺点效率低
④使用分布式任务调度平台
XXLJOB

分布式事物解决方案

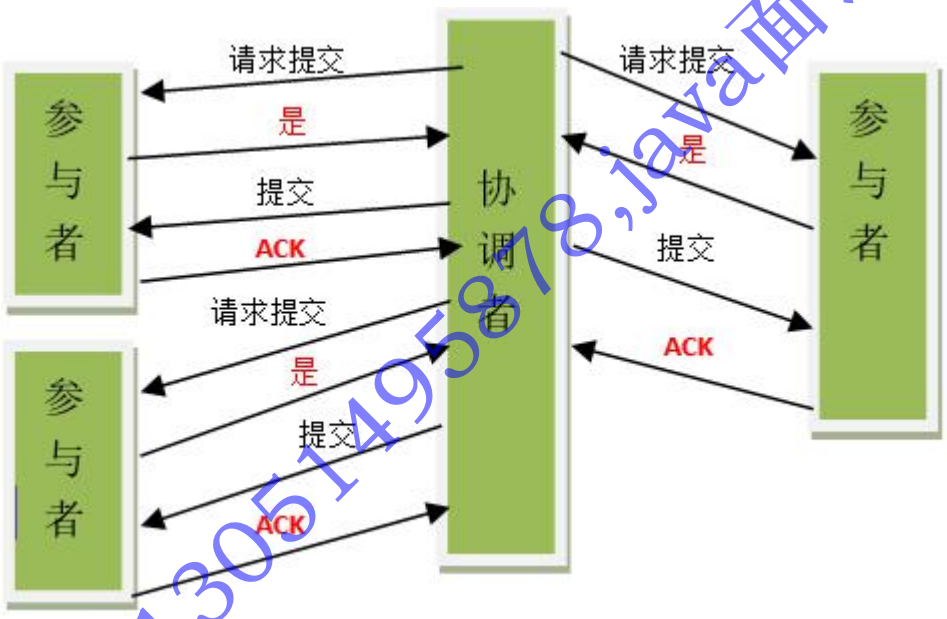
全局事物

使用全局事物两段提交协议，遵循 XA 协议规范，使用开源框架 jta+automatic 。

什么是两段提交协议：在第一阶段，所有参与全局事物的节点都开始准备，告诉事物管理器

它们准备好了，在是第二阶段，事物管理器告诉资源执行 ROLLBACK 还是 Commit，只要任何一方为 ROLLBACK ，则直接回滚。

参考案例：springboot 集成 automatic+jta



本地消息表

这种实现方式的思路，其实是源于 ebay，后来通过支付宝等公司的布道，在业内广泛使用。其基本的设计思想是将远程分布式事务拆分成一系列的本地事务。如果不考虑性能及设计优雅，借助关系型数据库中的表即可实现。

举个经典的跨行转账的例子来描述。

第一步，伪代码如下，扣款 1W，通过本地事务保证了凭证消息插入到消息表中。

第二步，通知对方银行账户上加 1W 了。那问题来了，如何通知到对方呢？

通常采用两种方式：

1. 采用时效性高的 MQ，由对方订阅消息并监听，有消息时自动触发事件
2. 采用定时轮询扫描的方式，去检查消息表的数据。

两种方式其实各有利弊，仅仅依靠 MQ，可能会出现通知失败的问题。而过于频繁的定时轮询，效率也不是最佳的（90%是无用功）。所以，我们一般会把

两种方式结合起来使用。

解决了通知的问题，又有新的问题了。万一这消息有重复被消费，往用户帐号上多加了钱，那岂不是后果很严重？

仔细思考，其实我们可以消息消费方，也通过一个“消费状态表”来记录消费状态。在执行“加款”操作之前，检测下该消息（提供标识）是否已经消费过，消费完成后，通过本地事务控制来更新这个“消费状态表”。这样子就避免重复消费的问题。

总结：上诉的方式是一种非常经典的实现，基本避免了分布式事务，实现了“最终一致性”。但是，关系型数据库的吞吐量和性能方面存在瓶颈，频繁的读写消息会给数据库造成压力。所以，在真正的高并发场景下，该方案也会有瓶颈和限制的。

MQ(非事物)

通常情况下，在使用非事务消息支持的 MQ 产品时，我们很难将业务操作与对 MQ 的操作放在一个本地事务域中管理。通俗点描述，还是以上述提到的“跨行转账”为例，我们很难保证在扣款完成之后对 MQ 投递消息的操作就一定能成功。这样一致性似乎很难保证。

先从消息生产者这端来分析，请看伪代码：

```
public void trans(){
  try{
      //1.操作数据库

       bool result = dao.update(model);//操作数据库失败，会抛出异常

      //2.如果第一步成功 则操作消息队列（投递消息）

       if(result){
          mq.append(model); //如果mq.append方法执行失败（投递消息失败），方法内部会抛出异常
       }

   } catch(Exception ex){
      rollback(); //如果发生异常 则回滚
   }
}
```

根据上述代码及注释，我们来分析下可能的情况：

1. 操作数据库成功，向 MQ 中投递消息也成功，皆大欢喜
2. 操作数据库失败，不会向 MQ 中投递消息了
3. 操作数据库成功，但是向 MQ 中投递消息时失败，向外抛出了异常，刚刚执行的更新数据库的操作将被回滚

从上面分析的几种情况来看，貌似问题都不大的。那么我们来分析下消费者端面临的问题：

1. 消息出列后，消费者对应的业务操作要执行成功。如果业务执行失败，消息不能失效或者丢失。需要保证消息与业务操作一致
2. 尽量避免消息重复消费。如果重复消费，也不能因此影响业务结果

如何保证消息与业务操作一致，不丢失？

主流的 MQ 产品都具有持久化消息的功能。如果消费者宕机或者消费失败，都可以执行重试机制的（有些 MQ 可以自定义重试次数）。

如何避免消息被重复消费造成的问题？

1. 保证消费者调用业务的服务接口的幂等性

2. 通过消费日志或者类似状态表来记录消费状态，便于判断（建议在业务上自行实现，而不依赖 MQ 产品提供该特性）

总结：这种方式比较常见，性能和吞吐量是优于使用关系型数据库消息表的方案。如果 MQ 自身和业务都具有高可用性，理论上是可以满足大部分的业务场景的。不过在没有充分测试的情况下，不建议在交易业务中直接使用。

MQ（事务消息）

其他补偿方式

做过支付宝交易接口的同学都知道，我们一般会在支付宝的回调页面和接口里，解密参数，然后调用系统中更新交易状态相关的服务，将订单更新为付款成功。同时，只有当我们回调页面中输出了 success 字样或者标识业务处理成功相应状态码时，支付宝才会停止回调请求。否则，支付宝会每间隔一段时间后，再向客户方发起回调请求，直到输出成功标识为止。

其实这就是一个很典型的补偿例子，跟一些 MQ 重试补偿机制很类似。

一般成熟的系统中，对于级别较高的服务和接口，整体的可用性通常都会很高。如果有些业务由于瞬时的网络故障或调用超时等问题，那么这种重试机制其实是非常有效的。

当然，考虑个比较极端的场景，假如系统自身有 bug 或者程序逻辑有问题，那么重试 1W 次那也是无济于事的。那岂不是就发生了“明明已经付款，却显示未付款不发货”类似的悲剧？

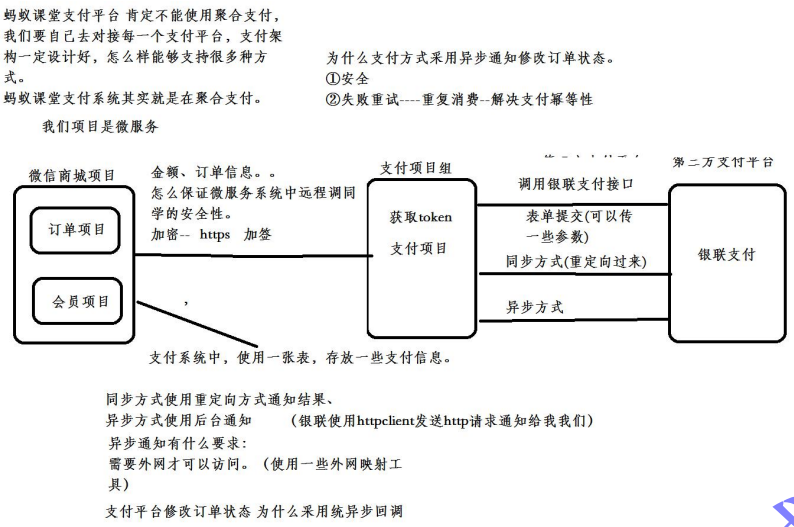
其实为了交易系统更可靠，我们一般会在类似交易这种高级别的服务代码中，加入详细日志记录的，一旦系统内部引发类似致命异常，会有邮件通知。同时，后台会有定时任务扫描和分析此类日志，检查出这种特殊的情况，会尝试通过程序来补偿并邮件通知相关人员。

在某些特殊的情况下，还会有“人工补偿”的，这也是最后一道屏障。

补充资料

项目问题

支付项目支付流程



支回调怎么保证幂等性

产生：第三方支付网关，重试机制造成幂等性

判断支付结果标识 注意：回调接口中，如果调用耗时代码，使用 mq 异步推送

支回调数据安全性

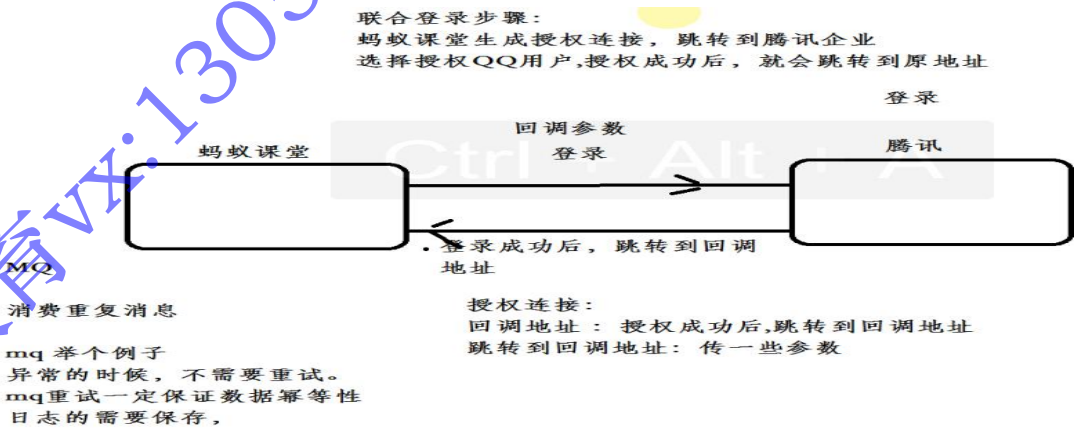
Token、对称加密 base64 加签、rsa

正回调中，项目宕机。

使用对账（第三方支付交易接口），去进行查询。

对账（第三方交易平台交易结果）

网页授权 OAuth2.



联合登录步骤：

蚂蚁课堂生成授权连接，跳转到腾讯企业

选择授权 QQ 用户,授权成功后，就会跳转到原地址

授权连接：

回调地址 ：授权成功后,跳转到回调地址

跳转到回调地址：传一些参数

跳转到回调地址：

传一个授权 code 有效期 10 分钟　授权 code 使用完毕之后，直接删除，不能重复使用

授权码的作用：使用授权码换取 aess_token，使用 aess_token 换取 openid

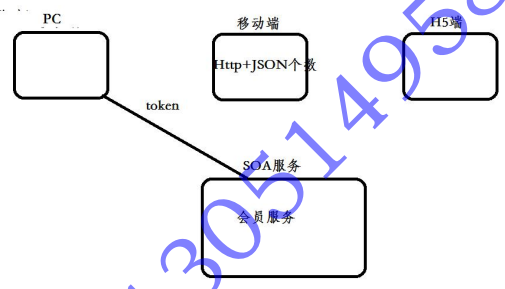
openid 作用：唯一用户主键（授权系统会员主键，不代码腾讯 userid）

openid 和我们用户表中存放一个 openid 进行关联

使用 openid 调用腾讯会员接口查询 QQ 信息

本地回调

你们登录流程



题目：你们告诉一个架构方案，登录，能够支持PC端、移动端、H5端

登录具体流程：

首先我们提供一个登录接口，登录成功后，生成token、将用户userId存放在redis中，对应的token返回给客户端会话保存使用token、

使用token，在redis查找userID、查询到数据库，对应返回信息给客户端。

使用redis+token登录 解决分布式情况，session共享，大公司不用session。

项目发布的时候，session失效了怎么办?

你在开发中遇到了那些难题，是怎么解决的？

跨域

跨域原因产生：在当前域名请求网站中，默认不允许通过 ajax 请求发送其他域名。

XMLHttpRequest cannot load 跨域问题解决办法

使用后台 response 添加 header

后台 response 添加 header，response.setHeader("Access-Control-Allow-Origin", "*")；支持所有网站

使用 JSONP

JSONP 的优缺点：

JSONP 只支持 get 请求不支持 psot 请求

什么是 SQL 语句注入

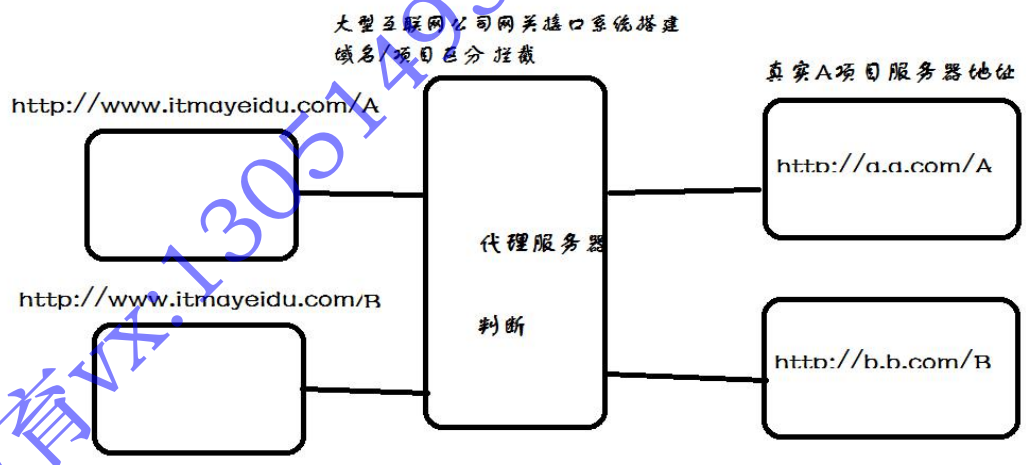
使用接口网关

使用 nginx 转发。

配置：

```
server {
        listen       80;
        server_name  www.itmayiedu.com;
        location /A {
              proxy_pass  http://a.a.com:81/A;
                index  index.html index.htm;
        }
           location /B {
              proxy_pass  http://b.b.com:81/B;
                index  index.html index.htm;
        }
    }
```

相关图：



使用内部服务器转发

内部服务器使用 HttpClient 技术进行转发

同步接口中保证数据一致性问题

例如 A 调用 B 接口，B 接口没有及时反应，怎么进行补偿？

日志记录，任务调度定时补偿，自动重试机制。

任务调度幂等性问题

①使用 zookeeper 实现分布式锁 缺点(需要创建临时节点、和事件通知不易于扩展)
②使用配置文件做一个开关 缺点发布后，需要重启
③数据库唯一约束，缺点效率低
④使用分布式任务调度平台
XXLJOB

MQ 幂等性问题

解决办法:
使用日志+msg-id 保存报文信息，作为去重和幂等的依据。
消费端代码抛出异常，不需要重试补偿，使用日志记录报文，下次发版本解决。

如何解决 session 共享？

①使用 spring-session+redis 解决
②使用负载均衡策略 ip 绑定
③使用 cookie 解决
④使用数据库存储
⑤tomcat 可以配 session 共享，但是非常占内存
⑥使用 tocken 重写 session

```
server.port=8080


spring.redis.database=0
spring.redis.host=192.168.110.187
spring.redis.port=6379
spring.redis.password=123456
spring.redis.pool.max-idle=8
spring.redis.pool.min-idle=0
spring.redis.pool.max-active=8
spring.redis.pool.max-wait=-1
spring.redis.timeout=5000


redis.hostname=192.168.110.185
redis.port=6379
redis.password=123456
```

```
@EnableRedisHttpSession(maxInactiveIntervalInSeconds = 1800)
public class SessionConfig {

    // 冒号后的值为没有配置文件时，制动装载的默认值
    @Value("${redis.hostname}")
    String HostName;
    @Value("${redis.port}")
    int Port;
    @Value("${redis.port:6379}")
    String pass;

    @Bean
    public JedisConnectionFactory connectionFactory() {
        JedisConnectionFactory connection = new JedisConnectionFactory();
        connection.setPort(Port);
        connection.setHostName(HostName);
        return connection;
    }
}
```

