# Ioc控制反转

是一种编程思想。控制反转，控制是什么？控制是控制对象的生命周期。00传统程序是我们使用对象的时候直接new出来，而使用ioc后，对象全部被放进了ioc容器中。使用对象的时候直接从容器中获取，spring在初始化的时候，容器里的对象就已经被创建出来了（也就是默认是非懒加载），默认是单例的。实现原理是反射，反转是什么？反转是依赖对象的获取被反转了。原来我们主动控制对象，现在翻了过来，由springioc容器控制对象，我们需要什么对象，就由spring容器帮我们查找。

# 依赖注入：比如说我们操作数据库需要一7对象。以前是我们手动获取一个对象。有了spring之后，我们只需要告诉spring，需要一个connection对象。至于什么时候创建，怎么创建我们不

# 需要关心，spring会在适当的时候给我们一个对象，这样就完成·了对象间的依赖关系。依赖注入的原理是反射，它允许程序在运行的时候动态生成对象，执行对象的方法，改变对象的属性。

依赖注入的三种方式：set、注解（@Autowired和@Resource注解）、构造方法:

# Aop

Aop面向切面编程，是一种编程思想。在面向切面编程中，把业务逻辑分为主业务逻辑和交叉业务逻辑。主业务逻辑就是对数据的增删改操作。其他的业务，像是日志，事务管理，权限管理等都是交叉业务逻辑。在没有使用aop之前，主业务逻辑和交叉业务逻辑代码都是放在一块。如果需要修改，就需要修改代码里面的很多地方，非常不方便。把主业务逻辑和交叉业务逻辑分开后，分别进行开发，然后再把它们编织在一起，这就是aop，aop底层采用的是动态代理的方式。cglib

Aop优点，可以将那些与业务无关的代码，但是很多业务都调用的代码封装起来，在不改变原有代码的情况下增强原业务的功能。可以降低耦合度，减少重复代码，利于扩展，利于维护。

五种通知：前置，后置，正常返回，异常返回，环绕

**aop中相关名词的概念：**

连接点：插入在方法的某个地方，前面，后面等等，这就是连接点。

切点：所有连接点的集合，在代码中execution表达式就是切点

通知：增强原有方法的功能，增强了什么，在代码上@before标签里面的代码就是通知。也就是说，通知就是指定要干什么。

切面：切点+通知就是切面。在代码中，加上了aspect标签的类就是切面

# 分布式锁：

分布式锁用的是**redis实现分布式锁**，在代码中使用setnx命令。比如说火车票请求出票，当一个账号在多个终端登录的时候，同时请求出票，开启了多个线程。此时都会执行到setnx的命令，当其中一个线程抢到锁之后，会占用该锁，同时其他线程会被阻塞住，占到锁的线程执行出票的逻辑代码，请求出票。当出票“”完毕后就把锁释放掉，释放锁的代码写在finally语句中，保证线程一定会释放锁。我们还会给锁设置一个过期时间，如果真的不能释放锁，在超过一定时间后线程也会把锁释放掉。

# 分布式事务**：**

什么是分布式事务，就是一个事务的操作分布在不同的节点上，此时也保证在同一个事务下，操作的原子性，即事务的操作同时成功或同时失败。

## 分布式事务解决方案：2pc、3pc、tcc

# 2pc：

在2pc中有参与者和协调者，比如说在这事务中有两个参与者操作，同时要引入一个协调者。两个参与者都进行了update的操作，但是不会提交，把操作的结果发送给协调者。协调者收到成功的消息后，给参与者发送提交的请求，这时两个参与者才会提交事务。如果有一个参与者update失败，协调者不会给另一个已经成功update的参与者发送提交事务的请求，而是请求回滚。以此来保证事务的一致性。但是如果协调者此时宕机了，参与者一直收不到协调者的请求，就会一直占用着锁，占用资源。导致其他线程阻塞。这样并发性很低。