高并发解决方案(高可用)？

数据库：

1. 慢查询定位 sql 语句
2. Sql 语句优化
3. 减少全表扫描
4. 使用索引(索引注意事项)
5. 分库分表(水平分割+垂直分割)
6. 水平 取模算法
7. 主从复制(mysql 集群)，原理二进制文件
8. 读写分离

缓存机制：

9. 使用 redis，redis 集群(主从复制)
10. 使用 redis 读写分离
11. 使用 redis 哨兵机制监听

服务器：

反向代理，配置负载均衡，集群，动静分离

客户端：

1. 减少请求
2. 用户请求好，最好使用 ajax 异步请求
3. 动静分离，CDN 加速

项目优化：

1. 代码重构
2. Jvm 虚拟机调优，垃圾回收机制，

品优购

1. 淘淘网站并发数：

经过压力测试可以支持 3000 左右的并发，可以满足目前的业务需求。由于我们的系统是分布式架构，支持水平扩展，如果将来并发量提高的话，可以增加服务器来提高并发量。

2. 人员配置

产品经理：3 人，确定需求以及给出产品原型图。
项目经理：1 人，项目管理。
前端团队：5 人，根据产品经理给出的原型制作静态页面。
后端团队：20 人，实现产品功能。
测试团队：5 人，测试所有的功能。
运维团队：3 人，项目的发布以及维护。

3. 开发周期

采用迭代开发的方式进行，一般一次迭代的周期为一个月左右。

4. Sku

最小库存量单位。
Sku==商品 id

5. 你说你用了 redis 缓存，你 redis 存的是什么格式的数据，是怎么存的

redis 中存储的都是 key-value 格式的。拿商品数据来说，key 就是商品 id，value 是商品相关信息的 json 数据。

6. 你前台 portal 采用 4 台服务器集群部署，那能前台高并发访问性能提上去了，那数据库会不会造成一个瓶颈，这一块你是怎么处理的？

portal 系统在高并发的情况下如果每次请求都请求都查询数据库确实会出现数据库的瓶颈。为了降低数据库压力，在服务层会添加一个缓存，用 redis 实现，这样的话请求先到缓存中查找是否有缓存的内容，如果有直接从缓存中取数据，如果没有再到数据库中查询。这样数据库的压力就不会那么大了。

7. 你购物车存 cookie 里边 可以实现不登录就可以使用购物车 那么我现在没有登录把商品存购物车了 然后登录了 然后我换台电脑并且登录了还能不能看见我购物车的信息？如果看不到怎么做到 cookie 同步，就是在另外一台电脑上可以看到购物车信息

回答：

淘淘商城现阶段使用的仅仅是把购物车的商品写入 cookie 中 ,这样服务端基本上么有存储的压力。但是弊端就是用户更换电脑后购物车不能同步。打算下一步这么实现：当用户没有登录时向购物车添加商品是添加到 cookie 中，当用户登录后购物车的信息是存储在 redis 中的并且是跟用户 id 向关联的 ,此时你更换电脑后使用同一账号登录购物车的信息就会展示出来。

8. 如果用户一直添加购物车添加商品怎么办？并且他添加一次你查询一次数据库？互联网上用户那么多，这样会对数据库造成很大压力你怎么办？

当前我们使用 cookie 的方式来保存购物车的数据 ,所以当用户往购物车中添加商品时，并不对数据库进行操作。将来把购物车商品放入 redis 中，redis 是可以持久化的可以永久保存 ,此时就算是频繁的往购物车中添加数据也没用什么问题。

9. 电商活动倒计时方案

1、确定一个基准时间。可以使用一个 sql 语句从数据库中取出一个当前时间。SELECT NOW()；

2、活动开始的时间是固定的。

3、使用活动开始时间-基准时间可以计算出一个秒为单位的数值。

4、在 redis 中设置一个 key（活动开始标识）。设置 key 的过期时间为第三步计算出来的时间。

5、展示页面的时候取出 key 的有效时间。Ttl 命令。使用 js 倒计时。

6、一旦活动开始的 key 失效，说明活动开始。

7、需要在活动的逻辑中，先判断活动是否开始。

10. 秒杀抢购库存解决方案

1、把商品的数量放到 redis 中。

2、秒杀时使用 decr 命令对商品数量减一。如果不是负数说明抢到。

3、一旦返回数值变为 0 说明商品已售完。

11. dubbo 服务开发流程，运行流程？zookeeper 注册中心的作用？

使用流程：

第一步：要在系统中使用 dubbo 应该先搭建一个注册中心，一般推荐使用 zookeeper。

第二步：有了注册中心然后是发布服务，发布服务需要使用 spring 容器和 dubbo 标签来发布服务。并且发布服务时需要指定注册中心的位置。

第三步：服务发布之后就是调用服务。一般调用服务也是使用 spring 容器和 dubbo 标签来引用服务，这样就可以在客户端的容器中生成一个服务的代理对象，在 action 或者 Controller 中直接调用 service 的方法即可。

Zookeeper 注册中心的作用主要就是注册和发现服务的作用。类似于房产中介的作用，在系统中并不参与服务的调用及数据的传输。

12. redis 为什么可以做缓存？项目中使用 redis 的目的是什么？redis 什么时候使用？

1)Redis 是 key-value 形式的 nosql 数据库。可以快速的定位到所查找的 key，并把其中的 value 取出来。并且 redis 的所有的数据都是放到内存中，存取的速度非常快，一般都是用来做缓存使用。

2) 项目中使用 redis 一般都是作为缓存来使用的，缓存的目的就是为了减轻数据库的压力提高存取的效率。

3) 在互联网项目中只要是涉及高并发或者是存在大量读数据的情况下都可以使用 redis 作为缓存。当然 redis 提供丰富的数据类型，除了缓存还可以根据实际的业务场景来决定 redis 的作用。例如使用 redis 保存用户的购物车信息、生成订单号、访问量计数器、任务队列、排行榜等。

13. acitveMQ 的作用、原理？（生产者。消费者。 p2p、订阅实现流程）

Activemq 的作用就是系统之间进行通信。当然可以使用其他方式进行系统间通信，如果使用 Activemq 的话可以对系统之间的调用进行解耦，实现系统间的异步通信。原理就是生产者生产消息，把消息发送给 activemq。Activemq 接收到消息，然后查看有多少个消费者，然后把消息转发给消费者，此过程中生产者无需参与。消费者接收到消息后做相应的处理和生产者没有任何关系。

14. activeMQ 在项目中如何应用的？

Activemq 在项目中主要是完成系统之间通信 ,并且将系统之间的调用进行解耦。例如在添加、修改商品信息后，需要将商品信息同步到索引库、同步缓存中的数据以及生成静态页面一系列操作。在此场景下就可以使用 activemq。一旦后台对商品信息进行修改后，就向 activemq 发送一条消息，然后通过 activemq 将消息发送给消息的消费端，消费端接收到消息可以进行相应的业务处理。

15. activeMQ 如果数据提交不成功怎么办？

Activemq 有两种通信方式，点到点形式和发布订阅模式。如果是点到点模式的话，如果消息发送不成功此消息默认会保存到 activemq 服务端知道有消费者将其消费，所以此时消息是不会丢失的。

如果是发布订阅模式的通信方式，默认情况下只通知一次，如果接收不到此消息就没有了。这种场景只适用于对消息送达率要求不高的情况。如果要求消息必须送达不可以丢失的话，需要配置持久订阅。每个订阅端定义一个 id，在订阅是向 activemq 注册。发布消息和接收消息时需要配置发送模式为持久化。此时如果客户端接收不到消息，消息会持久化到服务端，直到客户端正常接收后为止。

16. 当被问到某个模快存在安全性问题（sso 单点登录系统）时，如何回答？

目前淘淘商城的 sso 系统的解决方案中直接把 token 保存到 cookie 中，确实存在安全性问题。但是实现简单方便。如果想提高安全性可以使用 cas 框架实现单点登录。

17. 当技术面试官问到你某个技术点更深层次研究时，自己没有深入了解怎么回答？

如果没有深入研究就直接回答不知道就可以了。

18. solr 怎么设置搜索结果排名靠前（得分）？

可以设置文档中域的 boost 值，boost 值越高计算出来的相关度得分就越高，排名也就越靠前。此方法可以把热点商品或者是推广商品的排名提高。

19. solr 的原理

Solr 是基于 Lucene 开发的全文检索服务器，而 Lucene 就是一套实现了全文检索的 api，其本质就是一个全文检索的过程。全文检索就是把原始文档根据一定的规则拆分成若干个关键词，然后根据关键词创建索引，当查询时先查询索引找到对应的关键词，并根据关键词找到对应的文档，也就是查询结果，最终把查询结果展示给用户的过程。

20. solr 里面 IK 分词器的原理

IK 分析器的分词原理本质上是词典分词。现在内存中初始化一个词典，然后在分词过程中逐个读取字符，和字典中的字符相匹配，把文档中的所有的词语拆分出来的过程。

21. 支付接口是怎么做的？

面试中可以说支付这部分不是我们做的，我们项目中并没有涉及支付部分的处理。如果想了解支付是如何实现可以参考之前学过的易宝支付相关处理以及支付宝、微信支付相关文档。

22. 业务如何说？先说业务、说表、说具体实现？

先说总体的业务流程，然后再说具体业务的实现方法及使用的技术。最后说你在系统中负责的内容。不需要说表结构。

23. 单点登录系统，如果 cookie 禁用，你们怎么解决？

如果禁用 cookie 可以使用 url 中带参数，把 token 传递给服务端。当然此方法涉及安全性问题，其实在 cookie 中保存 token 同样存在安全性问题。推荐使用 sso 框架 CAS 实现单点登录。

24. 你们做移动端没有，如果没有移动端，你们为什么做单点登录？

单点登录并不是为移动端准备的，移动端有自己的登录方式。单点登录是解决在同一个公司内部多个互信网站之间进行跳转时不需要多次登录，多个系统统一登录入口。

25. 单点登录的核心是什么？

单点登录的核心是如何在多个系统之间共享身份信息。

26. 除了单点登陆，还做过什么登陆的方式？

这是什么狗屁问题？除了单点登录那就是普通登录方式，用户在同一个公司的多个系统之间跳转时需要多次登录。

27. 单点登录，http 无状态的，别人模仿如何在后端处理

http 是无状态的，如果别人模仿浏览器发送 http 请求，一般后台是无法识别的。如果对安全要求高的情况下应该是 https 协议。可以保证在通信过程中无法窃取通信内容。

28. 安全性问题（别的网站使用爬虫技术爬你的网站怎么办？有没有安全措施）

单位时间内请求次数超过某个阈值就让输入验证码，可以极大降低抓取的速度，如果多次超过某个阀值可以加入黑名单。还有就是页面内容使用 json 返回，数据经常变一变格式，或者 js 动态生成页面内容。

29. 商品存入数据库怎么保证数据库数据安全?

1)对用户安全管理

用户操作数据库时，必须通过数据库访问的身份认证。删除数据库中的默认用户，使用自定义的用户及高强度密码。

2)定义视图

为不同的用户定义不同的视图，可以限制用户的访问范围。通过视图机制把需要保密的数据对无权存取这些数据的用户隐藏起来，可以对数据库提供一定程度的安全保护。实际应用中常将视图机制与授权机制结合起来使用，首先用视图机制屏蔽一部分保密数据，然后在视图上进一步进行授权。

3)数据加密

数据加密是保护数据在存储和传递过程中不被窃取或修改的有效手段。

4)数据库定期备份

5)审计追踪机制

审计追踪机制是指系统设置相应的日志记录，特别是对数据更新、删除、修改的记录，以便日后查证。日志记录的内容可以包括操作人员的名称、使用的密码、用户的 IP 地址、登录时间、操作内容等。若发现系统的数据遭到破坏，可以根据日志记录追究责任，或者从日志记录中判断密码是否被盗，以便修改密码，重新分配权限，确保系统的安全。

30. 订单表的数据量太大,我把订单分到许多表中,那么我我想用一条 sql 查处所有的订单,怎么解决?

分库情况下：可以使用 mycat 数据库中间件实现多个表的统一管理。虽然物理上是把一个表中的数据保存到多个数据库中，但是逻辑上还是一个表，使用一条 sql 语句就可以把数据全部查询出来。

单库情况下：需要动态生成 sql 语句。先查询订单相关的表，然后将查询多个表的 sql 语句使用 union 连接即可。

31. 咱们单点登录模块中，别人伪造我们 cookie 中的 token 怎么办?

服务端是无法阻止伪造 cookie 的，如果对安全性要求高的话可以可使用 cas 框架。

32. 第一个是当两个客户同时买一件商品时库存只有一个了,怎么控制?

可以使用 mysql 的行锁机制，实现乐观锁，在更新商品之前将商品锁定，其他用户无法读取，当此用户操作完毕后释放锁。当并发量高的情况下，需要使用缓存工具例如 redis 来管理库存。

33. 对数据库只是采用了读写分离,并没有完全解决数据库的压力,那么有什么办法解决?

如果数据库压力确实很大的情况下可以考虑数据库分片，就是将数据库中表拆分到不同的数据库中保存。可以使用 mycat 中间件。

34. 同一账号以客户端登录怎么挤掉另一端。

用户登录后需要在 Session 中保存用户的 id。当用户登录时，从当前所有的 Session 中判断是否有此用户 id 的存在，如果存在的话就把保存此用户 id 的 Session 销毁。

35. solr 的索引查询为什么比数据库要快。

Solr 使用的是 Lucene API 实现的全文检索。全文检索本质上是查询的索引。而数据库中并不是所有的字段都建立的索引，更何况如果使用 like 查询时很大的可能是不使用索引，所以使用 solr 查询时要比查数据库快。

36. solr 索引库个别数据索引丢失怎么办。

首先 Solr 是不会丢失个别数据的。如果索引库中缺少数据，那就向索引库中添加。（靠！什么狗屁问题！！！）

37. Lucene 索引优化。

直接使用 Lucene 实现全文检索已经是过时的方案，推荐使用 solr。Solr 已经提供了完整的全文检索解决方案。

问题 1：为什么要选用 redis？

由于每个系统都单独部署运行一个单独的 tomcat，所以，不能将用户的登录信息保存到 session 中（多个 tomcat 的 session 不共享），所以选用 redis 来缓存登录信息，当用户登录时，将用户登录信息保存到 redis 中，并生成一个 token 保存到 cookie 中（不太确定是否是这么实现的？）

问题 2：单点登录系统的基本流程？

当用户点击登录按钮的时候，用户输入用户名和密码，检验用户名是否在数据库中存在，然后用户名密码是否正确。这里的密码是用了 spring 的 MD5 加密技术。当全部成功后，将 sessionId（也可以生成一个 UUID）写入 cookie 中供前端调用，写入浏览器的 cookie 中，然后存入到 redis 中（key 是 sessionId，value 是用户信息），并设置有效期。

这里的 cookie 是设置了共同的 name，所以不论是什么系统进行登录，前端页面都会存有这个 name 的 cookie，也就实现了所有子系统都可以访问到 cookie。

当用户登录其他子系统时，先从 cookie 中获取 token 信息（也就是 sessionId），根据 token 信息获取用户信息。用户每次与网站的交互，比如查看产品，则刷新一次 redis 的时间，重新设置有效期，这个效果是通过拦截器来实现的。

拦截器的拦截，在 springMVC.xml 中设置拦截的名称。

问题 3：单点登录之跨域问题？

将 cookie 存在一个公共的站点的页面上就可以了,这里我们管那个站点叫主站 S。

环境 1:a.xxx.com 需要跟 b.xxx.com 实现跨域,这种比较简单,只需要设置 cookie 的域名关联域就可以了 cookie.Domain = "xxx.com",这样两个域名间的 cookie 就可以互相访问,实现跨域。【项目中使用的这种环境】

环境 2:a.aaa.com 需要跟 b.bbb.com 实现跨域,这种不同域名的情况下,想要实现就必须换种方式.

在这里我将引入第三者,s.sss.com 这个站点,就是某个浏览器同时打开了这 3 个站点,我们访问 A 站点,先判断自身是否登录,如果 session 为空,就重定向到 S 站点,判断 S 站点上面是否有 cookie,如果 S 站点上面也没有 cookie,则由 S 站点重定向到 A 站点的登录页.

这样我们就实现了第一步,S 站做的的就是隐藏在幕后,子站先判断自己是否存在 session,如果不存在,就重定向到主站 S 上面去验证.

第二步,验证登录信息合法性.这里我引入 token(令牌),网上有很多资料,描述 token 的传递,工作方式是这样,A 登录成功,保存自身的 session,重定向到 S,S 在自己站点保存一个 session 跟 cookie,session 保存 token 对象 {tokenID,userName,startTime,endTime},cookie 保存 tokenID,tokenID 是一个 Guid,把 token 对象缓存在集合里面,另起一个线程,根据 endTime(过期时间)来定期清理集合列表,重定向到 A 的时候再将 tokenID 传递过去,拿到 tokenID 后,进入验证环节,S 站有提供一个接口,根据 tokenID 获取 token 对象,如果获取到对象,且没有失效,则 tokenID 合法,跳入 index 页面.情况 2,A 登录,直接打开 B,这时候 B 自身没有 session,会主动请求主站,主站会返回 cookieID(S 站存在客户端的 cookie),这个时候再走验证环节,如果通过,则 B 根据 token 对象创建自身的 session,再跳入 index。

问题 4：忘记密码找回密码的流程？

在注册的时候会设置找回密码的问题和答案，在非登录状态忘记密码后，需要通过回答问题和答案找回密码。根据用户名找到用户设置的问题，然后回答完问题后，生成一个 UUID，缓存在 redis 中设置有效期，并返回这个 UUID。然后前端将这个 UUID 和新密码传入重置密码的函数，将传入的 UUID 和之前缓存在 redis 中的 UUID 进行比较，如果相同，则对新密码进行 md5 加密后更新到数据库中。

1、电商项目中有没有用到多线程，哪些地方要用多线程？

2、你项目对于订单是怎么处理的，假如一个客户在下订单的时候没有购买怎么办，对于顾客在购买商品的时候你们怎么处理你们的库存？

3、计算一下 133 平方是多少？

4、你平时测试的流程？

5、你们数据库怎么设计的？

6、你们怎么处理 redis 缓存的数据，怎么删除的？

7、你觉得分布式开发的缺点是什么？

8、缓存技术你觉得在什么时候用的比较多？

9、你们怎么管理你们的内存？

10、说说你对于 web 前端的优化？

11、插入商品的话，要求级联插入几张表，你们当时是怎么实现的？

12、支付接口是怎么做的？

13、redis 为什么可以做缓存？

14、当被问到某个模快存在安全性问题（sso 单点登录系统）时，如何回答？

15、solr 怎么设置搜索结果排名靠前（得分）？

16、activeMQ 在项目中如何应用的？

17、activeMQ 如果数据提交不成功怎么办？

面试题：Nginx 是什么、可以做什么、有什么优势？

（1）Nginx 是一个 http 服务器。是一个使用 c 语言开发的高性能的 http 服务器及反向代理服务器。

（2）Nginx 可以作为 Web 服务器：Nginx 使用更少的资源，处理请求是异步非阻塞的，支持更多的并发连接，体现更高的效率，高可靠性；Nginx 作为负载均衡服务器，实现集群功能；nginx 可以作为邮件代理服务器。

（3）优势：

1、轻量级，同样起 web 服务，比 apache 占用更少的内存及资源

2、高并发，nginx 处理请求是异步非阻塞的，而 apache 则是阻塞型的，在高并发下 nginx 能保持低资源低消耗高性能

3、Nginx 本身就是一个反向代理服务器

4、Nginx 支持 7 层负载均衡

5、nginx 适合做静态，简单，效率高

面试题：什么是正向代理、什么是反向代理、有什么区别？

正向代理：假如我们要访问国外的某些网站，但是国内访问不了，而某服务器可以访问，我们将请求发送给这台服务器（称为代理服务器），让代理服务器

去访问国外的网站，然后将访问到的数据传递给我们！

正向代理最大的特点是客户端非常明确要访问的服务器地址；服务器只清楚请求来自哪个代理服务器，而不清楚来自哪个具体的客户端；正向代理模式屏蔽或者隐藏了真实客户端信息。

反向代理：多个客户端给服务器发送的请求，Nginx 服务器接收到之后，按照一定的规则分发给了后端的业务处理服务器进行处理了。此时~请求的来源也就是客户端是明确的，但是请求具体由哪台服务器处理的并不明确了，Nginx 扮演的就是一个反向代理角色。反向代理，主要用于服务器集群分布式部署的情况下，反向代理隐藏了服务器的信息！

什么是数据库分片

简单来说，就是指通过某种特定的条件，将我们存放在同一个数据库中的数据分散存放到多个数据库上面，以达到分散单台设备负载的效果。

数据的切分（Sharding）根据其切分规则的类型，可以分为两种切分模式。

1.一种是按照不同的表来切分到不同的数据库（主机）之上，这种切可以称之为数据的垂直切分

2.另外一种则是根据表中的数据的逻辑关系，将同一个表中的数据按照某种条件拆分到多台数据库上面，这种切分称之为数据的水平切分。

如何实现数据库分片

当数据库分片后，数据由一个数据库分散到多个数据库中。此时系统要查询时需要切换不同的数据库进行查询，那么系统如何知道要查询的数据在哪个数据库中？当添加一条记录时要向哪个数据库中插入呢？这些问题处理起来都是非常的麻烦。

这种情况下可以使用一个数据库中间件 mycat 来解决相关的问题。

什么是 Mycat?

简单的说，MyCAT 就是：一个新颖的数据库中间件产品，支持 mysql 集群，提供高可用性数据分片集群。你可以像使用 mysql 一样使用 mycat。对于开发人员来说根本感觉不到 mycat 的存在。

Mycat 读写分离

数据库读写分离对于大型系统或者访问量很高的互联网应用来说，是必不可少的一个重要功能。对于 MySQL 来说，标准的读写分离是主从模式，一个写节点 Master 后面跟着多个读节点，读节点的数量取决于系统的压力，通常是 1-3 个读节点的配置

单点登录系统

主要解决的是 Session 共享的问题。

1、使用 redis 管理 Session。

1）key：token

2）value：用户信息

3）可以需要设置有效期。

4）需要把 token 保存到 cookie 中。

购物车

1、未登录：使用 cookie 保存购物车数据

2、登录后：把购物车数据保存到 redis。

2、购物车合并，应该以服务端的购物车为准。

电商活动倒计时方案：

1、确定一个基准时间。可以使用一个 sql 语句从数据库中取出一个当前时间。SELECT NOW();

2、活动开始的时间是固定的。

3、使用活动开始时间-基准时间可以计算出一个秒为单位的数值。

4、在 redis 中设置一个 key（活动开始标识）。设置 key 的过期时间为第三步计算出来的时间。

5、展示页面的时候取出 key 的有效时间。Ttl 命令。使用 js 倒计时。

6、一旦活动开始的 key 失效，说明活动开始。
7、需要在活动的逻辑中，先判断活动是否开始。
秒杀方案：
8、把商品的数量放到 redis 中。
9、秒杀时使用 decr 命令对商品数量减一。如果不是负数说明抢到。
10、一旦返回数值变为 0 说明商品已售完。
由于宜立方商城是基于 SOA 的架构，表现层和服务层是不同的工程。所以要实现商品列表查询需要两个系统之间进行通信。
如何实现远程通信？
1、Webservice：效率不高基于 soap 协议。项目中不推荐使用。
2、使用 restful 形式的服务：http+json。很多项目中应用。如果服务太多，服务之间调用关系混乱，需要治疗服务。
3、使用 dubbo。使用 rpc 协议进行远程调用，直接使用 socket 通信。传输效率高，并且可以统计出系统之间的调用关系、调用次数。
什么是 dubbo
DUBBO 是一个分布式服务框架，致力于提供高性能和透明化的 RPC 远程服务调用方案
Dubbo 就是资源调度和治理中心的管理工具。

分布式锁

Java 提供了两种内置的锁的实现，一种是由 JVM 实现的 synchronized 和 JDK 提供的 Lock，当你的应用是单机或者说单进程应用时，可以使用 synchronized 或 Lock 来实现锁。当应用涉及到多机、多进程共同完成时，那么这时候就需要一个全局锁来实现多个进程之间的同步。

1. 使用场景

例如一个应用有手机 APP 端和 Web 端，如果在两个客户端同时进行一项操作时，那么就会导致这项操作重复进行。

2. 实现方式

2.1 数据库分布式锁

基于 MySQL 锁表

该实现方式完全依靠数据库唯一索引来实现。当想要获得锁时，就向数据库中插入一条记录，释放锁时就删除这条记录。如果记录具有唯一索引，就不会同时插入同一条记录。这种方式存在以下几个问题：

锁没有失效时间，解锁失败会导致死锁，其他线程无法再获得锁。

只能是非阻塞锁，插入失败直接就报错了，无法重试。

不可重入，同一线程在没有释放锁之前无法再获得锁。

采用乐观锁增加版本号

根据版本号来判断更新之前有没有其他线程更新过，如果被更新过，则获取锁失败。

2.2 Redis 分布式锁

基于 SETNX、EXPIRE

使用 SETNX（set if not exist）命令插入一个键值对时，如果 Key 已经存在，那么会返回 False，否则插入成功并返回 True。因此客户端在尝试获得锁时，先使用 SETNX 向 Redis 中插入一个记录，如果返回 True 表示获得锁，返回 False 表示已经有客户端占用锁。

EXPIRE 可以为一个键值对设置一个过期时间，从而避免了死锁的发生。

RedLock 算法

ReadLock 算法使用了多个 Redis 实例来实现分布式锁，这是为了保证在发生单点故障时还可用。

尝试从 N 个相互独立 Redis 实例获取锁，如果一个实例不可用，应该尽快尝试下一个。

计算获取锁消耗的时间，只有当这个时间小于锁的过期时间，并且从大多数（N/2+1）实例上获取了锁，那么就认为锁获取成功了。

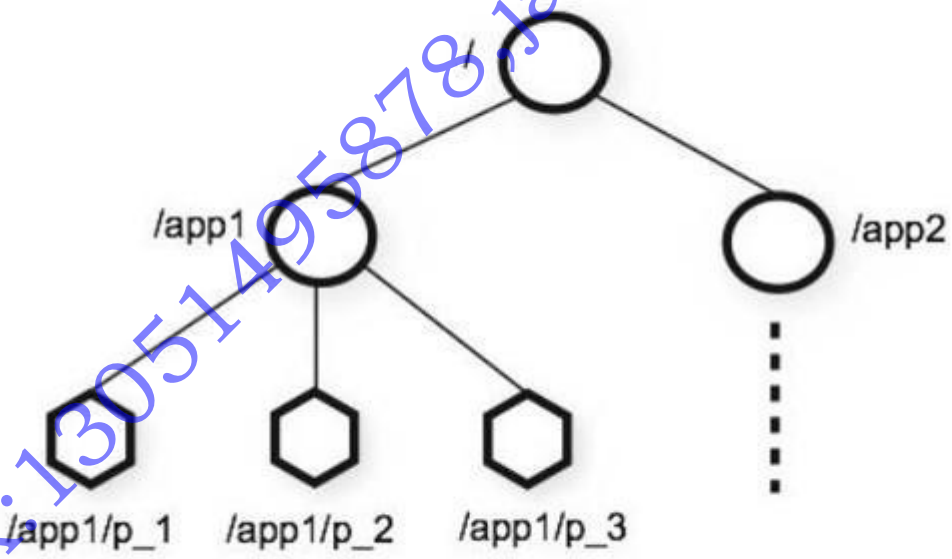
如果锁获取失败，会到每个实例上释放锁。

2.3 Zookeeper 分布式锁

Zookeeper 是一个为分布式应用提供一致性服务的软件，例如配置管理、分布式协同以及命名的中心化等，这些都是分布式系统中非常底层而且是必不可少的基本功能，但是如果自己实现这些功能而且要达到高吞吐、低延迟同时还要保持一致性和可用性，实际上非常困难。

抽象模型

Zookeeper 提供了一种树形结构级的命名空间，/app1/p_1 节点表示它的父节点为 /app1。



.

节点类型

永久节点：不会因为会话结束或者超时而消失；

临时节点：如果会话结束或者超时就会消失；

有序节点：会在节点名的后面加一个数字后缀，并且是有序的，例如生成的有序节点为 /lock/node-0000000000，它的下一个有序节点则为 /lock/node-0000000001，依次类推。

监听器

为一个节点注册监听器，在节点状态发生改变时，会给客户端发送消息。

分布式锁实现

创建一个锁目录 /lock。

在 /lock 下创建临时的且有序的子节点，第一个客户端对应的子节点为 /lock/lock-0000000000，第二个为 /lock/lock-0000000001，以此类推。

客户端获取 /lock 下的子节点列表，判断自己创建的子节点是否为当前子节点列表中序号最小的子节点，如果是则认为获得锁，否则监听自己的前一个子节点，获得子节点的变更通知后重复此步骤直至获得锁；

执行业务代码，完成后，删除对应的子节点。

会话超时

如果一个已经获得锁的会话超时了，因为创建的是临时节点，因此该会话对应的临时节点会被删除，其它会话就可以获得锁了。可以看到，Zookeeper 分布式锁不会出现数据库分布式锁的死锁问题。

羊群效应

在步骤二，一个节点未获得锁，需要监听监听自己的前一个子节点，这是因为如果监听所有的子节点，那么任意一个子节点状态改变，其它所有子节点都会收到通知，而我们只希望它的下一个子节点收到通知。

1.多系统之间怎么实现通信的？A 系统—》B 系统的服务

有两种通信方式，第一种是利用 HttpClient，HttpClient 提供了 http 服务的能力，其工作原理就类似于我们去打开浏览器访问一个网页去获取数据，最终网页将数据展现出来。HttpClient 可以利用 get 或者 post 请求去抓取一个接口的数据，从而得到我们需要的数据。

还有一种便是 MQ,使用前，首先搭建一个 rabbitMQ 的服务器，MQ 和 HttpClient 不同的地方在于 HttpClient 是同步调用，而 MQ 可以解耦的异步调用的，正是因为这个原因，MQ 才很好的解决了同步的响应速度慢的问题。在这里我们使用的是 rabbiMq,同类的产品还有例如 ActiveMQ，Kafka 等。什么时候使用异步，什么时候同步？ （比如我们的缓存系统）

2.Solr 集群的搭建

服务器的数量：zookeeper：3 台服务器
solr：4 台服务器
先搭建 zookeeper 集群，因为 zookeeper 集群有存活过半机制，一般服务器选用奇数台最少 3 台，因为一台就不叫集群了叫 zookeeper 服务器了，一个 leader 主节点，两个 follower 节点
搭建完 zookeeper 集群，启动 zookeeper,启动 4 台 tomcat 实例，更改 tomcat 端口号一般改为 8080,8081,8082,8083，再搭建 4 个单机版 solr 实例，让 zookeeper 集群集中管理配置文件，将配置文件上传到 zookeeper，将 conf 下面的内容上传到 zookeeper 集群中，修改 solr.xml 的文件，告诉每个 solr 实例 zookeeper 集群的位置，在每台 Tomcat 的 bin 目录下 catalina.bat 文件中加入 DzkHost 指定的 zookeeper 服务器地址

3.请你谈谈对 MQ 的理解？以及你们在项目中是怎么用的？

MQ（消息队列）是一种应用程序对应应用程序的通信方法，由于在高并发环境下，由于来不及同步处理，请求往往发生堵塞，通过消息队列，我们可以异步处理请求，缓解系统压力；MQ（ Message

Queue） ,即消息队列是在消息的传输过程中保存消息的容器。
通俗的说， 就是一个容器， 你把消息丢进去， 不需要立即处理。 然后有个程序去从你的容器里面把消息一条条读出来处理。一般用于应用系统解耦、 消息异步分发， 能够提高系统吞吐量。

消息队列
注册用户，发邮件（异步）
登录，发短信通知（异步），加积分（异步）
商品添加，异步更新 solr，异步更新静态页面

静态页面—库存—实时性较差
接口，库存修改后,重新生成新的静态页面

4.请你谈谈对 Redis 的认识？

Redis 是一种基于键值对的 NoSQL 数据库（非关系型数据库）；是一个 key-value 存储系统
Redis 有两个特点：高能性 可靠性
高能性：Redis 将所有数据都存储在内存中，所有读写性特别高
可靠性：Redis 将内存中的数据利用 RDB 和 AOF 的形式保存到硬盘中，这样就可以避免发生断点或机器故障时内存数据丢失的问题
功能应用
1.数据缓存功能，减少对数据库的访问压力
2.消息队列功能(轻量级)
Redis 提供了发布订阅功能和阻塞队列功能
3.计数器-应用保存用户凭证
比如计算浏览数，如果每次操作都要做数据库的对应更新操作，那将会给数据库的性能带来极大的挑战
缓存：优化网站性能，首页 （不常变的信息）
存储：单点登陆，购物车
计数器：登陆次数限制，incr
时效性：验证码 expire
订单号：数字