缺点：

阻塞：执行的过程中，所有参与者都不自动提交事务，导致线程一直占用锁，其他线程会被阻塞。 如果协调者宕机，参与者会一直占用锁，阻塞了其他线程

数据不一致：当协调者想参与者发送提交请求后，如果出现了网络问题，导致有的参与者收不到提 交请求，产生了数据不一致问题。3.00000

。

# 3pc：

3pc是在2pc的基础上引入了超时机制，当协调者宕机后，参与者如果一直收不到协调者的消息，会自动提交。但是这也会造成数据不一致的问题，如有在其中有的参与者操作失败需要进行回滚操作，但是另一部分完成了操作，进行了提交，这样就导致了数据不一致问题。

# Tcc：

我们常用的就是tcc事务：核心就是对每一个操作都注册一个与其对应的确认和补偿操作。

Tcc分为三步：第一步：执行try逻辑，完成所有业务检查(一致性)，预留必须业务资源(准隔离性)。如果try代码里面的逻辑没有问题。接着执行confirm逻辑，在confirm逻辑中，真正执行业务，不作任何业务检查。如果某个try逻辑出现了问题，就会让各个服务执行cancle逻辑，进行补偿操作，撤销之前的所有操作。

举例：

1.购买飞机票，一共要购买两张，一张中转票川航的，一张终点票东航的。在购买时必须要保证两张票同时购买成功。

2.如果两个都预留成功，就执行Confirm，confrim中发送购票请求。在这里，航空公司要确保购买时一定成功。

3.首先在try中分别向川航和东航发请求预留票，如果有其中一个预留失败，成功的那个要执行cancle取消掉。

如果在执行的过程中某个服务突然停掉了，tcc如何保证事务的一致性？

在tcc中有记录事务执行状态的表。假如订单系统突然挂掉了，再次重启的时候，会读取表中的数据，从上次的操作继续执行。

z

# Nginx：

Nginx是一个反向代理服务器。什么是反向代理？正向代理面对的是对象，反向代理面对的是服务器。

我们用来做动静分离和负载均衡。如何做动静分离？动静分离就是当服务器接收到访问静态资源的请求时，nginx不会将请求转发到web服务器，而是会自己根据请求查找到本地相应的静态资源，然后返回给客户端。如何实现动静分离？在nginx的配置文件中，通过修改location中匹配的访问路径，然后匹配到访问静态资源的请求，接着根据root中的配置，在本地查找到后返回。

**负载均衡怎么做?**

首先在location中配置访问web服务器的路径，然后配置proxy\_pass，接着配置upstream，upstream后面的值和proxy\_pass的一样。Upstream里面的server 就是负载均衡的ip。负载均衡的常用策略：轮询，iphash。

轮询存在的问题，session共享问题。Iphash存在的问题：可能会出现不同的iphash到同一个服务器上，导致服务器压力过大。

如何解决？使用轮询+redis的策略。使用redis保存session。

Location路径匹配顺序：=^~~/

=是精确匹配

^~和~是正则匹配，有^的正则匹配更优先。/是通配，如果前面三个都没有匹配上，就会进入通配/。

# http协议

http协议是一种位于应用层超文本传输协议。可以通过http协议在客户端和服务端之间通信。它是一种短连接，无状态的协议。http1.0是无连接无状态的，http1.1是短连接，无状态的。短连接是指一次请求和响应后，连接一会就关闭，这种方式可以节省资源。无状态是指服务端在收到客户端发来的消息后，只知道收到了一条消息，并不知道是谁发送的。解决这个问题可以在客户端加上cookie。

http协议分为请求协议和响应协议，请求协议有四部分组成，分别是请求行

请求头 请求空白行和请求体。请求行由请求方式，uri和http协议版本类型组成。请求方式常用的有get和post协议。Get请求方式将请求参放在请求行中，post请求方式将参数放在请求体中。如果请求的参数很多，可以使用post请求方式。

请求头中的值是以key-value的形式存放的。常见的参数：content-type：请求体的类型、content-length请求长度、cookie存放cookie。

响应协议：组成：响应行，响应头，响应体。响应行由http协议版本，状态码。1xx表示请求已经接收，2xx表示请求已经成功。3xx表示请求重定向，4xx表示客户端的错误。5xx表示服务端的错误。