5.redis 空间不够，怎么保证经常访问的数据？

淘汰策略:在 redis.conf 里面配置，来保证热点数据保存在 reids 里面。

6.redis 应用场景场景：

1.缓存数据服务器
SSO 单点登录
2.应对高速读写的场景
秒杀高可用
3.分布式锁
秒杀数据一致性
4.数据共享
库存数据

7.请你谈谈对 Spring 的认识？

方便解耦，简化开发：通过 spring 提供的 ioc 容器，可以将对象之间的依赖关系交由 spring 进行控制，避免硬编码所造成的过度程序耦合
aop 编程的支持；通过 spring 提供的 aop 功能，方便进行面向切面的编程
声明式事务的支持
方便几次各种优秀框架
降低 JavaEE API 的使用难度

Spring 是一个开放源代码的设计层面框架，解决业务逻辑层和其他各层的松耦合问题
IOC 是一个生产和管理 bean 的容器,原来调用类中 new 的东西，现在在 IOC 容器中产生;
ioc 是控制反转，是 spring 的核心思想，通过面向接口编程来实现对业务组件的动态依赖
Spring 的 IOC 有三种方式注入
1、根据属性注入—set 方法注入
2、根据构造方法注入
3、根据注解注入

AOP 是面向切面的编程，将程序中的交叉业务逻辑，封装成一个切面，然后注入目标对象；是一种编程思想，将系统中非核心的业务提取出来单独处理

8.请你谈谈单点登录的实现方案？你们怎么包括 cookie 的安全性？跨域取 cookie 的问题，你们怎么解决的？

单点登录使用了 Redis+Cookie 实现
把用户信息放在 Redis 中,Key 作为用户凭证存放在 Cookie 中放在客户端,通过获取 Cookie 凭证判断用户是否有登录
Cookie 的安全性,我们的凭证是唯一的 UUID,使用工具类统一字符串命名,并且设置了 Cookie,关闭 document.cookie 的取值功能
Cookie 的跨域问题,在二级域名使用共享 Cookie 的将多个系统的域名统一作为二级域名,统一平台提供使用主域名,cookie.setPath("/")设置 Cooie 路径为根路径,通过 cookie.setDomain(".父域名")使得项目之间跨域互相访问他们的 Cookie

9.请你谈谈购物车的实现方案？当商品信息发生变更，购物车中的商品信息是否可以同步到变化？

现实中购物车有两种情况，未登录时的购物车和登录时的购物车。我们用 Redis+Cookie 的方法来实现购物车。当点击“加入购物车”按钮时，先获取用户登录凭证，如果没有登录,就将商品的 id 保存在 Redis 未登录购物车中，当拦截器拦截到用户登录时，把购物车的内容合并到数据库中登录后购物车里，通过 json 解析商品 id 查到商品信息，所以购物车中的商品信息是可以变化的。

10.如何应对高并发问题？

1.HTML 静态化，消耗最小的纯静态化的 html 页面避免大量的数据库访问请求
2.分离图片服务器,对于 web 服务器来说,图片是最消耗资源的将图片资源和页面资源进行分离,进行不同的配置优化,保证更改的系统消耗和执行效率
3.数据库集群和库表散列,数据库集群由于在架构、成本、扩张性方面都会受到所采用的关系型的限制，

在应用程序安装业务和功能模块将数据库进行分离，不同的模块对应不同的数据库或者表，再进行更小的数据库散列，最终可以再配置让系统随时增加数据库补充系统性能；
4.缓存，使用外加的 redis 模块进行缓存，减轻数据库访问压力
5.负载均衡，在服务器集群中需一台服务器调度角色 Nginx，用户所有请求先由它接收，在分配某台服务器去处理；实现负载均衡：http 重定向实现，DNS 匹配，反向代理
6.动静态分离,对于动态请求交给 Tomcat 而其他静态请求,搭建专门的静态资源服务器,使用 nginx 进行请求分发

11.Zookeeper 应用场景

1.统一配置管理
持久化节点存放配置信息,监听内容修改
2.集群管理
临时节点机器(节点)退出或者加入,Master 选举投票
临时顺序节点选举时候直接使用编号最小的即可
3.分布式锁
创建临时节点,创建成功者获得锁,执行业务操作,独占操作
也可以进行顺序执行,通过最顺序临时节点的编号
4.命名服务
/dubbo

/provider:存放服务地址
/consumer:存放消费地址(没实际意义),
/conf:存放配置信息
…
consumer 通过监听 provider 节点的内容修改实现动态读取地址,并且支持集群,只需要在 provider 中存放多个地址然后程序中通过代码实现随机调用即可
12.Nginx 应用场景

1.Http 服务器,具有提供 http 服务的能力
location /{
root /home
}
2.虚拟主机,可以对不同的进行端口映射
server{
port:端口
server_name:域名
…
}

3.反向代理,负载均衡
tomcatlist{
ip1:port [weight=n1];
ip2:port [weight=n2];
}
location{

proxy_pass : tomcatlist;
}
4.动静态分离
location ~.(jpg|css|js|png|html|htm)

13.RabbitMq 应用场景

1.系统间异步调用
添加商品时,索引工程创建索引,详情工程生成静态页面
2.顺序消费
队列的特点
3.定时任务
订单的 30 分钟后关闭
4.请求削峰
双十一和平时的时候的处理
通过消息中间件的消息存储到队列中,服务层只拿取指定数量的消息进行消费从而保证服务层的稳定性,用时间的代价换取性能和稳定的保证再通过成本可以接受的集群搭建提高时间基础

14.谈谈你对 ThreadLocal 的理解，以及他的作用

答：线程局部变量 ThreadLocal 为每个使用该变量的线程提供独立的变量副本，每次调用 set（）方法的时候，每个当前线程都有一个 ThreadLocal。应用场景：当很多线程需要多次使用同一个对象，并且需要改对象具有相同初始化值的时候最适合使用 ThreadLocal
作用：解决多线程程序并发问题

15.Erueka 和 ZooKeeper 的区别

1.Erueka 是一个是服务端,ZooKeeper 是一个进程
2.Erueka 是自我保护机制,ZooKeeper 是过半存活机制
3.Erueka 是 AP 设计,ZooKeeper 是 CP 设计
4.Erueka 没有角色的概念,ZooKeeper 有 leader 和 follow

16.池化技术

对象池技术基本原理的核心有两点：缓存和共享，即对于那些被频繁使用的对象，在使用完后，不立即将它们释放，而是将它们缓存起来，以供后续的应用程序重复使用，从而减少创建对象和释放对象的次数，进而改善应用程序的性能。事实上，由于对象池技术将对象限制在一定的数量，也有效地减少了应用程序内存上的开销。

分布式事务

指事务的每个操作步骤都位于不同的节点上，需要保证事务的 AICD 特性。
1. 产生原因
数据库分库分表;
SOA 架构，比如一个电商网站将订单业务和库存业务分离出来放到不同的节点上。
2. 应用场景
下单：减少库存同时更新订单状态。库存和订单不在不同一个数据库，因此涉及分布式事务。
支付：买家账户扣款同时卖家账户入账。买家和卖家账户信息不在同一个数据库，因此涉及分布式事务。

3. 解决方案
3.1 两阶段提交协议
两阶段提交协议可以很好得解决分布式事务问题，它可以使用 XA 来实现，XA 它包含两个部分：事务管理器和本地资源管理器。其中本地资源管理器往往由数据库实现，比如 Oracle、DB2 这些商业数据库都实现了 XA 接口，而事务管理器作为全局的协调者，负责各个本地资源的提交和回滚。
3.2 消息中间件
消息中间件也可称作消息系统 (MQ)，它本质上是一个暂存转发消息的一个中间件。在分布式应用当中，我们可以把一个业务操作转换成一个消息，比如支付宝的余额转如余额宝操作，支付宝系统执行减少余额操作之后向消息系统发一个消息，余额宝系统订阅这条消息然后进行增加账户金额操作。
3.2.1 消息处理模型
点对点

发布/订阅

3.2.2 消息的可靠性
消息的发送端的可靠性：发送端完成操作后一定能将消息成功发送到消息系统。
消息的接收端的可靠性：接收端仅且能够从消息中间件成功消费一次消息。
发送端的可靠性
在本地数据建一张消息表，将消息数据与业务数据保存在同一数据库实例里，这样就可以利用本地数据库的事务机制。事务提交成功后，将消息表中的消息转移到消息中间件，若转移消息成功则删除消息表中的数据，否则继续重传。
接收端的可靠性
保证接收端处理消息的业务逻辑具有幂等性：只要具有幂等性，那么消费多少次消息，最后处理的结果都是一样的。
保证消息具有唯一编号，并使用一张日志表来记录已经消费的消息编号。
redis中有一个命令setnx (SET IF NOT EXISTS) ，如果不存在，就设置key，将 key 的值设为 value，当且仅当 key 不存在。若给定的 key 已经存在，则 SETNX 不做任何动作。基于这个特性，我们可以对需要锁住的对象加上key，这样，同一时间就只能有一个线程拥有这把锁，从而达到分布式锁的效果。下面用一个具体的Java实例来展示redis的分布式锁效果。

Java操作redis需要用到第三方的库类，所以先在pom.xml中引入依赖。

加入依赖后，做一个redis的工具方法，分别实现的是加锁和解锁的功能。

```
public class RedisLock {

    @Autowired
    private StringRedisTemplate redisTemplate;

    /**
     * 加锁
     *
     * @param key
     * @param value 当前时间+超时时间
     * @return
     */
    public boolean lock(String key, String value) {
        //相当于setnx命令
        if (redisTemplate.opsForValue().setIfAbsent(key, value)) {
            return true;
        }
        //下面的这段代码是判断之前加的锁是否超时，是的话就更新，一定要加这段代码
        //不然就有可能出现死锁。
        String currentValue = redisTemplate.opsForValue().get(key);
        //如果锁过期
        if (!StringUtils.isEmpty(currentValue)
                && Long.parseLong(currentValue) < System.currentTimeMillis()) {
            //获取上一个锁的时间，这段代码的判断是防止多线程进入这里，只会有一个线程拿到锁
            String oldValue = redisTemplate.opsForValue().getAndSet(key, value);
            if (!StringUtils.isEmpty(oldValue)
                    && oldValue.equals(currentValue)) {
                return true;
            }
        }
        return false;
    }
```

```
    /**
     * 解锁
     *
     * @param key
     * @param value
     */
    public void unLock(String key, String value) {
        try {
            String currentValue = redisTemplate.opsForValue().get(key);
            if (!StringUtils.isEmpty(currentValue)
                    && currentValue.equals(value)) {
                redisTemplate.opsForValue().getOperations().delete(key);
            }
        } catch (Exception e) {
            log.error("【redis分布式锁】 解锁异常，{}", e);
        }
    }
}
```

现在，我们模拟一个下单的场景，假设有一个秒杀的活动，同一时间有多个线程对同一个产品进行访问，然后分别看看加锁和没加锁的结果来做对比。下面是秒杀的模拟代码：

```
public class SecKillController {

    @Autowired
    private SecKillService secKillService;

    /**
     * 查询秒杀活动特价商品的信息
     * @param productId
     * @return
     * @throws Exception
     */
    @GetMapping("/query/{productId}")
    public String query(@PathVariable String productId) throws Exception{
        return secKillService.querySecKillProductInfo(productId);
    }

    /**
     * 秒杀的方法
     * @param productId
     * @return
     * @throws Exception
     */
    @GetMapping("/order/{productId}")
    public String skill(@PathVariable String productId) throws Exception{
        log.info("@skill request ,productId:" +productId);
        secKillService.orderProductKill(productId);
        return secKillService.querySecKillProductInfo(productId);
    }
}
public class SecKillServiceImpl implements SecKillService {

    private static final int TIME_OUT = 1 * 1000;

    @Autowired
    private RedisLock redisLock;

    static Map<String, Integer> products;
    static Map<String, Integer> stock;
    static Map<String, String> orders;

    static {
        /**
         * 模拟多个表，商品信息表，库存表，秒杀成功订单表
         */
        products = new HashMap<>();
        stock = new HashMap<>();
        orders = new HashMap<>();
```

```
        products.put("123", 100000);
        stock.put("123", 100000);
    }

    /**
     * @param productId  订单id
     * @return
     */
    private String queryMap(String productId) {
        return "限量份数" + products.get(productId)
                + "还剩:" + stock.get(productId) + "份"
                + "该商品成功下单用户数目："
                + orders.size() + "人";
    }

    @Override
    public String querySecKillProductInfo(String productId) {
        return this.queryMap(productId);
    }

    @Override
    public void orderProductKill(String productId) {

        //1.查询该商品库存，为0则活动结束
        int stockNum = stock.get(productId);
        if (stockNum == 0) {
            throw new RuntimeException("活动结束");
        } else {
            //2.下单(模拟不同用户openid不同)
            orders.put(KeyUtil.getUniqueKey(), productId);
            //3.减库存
            stockNum = stockNum - 1;
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            stock.put(productId, stockNum);
        }
    }
}
```

先模拟没加锁的下单状态，我们开启工程后，用Apache ab作为压测工具来模拟高并发访问过程

在浏览器上访问查询后的订单数量，结果显示如下：

可以看到，再高并发的访问环境下，如果我们没有对订单做锁的处理，那么就可能出现数据的紊乱，导致结果不对应，这显然不符合我们的需求，下面我们来看看加上redis锁之后的访问情况，先把service中的秒杀代码加上锁。

```
@Override
public void orderProductKill(String productId) {

    //加锁，保证下面的代码单线程的访问
    long time = System.currentTimeMillis() + TIME_OUT;
    if (!redisLock.lock(productId, String.valueOf(time))) {
        throw new RuntimeException( "下单失败");
    }
    //1.查询该商品库存，为0则活动结束
    int stockNum = stock.get(productId);
    if (stockNum == 0) {
        throw new RuntimeException("活动结束");
    } else {
        //2.下单(模拟不同用户openid不同)
        orders.put(KeyUtil.getUniqueKey(), productId);
        //3.减库存
        stockNum = stockNum - 1;
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        stock.put(productId, stockNum);
```

```
    }
    //解锁
    redisLock.unLock(productId, String.valueOf(time));
}
```

然后再进行同样的操作

我们可以看到，加上锁之后的订单处理数量是正确的，也就是redis锁是起到了作用的，这是符合我们的需求的。

上面的例子相对比较简单，因为精力能力有限，楼主没法给大家展示真正的分布式锁的实现效果，但从原理上其实是一样的，都是用redis的setnx命令来加上锁，保证分布式环境下锁住的对象只能被一个线程访问，而且从实现方式上来说也比较简单 （只需要一个命令就行，很深入人心 ），因此，redis在分布式锁的应用中也被广泛使用。

Redis分布式锁的正确实现方式
可靠性
首先，为了确保分布式锁可用，我们至少要确保锁的实现同时满足以下四个条件：
互斥性。在任意时刻，只有一个客户端能持有锁。
不会发生死锁。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。
具有容错性。只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。
解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

代码实现
组件依赖
首先我们要通过Maven引入Jedis开源组件，在pom.xml文件加入下面的代码：

```
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>2.9.0</version>
</dependency>
```

加锁代码
正确姿势
Talk is cheap, show me the code。先展示代码，再带大家慢慢解释为什么这样实现：

```
public class RedisTool {

    private static final String LOCK_SUCCESS = "OK";
    private static final String SET_IF_NOT_EXIST = "NX";
    private static final String SET_WITH_EXPIRE_TIME = "PX";

    /**
     * 尝试获取分布式锁
     * @param jedis Redis客户端
     * @param lockKey 锁
     * @param requestId 请求标识
     * @param expireTime 超期时间
     * @return 是否获取成功
     */
    public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, int
expireTime) {

        String result = jedis.set(lockKey, requestId, SET_IF_NOT_EXIST, SET_WITH_EXPIRE_TIME,
expireTime);

        if (LOCK_SUCCESS.equals(result)) {
            return true;
        }
        return false;

    }

}
```