响应体组成：响应首部字段，响应体。

**http和https有什么不同?。**

TTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是网景公司设计了SSL（Secure Sockets Layer）协议用于对HTTP协议传输的数据进行加密，从而就诞生了HTTPS。简单来说，HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。

　　HTTPS和HTTP的区别主要如下：

　　1、https协议需要申请证书，一般免费证书较少，因而需要一定费用。

　　2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

　　3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

　　4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

# 多线程

## CountDownLatch

是一个同步工具类，用来协调多个线程之间的同步，起到线程之间通信的作用。Countdownlatch使用计数器实现，每当有一个线程完成后，计数器就加一或者减一。当所有的线程都完成后，等待的线程就会恢复当计数器的值为0时，表示所有的线程都已经完成一些任务，然后在CountDownLatch上等待的线程就可以恢复执行接下来的任务。

CountDownLatch典型用法：1、某一线程在开始运行前等待n个线程执行完毕。将CountDownLatch的计数器初始化为new CountDownLatch(n)，每当一个任务线程执行完毕，就将计数器减1 countdownLatch.countDown()，当计数器的值变为0时，在CountDownLatch上await()的线程就会被唤醒。一个典型应用场景就是启动一个服务时，主线程需要等待多个组件加载完毕，之后再继续执行。

CountDownLatch典型用法：2、实现多个线程开始执行（任务的最大并行性。注意是并行性，不是并发，强调的是多个线程在某一时刻同时开始执行。类似于赛跑，将多个线程放到起点，等待发令枪响，然后同时开跑。做法是初始化一个共享的CountDownLatch(1)，将其计算器初始化为1，多个线程在开始执行任务前首先countdownlatch.await()，当主线程调用countDown()时，计数器变为0，多个线程同时被唤醒。

## Cyclicbarrie

CyclicBarrier和CountDownLatch的区别

1）CountDownLatch简单的说就是一个线程等待，直到他所等待的其他线程都执行完成并且调用countDown()方法发出通知后，当前线程才可以继续执行。

2）cyclicBarrier是所有线程都进行等待，直到所有线程都准备好进入await()方法之后，执行条件变量的signalAll方法唤醒等待的线程。所有线程同时开始执行！

3）CountDownLatch的计数器只能使用一次。而CyclicBarrier的计数器可以使用reset() 方法重置。所以CyclicBarrier能处理更为复杂的业务场景，比如如果计算发生错误，可以重置计数器，并让线程们重新执行一次。

## 信号灯 **Semaphore**（可以控制同时访问资源的线程数量）

## 获取灯：sp.acquire()；释放灯:sp.release();

## 多线程的4种创建方式

继承thread类，实现runnable接口，实现callable接口（有返回值），实现future接口（Future接口提供方法来检测任务是否被执行完 **isDone** ()，等待任务执行完获得结果，也可以设置任务执行的超时时间**get** (long timeout, TimeUnit unit)。

## 线程的生命周期（5个新建，就绪，运行，阻塞，死亡）

## CAS(自旋)，ABA问题

（AtomicStampedReference）解决aba问题就是在修改值时，加一个版本号或时间戳。

比较再交换，是一种乐观锁机制，没有使用synchronized。CAS有3个操作数，内存值v1，v2，旧的预期值v5，自增量v4。当且仅当预期值v5和内存值v1，v2相同时，将内存值V修改为v5+v4，否则循环比较。

## 读写锁

ReadWriteLock rwl=new ReentrantReadWriteLock(); rwl.readlock.lock(); rwl.readLock().unlock()写在finally中; rwl.writeLock().lock();rwl.writeLock().unlock();

## volatile

修饰变量的关键字，**可见性**。线程在工作的时候有一个工作内存，内存中有主内存区。当使用vlo编关键字后，线程修改的变量可以直接从工作内存中更新到主内存。然后其他线程取的时候，一定是最新的值。保证了可见性。Vol和cas一起就保证了原子性。