可以看到，我们加锁就一行代码：jedis.set(String key, String value, String nxxx, String expx, int time)，这个set()方法一共有五个形参：

第一个为key，我们使用key来当锁，因为key是唯一的。

第二个为value，我们传的是requestId，很多童鞋可能不明白，有key作为锁不就够了吗，为什么还要用到value？原因就是我们在上面讲到可靠性时，分布式锁要满足第四个条件解铃还须系铃人，通过给value赋值为requestId，我们就知道这把锁是哪个请求加的了，在解锁的时候就可以有依据。requestId可以使用UUID.randomUUID().toString()方法生成。

第三个为nxxx，这个参数我们填的是NX，意思是SET IF NOT EXIST，即当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；

第四个为expx，这个参数我们传的是PX，意思是我们要给这个key加一个过期的设置，具体时间由第五个参数决定。

第五个为time，与第四个参数相呼应，代表key的过期时间。

总的来说，执行上面的set()方法就只会导致两种结果：1. 当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端。2. 已有锁存在，不做任何操作。
心细的童鞋就会发现了，我们的加锁代码满足我们可靠性里描述的三个条件。首先，set()加入了NX参数，可以保证如果已有key存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即key被删除），不会发生死锁。最后，因为我们将value赋值为requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。由于我们只考虑Redis单机部署的场景，所以容错性我们暂不考虑。
错误示例1
比较常见的错误示例就是使用jedis.setnx()和jedis.expire()组合实现加锁，代码如下：

```
public static void wrongGetLock1(Jedis jedis, String lockKey, String requestId, int expireTime) {

    Long result = jedis.setnx(lockKey, requestId);
    if (result == 1) {
        // 若在这里程序突然崩溃，则无法设置过期时间，将发生死锁        jedis.expire(lockKey, expireTime);
    }

}
```

setnx()方法作用就是SET IF NOT EXIST，expire()方法就是给锁加一个过期时间。乍一看好像和前面的set()方法结果一样，然而由于这是两条Redis命令，不具有原子性，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。网上之所以有人这样实现，是因为低版本的jedis并不支持多参数的set()方法。
错误示例2

```
public static boolean wrongGetLock2(Jedis jedis, String lockKey, int expireTime) {

    long expires = System.currentTimeMillis() + expireTime;
    String expiresStr = String.valueOf(expires);

    // 如果当前锁不存在，返回加锁成功
    if (jedis.setnx(lockKey, expiresStr) == 1) {
        return true;
    }

    // 如果锁存在，获取锁的过期时间
    String currentValueStr = jedis.get(lockKey);
    if (currentValueStr != null && Long.parseLong(currentValueStr) < System.currentTimeMillis()) {
        // 锁已过期，获取上一个锁的过期时间，并设置现在锁的过期时间
        String oldValueStr = jedis.getSet(lockKey, expiresStr);
        if (oldValueStr != null && oldValueStr.equals(currentValueStr)) {
            // 考虑多线程并发的情况，只有一个线程的设置值和当前值相同，它才有权利加锁
            return true;
        }
    }

    // 其他情况，一律返回加锁失败
    return false;

}
```

这一种错误示例就比较难以发现问题，而且实现也比较复杂。实现思路：使用jedis.setnx()命令实现加锁，其中key是锁，value是锁的过期时间。执行过程：1. 通过setnx()方法尝试加锁，如果当前锁不存在，返回加锁成功。2. 如果锁已经存在则获取锁的过期时间，和当前时间比较，如果锁已经过期，则设置新的过期时间，返回加锁成功。代码如下：

那么这段代码问题在哪里？1. 由于是客户端自己生成过期时间，所以需要强制要求分布式下每个客户端的时间必须同步。 2. 当锁过期的时候，如果多个客户端同时执行jedis.getSet()方法，那么虽然最终只有一个客户端可以加锁，但是这个客户端的锁的过期时间可能被其他客户端覆盖。3. 锁不具备拥有者标识，即任何客户端都可以解锁。
解锁代码
正确姿势
还是先展示代码，再带大家慢慢解释为什么这样实现：

```
public class RedisTool {

    private static final Long RELEASE_SUCCESS = 1L;

    /**
     * 释放分布式锁
     * @param jedis Redis客户端
     * @param lockKey 锁
     * @param requestId 请求标识
     * @return 是否释放成功
     */
    public static boolean releaseDistributedLock(Jedis jedis, String lockKey, String requestId) {

        String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1])
else return 0 end";
        Object result = jedis.eval(script, Collections.singletonList(lockKey),
Collections.singletonList(requestId));

        if (RELEASE_SUCCESS.equals(result)) {
            return true;
        }
        return false;

    }

}
```

可以看到，我们解锁只需要两行代码就搞定了！第一行代码，我们写了一个简单的Lua脚本代码，上一次见到这个编程语言还是在《黑客与画家》里，没想到这次居然用上了。第二行代码，我们将Lua代码传到jedis.eval()方法里，并使参数KEYS[1]赋值为lockKey，ARGV[1]赋值为requestId。eval()方法是将Lua代码交给Redis服务端执行。
那么这段Lua代码的功能是什么呢？其实很简单，首先获取锁对应的value值，检查是否与requestId相等，如果相等则删除锁（解锁）。那么为什么要使用Lua语言来实现呢？因为要确保上述操作是原子性的。关于非原子性会带来什么问题，可以阅读【解锁代码-错误示例2】 。那么为什么执行eval()方法可以确保原子性，源于Redis的特性，下面是官网对eval命令的部分解释：

简单来说，就是在eval命令执行Lua代码的时候，Lua代码将被当成一个命令去执行，并且直到eval命令执行完成，Redis才会执行其他命令。
错误示例1
最常见的解锁代码就是直接使用jedis.del()方法删除锁，这种不先判断锁的拥有者而直接解锁的方式，会导致任何客户端都可以随时进行解锁，即使这把锁不是它的。

```
public static void wrongReleaseLock1(Jedis jedis, String lockKey) {
    jedis.del(lockKey);
}
```

错误示例2
这种解锁代码乍一看也是没问题，甚至我之前也差点这样实现，与正确姿势差不多，唯一区别的是分成两条命令去执行，代码如下：

```
public static void wrongReleaseLock2(Jedis jedis, String lockKey, String requestId) {

    // 判断加锁与解锁是不是同一个客户端
    if (requestId.equals(jedis.get(lockKey))) {
        // 若在此时，这把锁突然不是这个客户端的，则会误解锁        jedis.del(lockKey);
    }

}
```

如代码注释，问题在于如果调用jedis.del()方法的时候，这把锁已经不属于当前客户端的时候会解除他人加的锁。那么是否真的有这种场景？答案是肯定的，比如客户端A加锁，一段时间之后客户端A解锁，在执行jedis.del()之前，锁突然过期了，此时客户端B尝试加锁成功，然后客户端A再执行del()方法，则将客户端B的锁给解除了。

RabbitMQ 使用场景

场景 1：单发送单接收

使用场景：简单的发送与接收，没有特别的处理。

Producer:

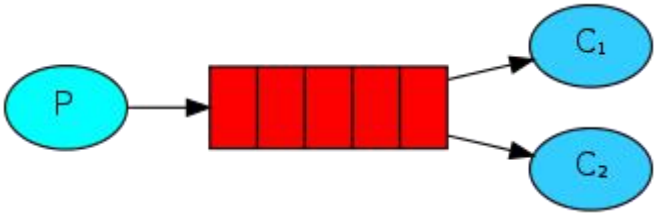
```
import com.rabbitmq.client.ConnectionFactory;import com.rabbitmq.client.Connection;import
com.rabbitmq.client.Channel;
public class Send {

  private final static String QUEUE_NAME = "hello";

  public static void main(String[] argv) throws Exception {

    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    String message = "Hello World!";
    channel.basicPublish("", QUEUE_NAME, null, message.getBytes());
    System.out.println(" [x] Sent '" + message + "'");

    channel.close();
    connection.close();
  }
}
```

Consumer:

```
import com.rabbitmq.client.ConnectionFactory;import com.rabbitmq.client.Connection;import
com.rabbitmq.client.Channel;import com.rabbitmq.client.QueueingConsumer;
public class Recv {

    private final static String QUEUE_NAME = "hello";

    public static void main(String[] argv) throws Exception {

    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.queueDeclare(QUEUE_NAME, false, false, false, null);
    System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

    QueueingConsumer consumer = new QueueingConsumer(channel);
    channel.basicConsume(QUEUE_NAME, true, consumer);

    while (true) {
      QueueingConsumer.Delivery delivery = consumer.nextDelivery();
      String message = new String(delivery.getBody());
      System.out.println(" [x] Received '" + message + "'");
    }
  }
}
```

场景 2：单发送多接收

使用场景：一个发送端，多个接收端，如分布式的任务派发。为了保证消息发送的可靠性，不丢失消息，使消息持久化了。同时为了防止接收端在处理消息时down掉，只有在消息处理完成后才发送ack消息。



Producer:

```
import com.rabbitmq.client.ConnectionFactory;import com.rabbitmq.client.Connection;import
com.rabbitmq.client.Channel;import com.rabbitmq.client.MessageProperties;
public class NewTask {

  private static final String TASK_QUEUE_NAME = "task_queue";

  public static void main(String[] argv) throws Exception {

    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.queueDeclare(TASK_QUEUE_NAME, true, false, false, null);

    String message = getMessage(argv);

    channel.basicPublish( "", TASK_QUEUE_NAME,
              MessageProperties.PERSISTENT_TEXT_PLAIN,
              message.getBytes());
    System.out.println(" [x] Sent '" + message + "'");

    channel.close();
    connection.close();
  }

  private static String getMessage(String[] strings){
    if (strings.length < 1)
      return "Hello World!";
    return joinStrings(strings, " ");
  }

  private static String joinStrings(String[] strings, String delimiter) {
    int length = strings.length;
    if (length == 0) return "";
    StringBuilder words = new StringBuilder(strings[0]);
    for (int i = 1; i < length; i++) {
      words.append(delimiter).append(strings[i]);
    }
    return words.toString();
  }
}
```

发送端和场景1不同点：

1、使用“task_queue”声明了另一个Queue，因为RabbitMQ不容许声明2个相同名称、配置不同的Queue

2、使"task_queue"的Queue的durable的属性为true，即使消息队列durable

3、使用MessageProperties.PERSISTENT_TEXT_PLAIN使消息durable

When RabbitMQ quits or crashes it will forget the queues and messages unless you tell it not to. Two things are required to make sure that messages aren't lost: we need to mark both the queue and messages as durable.

Consumer:

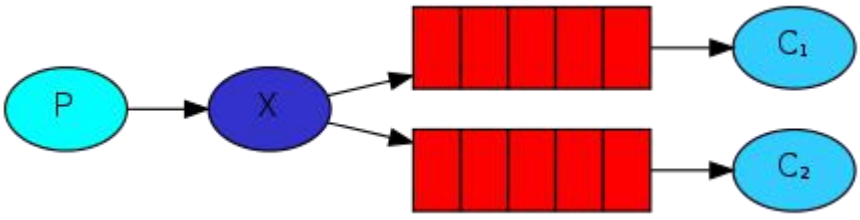
```
import com.rabbitmq.client.ConnectionFactory;import com.rabbitmq.client.Connection;import
com.rabbitmq.client.Channel;import com.rabbitmq.client.QueueingConsumer;
  public class Worker {

  private static final String TASK_QUEUE_NAME = "task_queue";
```

```
public static void main(String[] argv) throws Exception {

  ConnectionFactory factory = new ConnectionFactory();
  factory.setHost("localhost");
  Connection connection = factory.newConnection();
  Channel channel = connection.createChannel();

  channel.queueDeclare(TASK_QUEUE_NAME, true, false, false, null);
  System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

  channel.basicQos(1);

  QueueingConsumer consumer = new QueueingConsumer(channel);
  channel.basicConsume(TASK_QUEUE_NAME, false, consumer);

  while (true) {
    QueueingConsumer.Delivery delivery = consumer.nextDelivery();
    String message = new String(delivery.getBody());

    System.out.println(" [x] Received '" + message + "'");
    doWork(message);
    System.out.println(" [x] Done");

    channel.basicAck(delivery.getEnvelope().getDeliveryTag(), false);
  }
}

private static void doWork(String task) throws InterruptedException {
  for (char ch: task.toCharArray()) {
    if (ch == '.') Thread.sleep(1000);
  }
}
}
```

接收端和场景1不同点：
1、使用“task_queue”声明消息队列，并使消息队列durable
2、在使用channel.basicConsume接收消息时使autoAck为false，即不自动会发ack，由channel.basicAck()在消息处理完成后发送消息。
3、使用了channel.basicQos(1)保证在接收端一个消息没有处理完时不会接收另一个消息，即接收端发送了ack后才会接收下一个消息。在这种情况下发送端会尝试把消息发送给下一个not busy的接收端。
注意点：
1）It's a common mistake to miss the basicAck. It's an easy error, but the consequences are serious. Messages will be redelivered when your client quits (which may look like random redelivery), but RabbitMQ will eat more and more memory as it won't be able to release any unacked messages.
2）Note on message persistence
Marking messages as persistent doesn't fully guarantee that a message won't be lost. Although it tells RabbitMQ to save the message to disk, there is still a short time window when RabbitMQ has accepted a message and hasn't saved it yet. Also, RabbitMQ doesn't do fsync(2) for every message -- it may be just saved to cache and not really written to the disk. The persistence guarantees aren't strong, but it's more than enough for our simple task queue. If you need a stronger guarantee you can wrap the publishing code in a transaction.
3）Note about queue size
If all the workers are busy, your queue can fill up. You will want to keep an eye on that, and maybe add more workers, or have some other strategy.
4）RabbitMQ allows you to set Time To Live for both messages and queues. https://www.rabbitmq.com/ttl.html

场景 3：Publish/Subscribe

使用场景：发布、订阅模式，发送端发送广播消息，多个接收端接收。



Producer:

```
import com.rabbitmq.client.ConnectionFactory;import com.rabbitmq.client.Connection;import
com.rabbitmq.client.Channel;
public class EmitLog {

  private static final String EXCHANGE_NAME = "logs";

  public static void main(String[] argv) throws Exception {

    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.exchangeDeclare(EXCHANGE_NAME, "fanout");

    String message = getMessage(argv);

    channel.basicPublish(EXCHANGE_NAME, "", null, message.getBytes());
    System.out.println(" [x] Sent '" + message + "'");

    channel.close();
    connection.close();
  }

  private static String getMessage(String[] strings){
    if (strings.length < 1)
            return "info: Hello World!";
    return joinStrings(strings, " ");
  }

  private static String joinStrings(String[] strings, String delimiter) {
    int length = strings.length;
    if (length == 0) return "";
    StringBuilder words = new StringBuilder(strings[0]);
    for (int i = 1; i < length; i++) {
        words.append(delimiter).append(strings[i]);
    }
    return words.toString();
  }
}
```

发送端：
发送消息到一个名为“logs”的exchange上，使用“fanout”方式发送，即广播消息，不需要使用queue，发送端不需要关心谁接收。
Consumer:

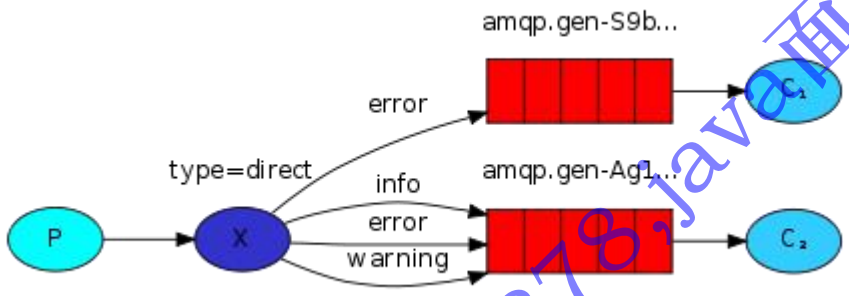
```
import com.rabbitmq.client.ConnectionFactory;import com.rabbitmq.client.Connection;import
com.rabbitmq.client.Channel;import com.rabbitmq.client.QueueingConsumer;
public class ReceiveLogs {

  private static final String EXCHANGE_NAME = "logs";

  public static void main(String[] argv) throws Exception {

    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.exchangeDeclare(EXCHANGE_NAME, "fanout");
    String queueName = channel.queueDeclare().getQueue();
    channel.queueBind(queueName, EXCHANGE_NAME, "");

    System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

    QueueingConsumer consumer = new QueueingConsumer(channel);
    channel.basicConsume(queueName, true, consumer);
```

```
    while (true) {
      QueueingConsumer.Delivery delivery = consumer.nextDelivery();
      String message = new String(delivery.getBody());

      System.out.println(" [x] Received '" + message + "'");
    }
  }
}
```

接收端：
1、声明名为“logs”的exchange的，方式为"fanout"，和发送端一样。
2、channel.queueDeclare().getQueue();该语句得到一个随机名称的Queue，该queue的类型为non-durable、exclusive、auto-delete的，将该queue绑定到上面的exchange上接收消息。
3、注意binding queue的时候，channel.queueBind()的第三个参数Routing key为空，即所有的消息都接收。如果这个值不为空，在exchange type为“fanout”方式下该值被忽略！

场景 4：Routing (按路线发送接收)

使用场景：发送端按 routing key 发送消息，不同的接收端按不同的 routing key 接收消息。



Producer:

```
import com.rabbitmq.client.ConnectionFactory;import com.rabbitmq.client.Connection;import
com.rabbitmq.client.Channel;
public class EmitLogDirect {

  private static final String EXCHANGE_NAME = "direct_logs";

  public static void main(String[] argv) throws Exception {

    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.exchangeDeclare(EXCHANGE_NAME, "direct");

    String severity = getSeverity(argv);
    String message = getMessage(argv);

    channel.basicPublish(EXCHANGE_NAME, severity, null, message.getBytes());
    System.out.println(" [x] Sent '" + severity + "':'" + message + "'");

    channel.close();
    connection.close();
  }

  private static String getSeverity(String[] strings){
    if (strings.length < 1)
          return "info";
    return strings[0];
  }

  private static String getMessage(String[] strings){
```

```
    if (strings.length < 2)
           return "Hello World!";
    return joinStrings(strings, " ", 1);
  }

  private static String joinStrings(String[] strings, String delimiter, int startIndex) {
    int length = strings.length;
    if (length == 0 ) return "";
    if (length < startIndex ) return "";
    StringBuilder words = new StringBuilder(strings[startIndex]);
    for (int i = startIndex + 1; i < length; i++) {
        words.append(delimiter).append(strings[i]);
    }
    return words.toString();
  }
}
```

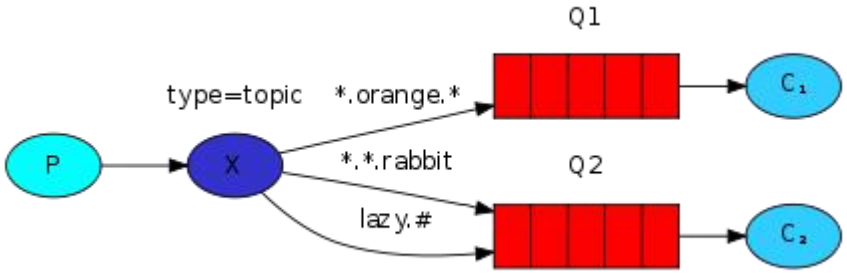
发送端和场景3的区别：
1、exchange的type为direct
2、发送消息的时候加入了routing key
Consumer：

```
import com.rabbitmq.client.ConnectionFactory;import com.rabbitmq.client.Connection;import
com.rabbitmq.client.Channel;import com.rabbitmq.client.QueueingConsumer;
public class ReceiveLogsDirect {

  private static final String EXCHANGE_NAME = "direct_logs";

  public static void main(String[] argv) throws Exception {

    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost("localhost");
    Connection connection = factory.newConnection();
    Channel channel = connection.createChannel();

    channel.exchangeDeclare(EXCHANGE_NAME, "direct");
    String queueName = channel.queueDeclare().getQueue();

    if (argv.length < 1){
      System.err.println("Usage: ReceiveLogsDirect [info] [warning] [error]");
      System.exit(1);
    }

    for(String severity : argv){
      channel.queueBind(queueName, EXCHANGE_NAME, severity);
    }

    System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

    QueueingConsumer consumer = new QueueingConsumer(channel);
    channel.basicConsume(queueName, true, consumer);

    while (true) {
      QueueingConsumer.Delivery delivery = consumer.nextDelivery();
      String message = new String(delivery.getBody());
      String routingKey = delivery.getEnvelope().getRoutingKey();

      System.out.println(" [x] Received '" + routingKey + "':'" + message + "'");
    }
  }
}
```

接收端和场景3的区别：
在绑定queue和exchange的时候使用了routing key，即从该exchange上只接收routing key指定的消息。

场景 5：Topics (按 topic 发送接收)

使用场景：发送端不只按固定的routing key发送消息，而是按字符串“匹配”发送，接收端同样如此。



Producer:

```
import com.rabbitmq.client.ConnectionFactory;import com.rabbitmq.client.Connection;import
com.rabbitmq.client.Channel;
public class EmitLogTopic {

  private static final String EXCHANGE_NAME = "topic_logs";

  public static void main(String[] argv) {
    Connection connection = null;
    Channel channel = null;
    try {
      ConnectionFactory factory = new ConnectionFactory();
      factory.setHost("localhost");

      connection = factory.newConnection();
      channel = connection.createChannel();

      channel.exchangeDeclare(EXCHANGE_NAME, "topic");

      String routingKey = getRouting(argv);
      String message = getMessage(argv);

      channel.basicPublish(EXCHANGE_NAME, routingKey, null, message.getBytes());
      System.out.println(" [x] Sent '" + routingKey + "':'" + message + "'");

    }
    catch  (Exception e) {
      e.printStackTrace();
    }
    finally {
      if (connection != null) {
        try {
          connection.close();
        }
        catch (Exception ignore) {}
      }
    }
  }

  private static String getRouting(String[] strings){
    if (strings.length < 1)
          return "anonymous.info";
    return strings[0];
  }

  private static String getMessage(String[] strings){
    if (strings.length < 2)
          return "Hello World!";
    return joinStrings(strings, " ", 1);
  }

  private static String joinStrings(String[] strings, String delimiter, int startIndex) {
    int length = strings.length;
    if (length == 0 ) return "";
```

```
    if (length < startIndex ) return "";
    StringBuilder words = new StringBuilder(strings[startIndex]);
    for (int i = startIndex + 1; i < length; i++) {
        words.append(delimiter).append(strings[i]);
    }
    return words.toString();
  }
}
```

发送端和场景4的区别：
1、exchange的type为topic
2、发送消息的routing key不是固定的单词，而是匹配字符串，如"*.lu.#"，*匹配一个单词，#匹配0个或多个单词。
Consumer：

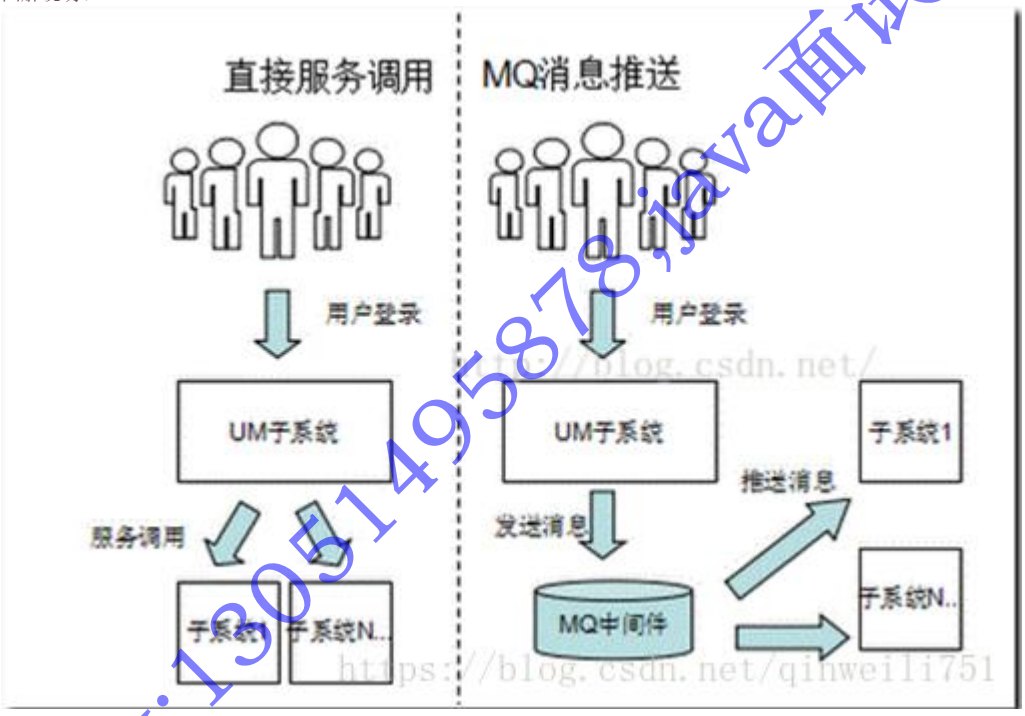
```
import com.rabbitmq.client.ConnectionFactory;import com.rabbitmq.client.Connection;import
com.rabbitmq.client.Channel;import com.rabbitmq.client.QueueingConsumer;
public class ReceiveLogsTopic {

  private static final String EXCHANGE_NAME = "topic_logs";

  public static void main(String[] argv) {
    Connection connection = null;
    Channel channel = null;
    try {
      ConnectionFactory factory = new ConnectionFactory();
      factory.setHost("localhost");

      connection = factory.newConnection();
      channel = connection.createChannel();

      channel.exchangeDeclare(EXCHANGE_NAME, "topic");
      String queueName = channel.queueDeclare().getQueue();

      if (argv.length < 1){
        System.err.println("Usage: ReceiveLogsTopic [binding_key]...");
        System.exit(1);
      }

      for(String bindingKey : argv){
        channel.queueBind(queueName, EXCHANGE_NAME, bindingKey);
      }

      System.out.println(" [*] Waiting for messages. To exit press CTRL+C");

      QueueingConsumer consumer = new QueueingConsumer(channel);
      channel.basicConsume(queueName, true, consumer);

      while (true) {
        QueueingConsumer.Delivery delivery = consumer.nextDelivery();
        String message = new String(delivery.getBody());
        String routingKey = delivery.getEnvelope().getRoutingKey();

        System.out.println(" [x] Received '" + routingKey + "':'" + message + "'");
      }
    }
    catch  (Exception e) {
      e.printStackTrace();
    }
    finally {
      if (connection != null) {
        try {
          connection.close();
        }
        catch (Exception ignore) {}
      }
    }
  }
}
```

ActiveMQ 的作用总结(应用场景及优势)

ActiveMQ 编辑

ActiveMQ 是Apache出品，最流行的，能力强劲的开源消息总线。ActiveMQ 是一个完全支持JMS1.1和J2EE 1.4规范的 JMS Provider实现，尽管JMS规范出台已经是很久的事情了，但是JMS在当今的J2EE应用中间仍然扮演着特殊的地位。

业务场景说明：
消息队列在大型电子商务类网站，如京东、淘宝、去哪儿等网站有着深入的应用，
队列的主要作用是消除高并发访问高峰，加快网站的响应速度。
在不使用消息队列的情况下，用户的请求数据直接写入数据库，在高并发的情况下，会对数据库造成巨大的压力，同时也使得系统响应延迟加剧。
在使用队列后，用户的请求发给队列后立即返回，
（例如：当然不能直接给用户提示订单提交成功，京东上提示：您“您提交了订单，请等待系统确认”），
再由消息队列的消费者进程从消息队列中获取数据，异步写入数据库。
由于消息队列的服务处理速度远快于数据库，因此用户的响应延迟可得到有效改善。
图解说明：



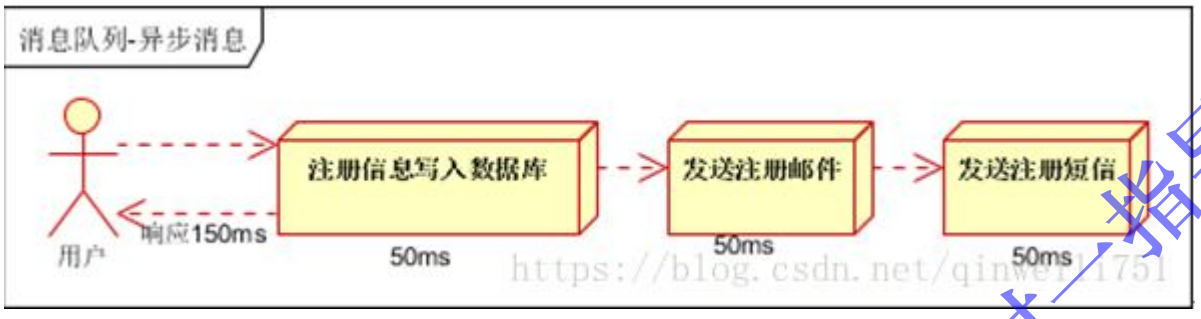
1. 消息队列说明
消息队列中间件是分布式系统中重要的组件，主要解决应用耦合，异步消息，流量削锋等问题。
实现高性能，高可用，可伸缩和最终一致性架构。是大型分布式系统不可缺少的中间件。
目前在生产环境，使用较多的消息队列有ActiveMQ，RabbitMQ，ZeroMQ，Kafka，MetaMQ，RocketMQ等。

2. 消息队列应用场景
消息队列在实际应用中常用的使用场景。异步处理，应用解耦，流量削锋和消息通讯四个场景。

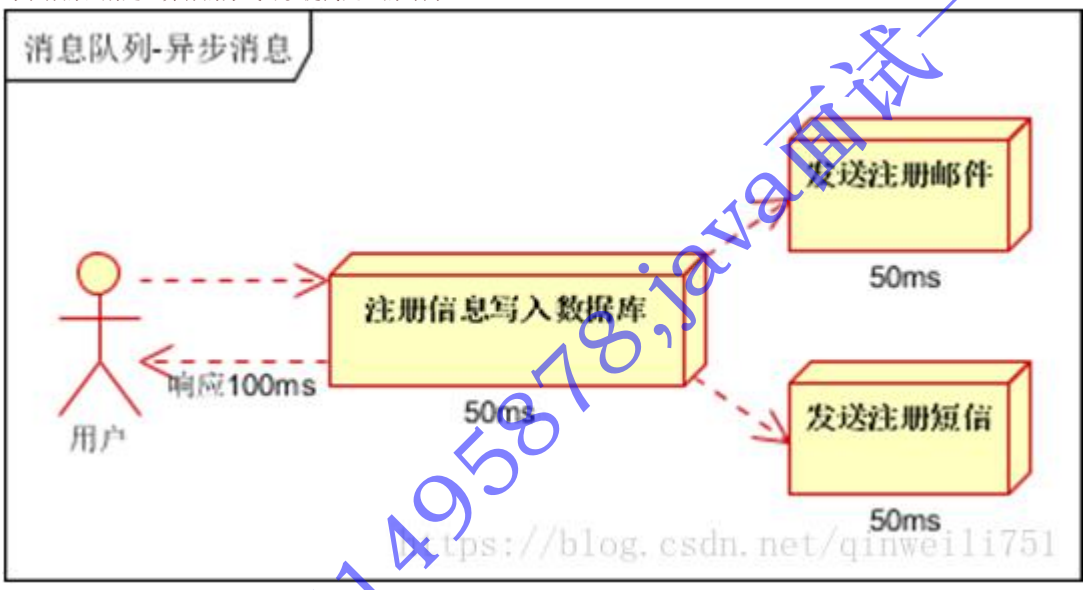
2.1. 异步处理

场景说明：用户注册后，需要发注册邮件和注册短信。传统的做法有两种1.串行的方式；2.并行方式。

（1）串行方式：将注册信息写入数据库成功后，发送注册邮件，再发送注册短信。以上三个任务全部完成后，返回给客户端。

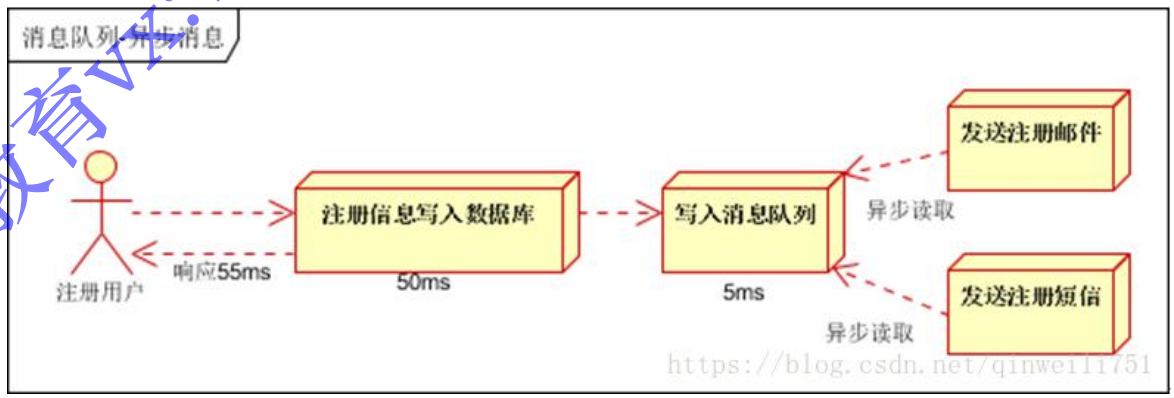


（2）并行方式：将注册信息写入数据库成功后，发送注册邮件的同时，发送注册短信。以上三个任务完成后，返回给客户端。
与串行的差别是，并行的方式可以提高处理的时间。



假设三个业务节点每个使用50毫秒钟，不考虑网络等其他开销，则串行方式的时间是150毫秒，并行的时间可能是100毫秒。
因为CPU在单位时间内处理的请求数是一定的，假设CPU1秒内吞吐量是100次。
则串行方式1秒内CPU可处理的请求量是7次（1000/150）。并行方式处理的请求量是10次（1000/100）。