## 4种线程池

newsinglethreadexecutor，newfixedthreadpool，newcachedthreadpool，newschedulethreadpool

https://blog.csdn.net/lhd992692552/article/details/122498816

## 自定义线程池的7个参数

核心线程池数量(活跃)，最大核心线程池数量，超时时间，超时单位，阻塞队列，线程工厂，拒绝策略(4种，丢弃抛异常，只丢弃不抛异常，喜新厌旧，不丢弃直接等待后进入线程执行)

## 并发编程三要素：解决线程安全问题

**当多个线程要共享一个实例对象的值的时候，那么在考虑安全的多线程并发编程时就要保证下面3个要素。**

1. **原子性（Synchronized, Lock）一个或某几个操作只能在一个线程执行完之后，另一个线程才能开始执行该操作，也就是说这些操作是不可分割的，线程不能在这些操作上交替执行。**
2. **可见性(Volatile，Synchronized,Lock)一个线程对共享变量的修改,另一个线程能够立刻看到。**
3. **有序性(Volatile，Synchronized, Lock)程序执行的顺序按照代码的先后顺序执行。（处理器可能会对指令进行重排）。**

当完成了并发三要素后，自然也就解决了线程安全问题。

线程安全问题：**当多个线程访问某个方法时，不管你通过怎样的调用方式、或者说这些线程如何交替地执行，我们在主程序中不需要去做任何的同步，这个类的结果行为都是我们设想的正确行为，那么我们就可以说这个类是线程安全的。**

## sleep和wait区别

sleep是来自于Thread类，wiat是来自于Object类

线程sleep后，不会释放锁，进入阻塞队列。线程wait~~后~~释放了锁，进入等待队列。

sleep可以在任何地方使用，wait只能在同步代码里使用

## synchronized和Lock的异同

主要相同点：Lock能完成synchronized所实现的所有功能；

主要不同点：

Syschronized是一个关键字，而Lock是一个类；

Lock有比synchronized更精确的线程语义和更好的性能，而且不强制性的要求一定要获得锁；

·Syschronized同步数据少量的话，性能比Lock好，而数据大量同步，Lock性能要好

synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且最好在finally块中释放（这是释放外部资源的最好的地方）。

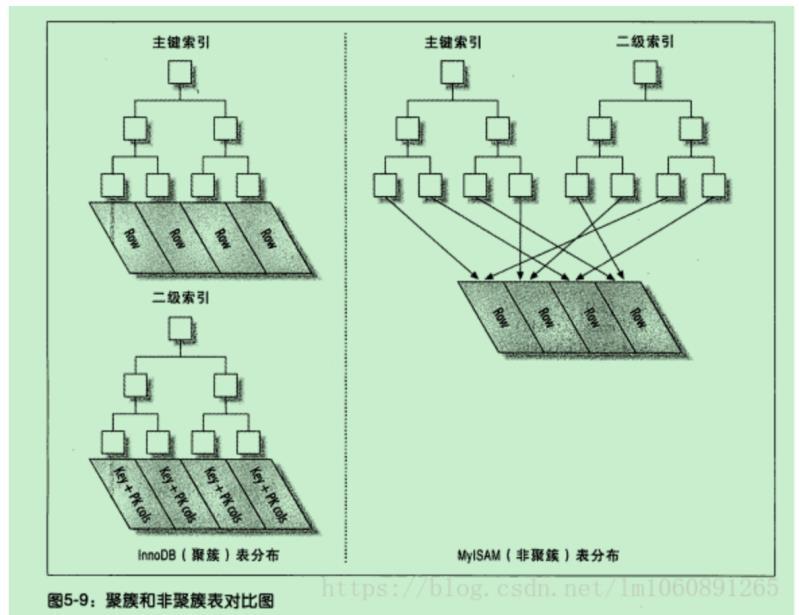
# 索引

**聚簇索引**

Innodb表中，表数据文件本身就是按照b+树组织的一个索引。叶子节点中存放的是表的每一行数据。Innodb通过主键聚集数据，如果没有主键，innodb会选择非空的唯一索引代替。如果没有这样的索引，innodb会隐式定义一个主键作为聚簇索引。