小结：如以上案例描述，传统的方式系统的性能（并发量，吞吐量，响应时间）会有瓶颈。如何解决这个问题呢？
引入消息队列，将不是必须的业务逻辑，异步处理。改造后的架构如下：

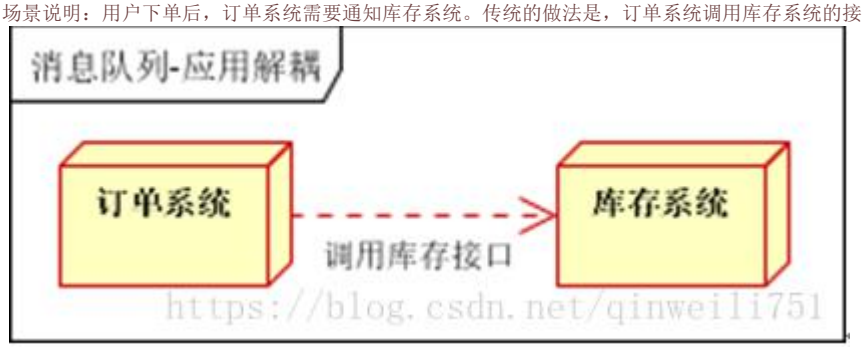


按照以上约定，用户的响应时间相当于是注册信息写入数据库的时间，也就是50毫秒。

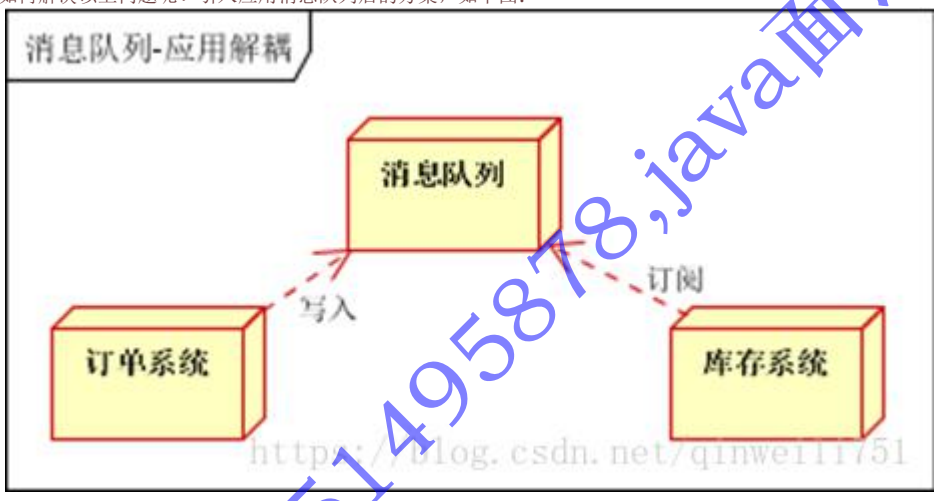
注册邮件，发送短信写入消息队列后，直接返回，因此写入消息队列的速度很快，基本可以忽略，
因此用户的响应时间可能是50毫秒。所以基于此架构改变后，系统的吞吐量提高到每秒20 QPS。比串行提高了3倍，比并行提高了两倍。

2.2. 应用解耦

场景说明：用户下单后，订单系统需要通知库存系统。传统的做法是，订单系统调用库存系统的接口。如下图：



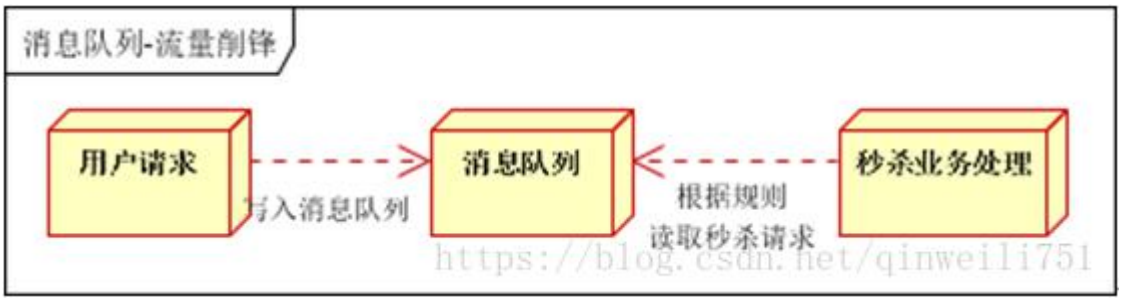
传统模式的缺点：
1） 假如库存系统无法访问，则订单减库存将失败，从而导致订单失败；
2） 订单系统与库存系统耦合；
如何解决以上问题呢？引入应用消息队列后的方案，如下图：



1:订单系统：用户下单后，订单系统完成持久化处理，将消息写入消息队列，返回用户订单下单成功,请等待物流配送。
2:库存系统：订阅下单的消息，采用拉/推的方式，获取下单信息，库存系统根据下单信息，进行库存操作。
3:假如：在下单时库存系统不能正常使用。也不影响正常下单，
因为下单后，订单系统写入消息队列就不再关心其他的后续操作了。实现订单系统与库存系统的应用解耦。

2.3. 流量削锋

流量削锋也是消息队列中的常用场景，一般在秒杀或团抢活动中使用广泛。
应用场景：秒杀活动，一般会因为流量过大，导致流量暴增，应用容易挂掉。为解决这个问题，一般需要在应用前端加入消息队列。
可以控制活动的人数.
可以缓解短时间内高流量压垮应用;



用户的请求，服务器接收后，首先写入消息队列。假如消息队列长度超过最大数量，则直接抛弃用户请求或跳转到错误页面；秒杀业务根据消息队列中的请求信息，再做后续处理。

2.4. 消息通讯

消息通讯是指，消息队列一般都内置了高效的通信机制，因此也可以用在纯的消息通讯。比如实现点对点消息队列，或者聊天室等。

点对点通讯：

客户端A和客户端B使用同一队列，进行消息通讯。

聊天室通讯：

客户端A，客户端B，客户端N订阅同一主题，进行消息发布和接收。实现类似聊天室效果。

以上实际是消息队列的两种消息模式，点对点或发布订阅模式。

Zookeeper 集群为什么是高可用的?

集群组成

要搭建一个高可用的 ZooKeeper 集群，我们首先需要确定好集群的规模。关于 ZooKeeper 集群的服务器组成，相信很多对 ZooKeeper 了解但是理解不够深入的读者，都存在或曾经存在过这样一个错误的认识：为了使得 ZooKeeper 集群能够顺利地选举出 Leader，必须将 ZooKeeper 集群的服务器数部署成奇数。这里我们需要澄清的一点是：任意台 ZooKeeper 服务器都能部署且能正常运行。

其实关于 ZooKeeper 集群服务器数，ZooKeeper 官方确实给出了关于奇数的建议，但绝大部分 ZooKeeper 用户对于这个建议认识有偏差。在本书前面提到的“过半存活即可用”特性中，我们已经了解了，一个 ZooKeeper 集群如果要对外提供可用的服务，那么集群中必须要有过半的机器正常工作并且彼此之间能够正常通信。基于这个特性，如果想搭建一个能够允许 N 台机器 down 掉的集群，那么就要部署一个由 2*N+1 台服务器构成的 ZooKeeper 集群。因此，一个由 3 台机器构成的 ZooKeeper 集群，能够在挂掉 1 台机器后依然正常工作，而对于一个由 5 台服务器构成的 ZooKeeper 集群，能够对 2 台机器挂掉的情况进行容灾。注意，如果是一个由6台服务器构成的 ZooKeeper 集群，同样只能够挂掉 2 台机器，因为如果挂掉 3 台，剩下的机器就无法实现过半了。

因此，从上面的讲解中，我们其实可以看出，对于一个由 6 台机器构成的 ZooKeeper 集群来说，和一个由 5 台机器构成的 ZooKeeper 集群，其在容灾能力上并没有任何显著的优势，反而多占用了一个服务器资源。基于这个原因，ZooKeeper 集群通常设计部署成奇数台服务器即可。

容灾

所谓容灾，在 IT 行业通常是指我们的计算机信息系统具有的一种在遭受诸如火灾、地震、断电和其他基础网络设备故障等毁灭性灾难的时候，依然能够对外提供可用服务的能力。

对于一些普通的应用，为了达到容灾标准，通常我们会选择在多台机器上进行部署来组成一个集群，这样即使在集群的一台或是若干台机器出现故障的情况下，整个集群依然能够对外提供可用的服务。

而对于一些核心应用，不仅要通过使用多台机器构建集群的方式来提供服务，而且还要将集群中的机器部署在两个机房，这样的话，即使其中一个机房遭遇灾难，依然能够对外提供可用的服务。

上面讲到的都是应用层面的容灾模式，那么对于 ZooKeeper 这种底层组件来说，如何进行容灾呢？讲到这里，可能多少读者会有疑问，ZooKeeper 既然已经解决了单点问题，那为什么还要进行容灾呢？

单点问题

单点问题是分布式环境中最常见也是最经典的问题之一，在很多分布式系统中都会存在这样的单点问题。具体地说，单点问题是指在一个分布式系统中，如果某一个组件出现故障就会引起整个系统的可用性大大下降甚至是处于瘫痪状态，那么我们就认为该组件存在单点问题。

ZooKeeper 确实已经很好地解决了单点问题。我们已经了解到，基于“过半”设计原则，ZooKeeper 在运行期间，集群中至少有过半的机器保存了最新的数据。因此，只要集群中超过半数的机器还能够正常工作，整个集群就能够对外提供服务。

容灾

解决了单点问题，是不是该考虑容灾了呢？答案是肯定的，在搭建一个高可用的集群的时候依然需要考虑容灾问题。正如上面讲到的，如果集群中超过半数的机器还在正常工作，集群就能够对外提供正常的服务。那么，如果整个机房出现灾难性的事故，这时候已经不是单点问题的范畴了。

在进行 ZooKeeper 的容灾方案设计过程中，我们要充分考虑到“过半原则”。也就是说，无论发生什么情况，我们必须保证 ZooKeeper 集群中有超过半数的机器能够正常工作。因此，通常有以下两种部署方案。

双机房部署

在进行容灾方案的设计时，我们通常是以机房为单位来考虑问题。在现实中，很多公司的机房规模并不大，因此双机房部署是个比较常见的方案。但是遗憾的是，在目前版本的 ZooKeeper 中，还没有办法能够在双机房条件下实现比较好的容灾效果——因为无论哪个机房发生异常情况，都有可能使得 ZooKeeper 集群中可用的机器无法超过半数。当然，在拥有两个机房的场景下，通常有一个机房是主要机房（一般而言，公司会花费更多的钱去租用一个稳定性更好、设备更可靠的机房，这个机房就是主要机房，而另外一个机房则更加廉价一些）。我们唯一能做的，就是尽量在主要机房部署更多的机器。例如，对于一个由 7 台机器组成的 ZooKeeper 集群，通常在主要机房中部署 4 台机器，剩下的 3 台机器部署到另外一个机房中。

三机房部署

既然在双机房部署模式下并不能实现好的容灾效果，那么对于有条件的公司，选择三机房部署无疑是个更好的选择，无论哪个机房发生了故障，剩下两个机房的机器数量都超过半数。假如我们有三个机房可以部署服务，并且这三个机房间的网络状况良好，那么就可以在三个机房中都部署若干个机器来组成一个 ZooKeeper 集群。

我们假定构成 ZooKeeper 集群的机器总数为 N，在三个机房中部署的 ZooKeeper 服务器数分别为 N1、N2 和 N3，如果要使该 ZooKeeper 集群具有较好的容灾能力，我们可以根据如下算法来计算 ZooKeeper 集群的机器部署方案。

1.）计算 N1

如果 ZooKeeper 集群的服务器总数是 N，那么：
N1 = (N-1)/2
在 Java 中，“/” 运算符会自动对计算结果向下取整操作。举个例子，如果 N=8，那么 N1=3；如果 N=7，那么 N1 也等于 3。
2.）计算 N2 的可选值
N2 的计算规则和 N1 非常类似，只是 N2的取值是在一个取值范围内：
N2 的取值范围是 1~(N-N1)/2
即如果 N=8，那么 N1=3，则 N2 的取值范围就是 1~2，分别是 1 和 2。注意，1 和 2 仅仅是 N2 的可选值，并非最终值——如果 N2 为某个可选值的时候，无法计算出 N3 的值，那么该可选值也无效。
3.） 计算 N3，同时确定 N2 的值
很显然，现在只剩下 N3 了，可以简单的认为 N3 的取值就是剩下的机器数，即：
N3 = N - N1 - N2
只是 N3 的取值必须满足 N3 < N1+N2。在满足这个条件的基础下，我们遍历步骤 2 中计算得到的 N2 的可选值，即可得到三机房部署时每个机房的服务器数量了。
现在我们以 7 台机器为例，来看看如何分配三机房的机器分布。根据算法的步骤 1，我们首先确定 N1 的取值为 3。根据算法的步骤 2，我们确定了 N2 的可选值为 1 和 2。最后根据步骤 3，我们遍历 N2 的可选值，即可得到两种部署方案，分别是 (3,1,3) 和 (3,2,2)。以下是 Java 程序代码对以上算法的一种简单实现：

```
public class Allocation {

    static final int n = 7;
    public static void main(String[] args){
        int n1,n2,n3;
        n1 = (n-1) / 2;
        int n2_max = (n-n1) / 2;
        for(int i=1; i<=n2_max; i++){
            n2 = i;
            n3 = n - n1 -n2;
            if(n3 >= (n1+n2)){
                continue;
            }
            System.out.println("("+n1+","+n2+","+n3+")");
        }
    }
}
```

水平扩容

水平可扩容可以说是对一个分布式系统在高可用性方面提出的基本的，也是非常重要的一个要求，通过水平扩容能够帮助系统在不进行或进行极少改进工作的前提下，快速提高系统对外的服务支撑能力。简单地讲，水平扩容就是向集群中添加更多的机器，以提高系统的服务质量。
很遗憾的是，ZooKeeper 在水平扩容扩容方面做得并不十分完美，需要进行整个集群的重启。通常有两种重启方式，一种是集群整体重启，另外一种是逐台进行服务器的重启。

（1）整体重启

所谓集群整体重启，就是先将整个集群停止，然后更新 ZooKeeper 的配置，然后再次启动。如果在你的系统中，ZooKeeper 并不是个非常核心的组件，并且能够允许短暂的服务停止（通常是几秒钟的时间间隔），那么不妨选择这种方式。在整体重启的过程中，所有该集群的客户端都无法连接上集群。等到集群再次启动，这些客户端就能够自动连接上——注意，整体启动前建立起的客户端会话，并不会因为此次整体重启而失效。也就是说，在整体重启期间花费的时间将不计入会话超时时间的计算中。

（2）逐台重启

这种方式更适合绝大多数的实际场景。在这种方式中，每次仅仅重启集群中的一台机器，然后逐台对整个集群中的机器进行重启操作。这种方式可以在重启期间依然保证集群对外的正常服务。

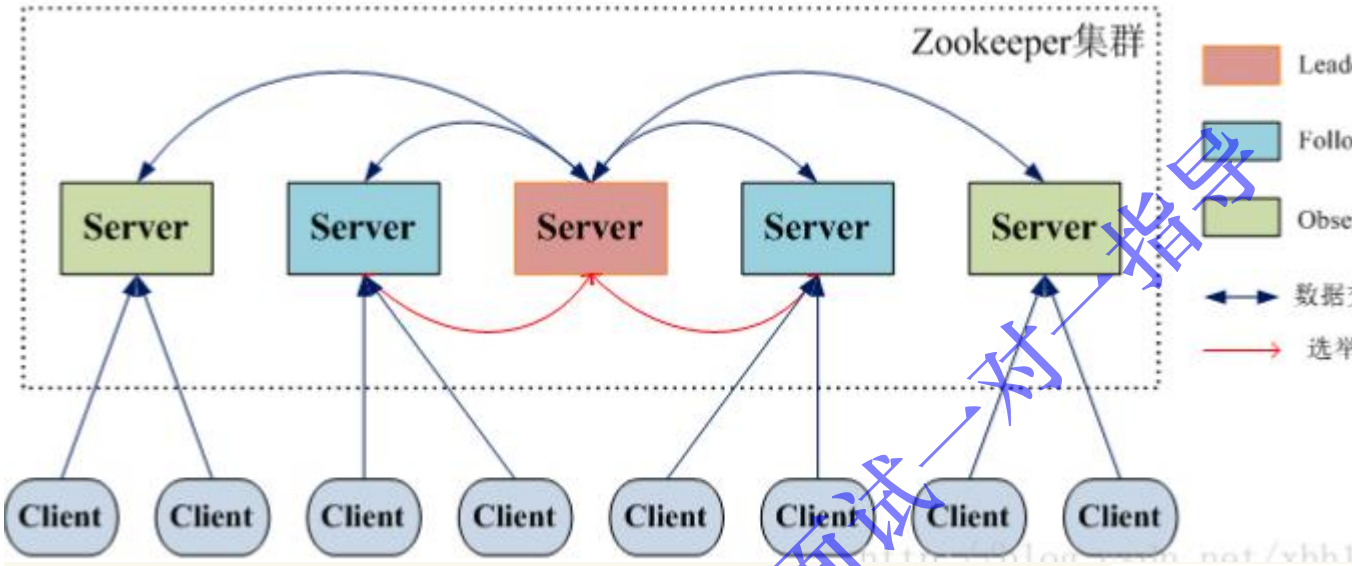
1、Zookeeper 的角色

» 领导者（leader），负责进行投票的发起和决议，更新系统状态

» 学习者（learner），包括跟随者（follower）和观察者（observer），follower 用于接受客户端请求并想客户端返回结果，在选主过程中参与投票

» Observer 可以接受客户端连接，将写请求转发给 leader，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



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。

- 为了保证事务的顺序一致性，zookeeper 采用了递增的事务 id 号（zxid）来标识事务。所有的提议（

proposal）都在被提出的时候加上了 zxid。实现中 zxid 是一个 64 位的数字，它高 32 位是 epoch 用来标识

leader 关系是否改变，每次一个 leader 被选出来，它都会有一个新的 epoch，标识当前属于那个 leader 的

统治时期。低 32 位用于递增计数。

- 每个 Server 在工作过程中有三种状态：

  LOOKING：当前 Server 不知道 leader 是谁，正在搜寻

  LEADING：当前 Server 即为选举出来的 leader

  FOLLOWING：leader 已经选举出来，当前 Server 与之同步

2、Zookeeper 的读写机制

» Zookeeper 是一个由多个 server 组成的集群

» 一个 leader，多个 follower

» 每个 server 保存一份数据副本

» 全局数据一致

» 分布式读写

» 更新请求转发，由 leader 实施

3、Zookeeper 的保证

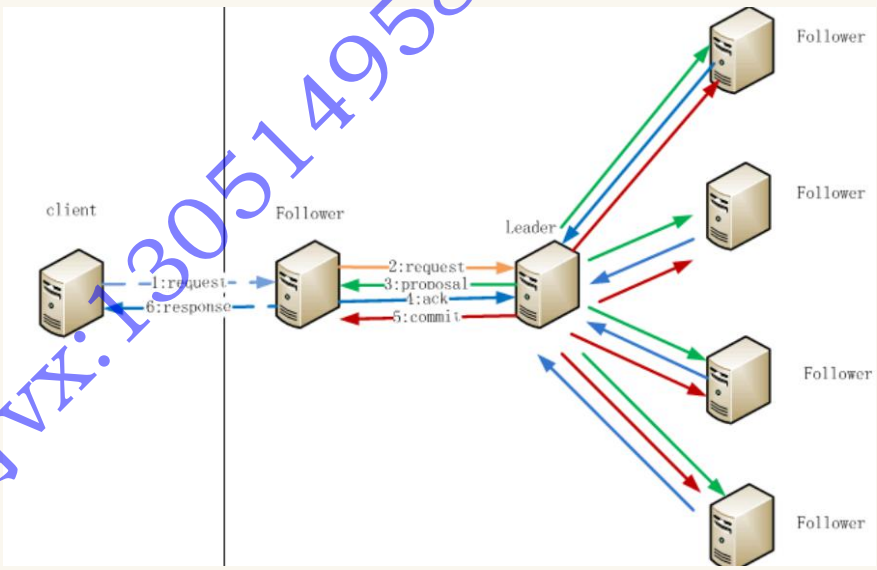
» 更新请求顺序进行，来自同一个 client 的更新请求按其发送顺序依次执行

» 数据更新原子性，一次数据更新要么成功，要么失败

» 全局唯一数据视图，client 无论连接到哪个 server，数据视图都是一致的

» 实时性，在一定事件范围内，client 能读到最新数据

4、Zookeeper 节点数据操作流程



注：1.在 Client 向 Follwer 发出一个写的请求

2.Follwer 把请求发送给 Leader

3.Leader 接收到以后开始发起投票并通知 Follwer 进行投票

4.Follwer 把投票结果发送给 Leader

5.Leader 将结果汇总后如果需要写入，则开始写入同时把写入操作通知给 Leader，然后 commit;

6.Follwer 把请求结果返回给 Client

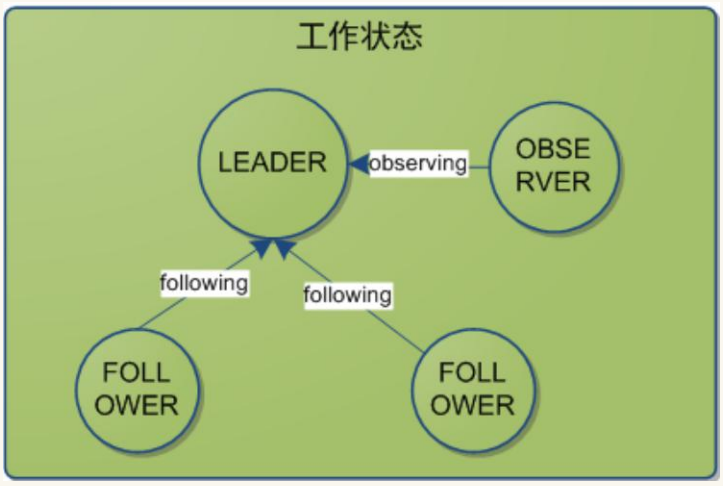
- Follower 主要有四个功能：
- 1. 向 Leader 发送请求（PING 消息、REQUEST 消息、ACK 消息、REVALIDATE 消息）；
- 2 .接收 Leader 消息并进行处理；
- 3 .接收 Client 的请求，如果为写请求，发送给 Leader 进行投票；
- 4 .返回 Client 结果。
- Follower 的消息循环处理如下几种来自 Leader 的消息：
- 1 .PING 消息： 心跳消息；
- 2 .PROPOSAL 消息：Leader 发起的提案，要求 Follower 投票；
- 3 .COMMIT 消息：服务器端最新一次提案的信息；
- 4 .UPTODATE 消息：表明同步完成；
- 5 .REVALIDATE 消息：根据 Leader 的 REVALIDATE 结果，关闭待 revalidate 的 session 还是允许其接受消息；
- 6 .SYNC 消息：返回 SYNC 结果到客户端，这个消息最初由客户端发起，用来强制得到最新的更新。

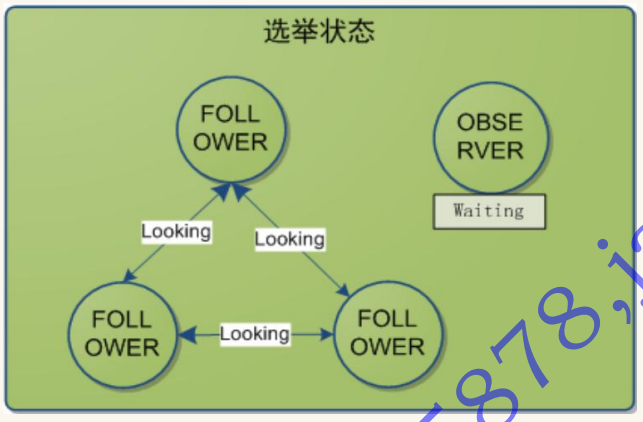
5、Zookeeper leader 选举

- 半数通过
  - 3 台机器 挂一台 2>3/2
  - 4 台机器 挂 2 台 2！>4/2

- A 提案说，我要选自己，B 你同意吗？C 你同意吗？B 说，我同意选 A；C 说，我同意选 A。(注意，这里超过半数了，其实在现实世界选举已经成功了。

  但是计算机世界是很严格，另外要理解算法，要继续模拟下去。)
- 接着 B 提案说，我要选自己，A 你同意吗；A 说，我已经超半数同意当选，你的提案无效；C 说，A 已经超半数同意当选，B 提案无效。
- 接着 C 提案说，我要选自己，A 你同意吗；A 说，我已经超半数同意当选，你的提案无效；B 说，A 已经超半数同意当选，C 的提案无效。
- 选举已经产生了 Leader，后面的都是 follower，只能服从 Leader 的命令。而且这里还有个小细节，就是其实谁先启动谁当头。





6、zxid

- znode 节点的状态信息中包含 czxid，那么什么是 zxid 呢?
- ZooKeeper 状态的每一次改变，都对应着一个递增的 Transaction id，该 id 称为 zxid. 由于 zxid 的递增性质，如果 zxid1 小于 zxid2，那么 zxid1 肯定先于 zxid2 发生.

创建任意节点，或者更新任意节点的数据，或者删除任意节点，都会导致 Zookeeper 状态发生改变，从而导致 zxid 的值增加.

7、Zookeeper 工作原理

» 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和广播模式。

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 的完成了和 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

» 一旦 leader 已经和多数的 follower 进行了状态同步后，他就可以开始广播消息了，即进入广播状态。这时候当一个 server 加入 zookeeper 服务中，它会在恢复模式下启动，

发现 leader，并和 leader 进行状态同步。待到同步结束，它也参与消息广播。Zookeeper 服务一直维持在 Broadcast 状态，直到 leader 崩溃了或者 leader 失去了大部分

的 followers 支持。

» 广播模式需要保证 proposal 被按顺序处理，因此 zk 采用了递增的事务 id 号(zxid)来保证。所有的提议(proposal)都在被提出的时候加上了 zxid。

实现中 zxid 是一个 64 为的数字，它高 32 位是 epoch 用来标识 leader 关系是否改变，每次一个 leader 被选出来，它都会有一个新的 epoch。低 32 位是个递增计数。

» 当 leader 崩溃或者 leader 失去大多数的 follower，这时候 zk 进入恢复模式，恢复模式需要重新选举出一个新的 leader，让所有的 server 都恢复到一个正确的状态。

» 每个 Server 启动以后都询问其它的 Server 它要投票给谁。
» 对于其他 server 的询问，server 每次根据自己的状态都回复自己推荐的 leader 的 id 和上一次处理事务的 zxid（系统启动时每个 server 都会推荐自己）
» 收到所有 Server 回复以后，就计算出 zxid 最大的哪个 Server，并将这个 Server 相关信息设置成下一次要投票的 Server。
» 计算这过程中获得票数最多的的 sever 为获胜者，如果获胜者的票数超过半数，则改 server 被选为 leader。否则，继续这个过程，直到 leader 被选举出来

» leader 就会开始等待 server 连接
» Follower 连接 leader，将最大的 zxid 发送给 leader
» Leader 根据 follower 的 zxid 确定同步点
» 完成同步后通知 follower 已经成为 uptodate 状态
» Follower 收到 uptodate 消息后，又可以重新接受 client 的请求进行服务了

8、数据一致性与 paxos 算法

• 据说 Paxos 算法的难理解与算法的知名度一样令人敬仰，所以我们先看如何保持数据的一致性，这里有个原则就是：
• 在一个分布式数据库系统中，如果各节点的初始状态一致，每个节点都执行相同的操作序列，那么他们最后能得到一个一致的状态。
• Paxos 算法解决的什么问题呢，解决的就是保证每个节点执行相同的操作序列。好吧，这还不简单，master 维护一个

全局写队列，所有写操作都必须 放入这个队列编号，那么无论我们写多少个节点，只要写操作是按编号来的，就能保证一

致性。没错，就是这样，可是如果 master 挂了呢。
• Paxos 算法通过投票来对写操作进行全局编号，同一时刻，只有一个写操作被批准，同时并发的写操作要去争取选票，

只有获得过半数选票的写操作才会被 批准（所以永远只会有一个写操作得到批准），其他的写操作竞争失败只好再发起一

轮投票，就这样，在日复一日年复一年的投票中，所有写操作都被严格编号排 序。编号严格递增，当一个节点接受了一个

编号为 100 的写操作，之后又接受到编号为 99 的写操作（因为网络延迟等很多不可预见原因），它马上能意识到自己 数据

不一致了，自动停止对外服务并重启同步过程。任何一个节点挂掉都不会影响整个集群的数据一致性（总 2n+1 台，除非挂掉大于 n 台）。

总结

• Zookeeper 作为 Hadoop 项目中的一个子项目，是 Hadoop 集群管理的一个必不可少的模块，它主要用来控制集群中的数据，

如它管理 Hadoop 集群中的 NameNode，还有 Hbase 中 Master Election、Server 之间状态同步等。\

关于 Paxos 算法可以查看文章 Zookeeper 全解析——Paxos 作为灵魂

9、Observer

• Zookeeper 需保证高可用和强一致性；
• 为了支持更多的客户端，需要增加更多 Server；
• Server 增多，投票阶段延迟增大，影响性能；
• 权衡伸缩性和高吞吐率，引入 Observer
• Observer 不参与投票；
• Observers 接受客户端的连接，并将写请求转发给 leader 节点；
• 加入更多 Observer 节点，提高伸缩性，同时不影响吞吐率

10、 为什么 zookeeper 集群的数目，一般为奇数个？

•Leader 选举算法采用了 Paxos 协议；
•Paxos 核心思想：当多数 Server 写成功，则任务数据写成功如果有 3 个 Server，则两个写成功即可；如果有 4 或 5 个 Server，则三个写成功即可。
•Server 数目一般为奇数（3、5、7）如果有 3 个 Server，则最多允许 1 个 Server 挂掉；如果有 4 个 Server，则同样最多允许 1 个 Server 挂掉由此，

我们看出 3 台服务器和 4 台服务器的的容灾能力是一样的，所以为了节省服务器资源，一般我们采用奇数个数，作为服务器部署个数。

11、Zookeeper 的数据模型

» 层次化的目录结构，命名符合常规文件系统规范
» 每个节点在 zookeeper 中叫做 znode,并且其有一个唯一的路径标识
» 节点 Znode 可以包含数据和子节点，但是 EPHEMERAL 类型的节点不能有子节点
» Znode 中的数据可以有多个版本，比如某一个路径下存有多个数据版本，那么查询这个路径下的数据就需要带上版本
» 客户端应用可以在节点上设置监视器
» 节点不支持部分读写，而是一次性完整读写

12、Zookeeper 的节点

» Znode 有两种类型，短暂的（ephemeral）和持久的（persistent）
» Znode 的类型在创建时确定并且之后不能再修改
» 短暂 znode 的客户端会话结束时，zookeeper 会将该短暂 znode 删除，短暂 znode 不可以有子节点
» 持久 znode 不依赖于客户端会话，只有当客户端明确要删除该持久 znode 时才会被删除

» Znode 有四种形式的目录节点
» PERSISTENT（持久的）
» EPHEMERAL(暂时的)
» PERSISTENT_SEQUENTIAL（持久化顺序编号目录节点）
» EPHEMERAL_SEQUENTIAL（暂时化顺序编号目录节点）

Docker 的优点：

Docker 五大优势：持续集成、版本控制、可移植性、隔离性和安全性