**非聚簇索引：**

非聚簇索引和聚簇索引的区别在于:非聚簇索引中的主键索引和二级索引中的叶子结点存储的都是数据的物理地址，通过索引找到物理位置后，再找到数据。



主键索引：

**二级索引**：

聚簇的在二级索引中，叶子结点不是行记录了，而是主键值。所以使用二级索引查找时，首先在二级索引中找到的是主键值，然后通过主键值找到叶子结点中的数据行。

**索引的优点：**

1数据访问更快

2聚簇索引对于主键排序查找和范围查找更快

缺点：1插入速度严重依赖于插入顺序，对于innodb，一般都会定义一个自增的主键。

2.更新主键的代价很高，每次更新后都需要对索引进行维护

如何建立索引？

create index index\_name on table\_name (column\_name）

索引不被使用的情况：

查询语句中使用 \*

索引列上有函数运算

语句中有隐式转换

表中的数据较小，或者需要查询大部分的数据。数据库自动选择最优，可能不走索引。

语句中有！=可能不走索引

如何查看语句是否走了索引？

在语句上加入关键字explain

# redis自增自减

decr 自减 incr 自增

# redis持久化RDB机制

三种触发机制：save（阻塞）、

bgsave（执行fork操作创建子进程，RDB持久化过程由子进程负责，完成后自动结束。阻塞只发生在fork阶段，一般时间很短。基本上 Redis 内部所有的RDB操作都是采用 bgsave 命令。）

自动化（redis.conf配置文件配置）。

**RDB其实就是把数据以快照的形式保存在磁盘上。**

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘。实际操作是**创建一个fork**进程，先将数据写入一个临时文件，写入成功后替换掉原来的文件。是默认的持久化方式，默认文件名为dump.rdb。

优点：恢复快，使用fork后整个redis数据库将只包含一个文件，当需要恢复时，将这单个文件可以方便的转移到其他存储介质上。

性能

高：使用rof只需要创建一个fork进程，然后由这个进程完成持久化过程，这样就可让服务进行避免io操作。

缺点：如果服务器宕机了，没有写入磁盘的数据会丢失。

# redis持久化AOF机制

**Aof以日志的形式记录服务器的每一次处理，以文本的方式记录**。

优点：可以保证更高的数据一致性。

Aof同步有三种策略，分别是每秒同步，每次操作同步和无同步。如果是每秒同步，服务器宕机后，一秒钟之内未同步的数据会丢失，如果是操作同步不会丢失数据但效率是最低的。

因为使用是是rewrite，将最新的修改记录不断追加到老的磁盘文件中，可以更好的保证数据的一致性。

缺点：恢复速度较慢，效率较低。

选择的标准，如果注重恢复速度快，性能，就使用rdb。如果注重缓存一致性，就使用aof

# Zookeeper集群

是用来对分布式系统的提供协调服务的，通过zab协议保证分布式事务的一致性。

**Zookeeper 客户端会随机的链接到 zookeeper 集群中的一个节点，如果是读请求，就直接从当前节点中读取数据；如果是写请求，那么节点就会向 Leader 提交事务，Leader 接收到事务提交，会广播该事务，只要超过半数节点写入成功，该事务就会被提交。**

：

**Zab协议原理**

Zab协议要求每个 Leader 都要经历三个阶段：**发现，同步，广播**。

* **发现**：要求zookeeper集群必须选举出一个 Leader 进程，同时 Leader 会维护一个 Follower 可用客户端列表。将来客户端可以和这些 Follower节点进行通信。
* **同步**：Leader 要负责将本身的数据与 Follower 完成同步，做到多副本存储。这样也是提现了CAP中的高可用和分区容错。Follower将队列中未处理完的请求消费完成后，写入本地事务日志中。
* **广播**：Leader 可以接受客户端新的事务请求，将新的请求广播给所有的 Follower

zookeeper集群中有三种角色，分别是leader，follower，observer。

对于客户端的查询操作，可以保证客户端的每次请求所返回的结果都是一样的，对于删除，修改等改变数据的操作，需要leader来协调的。在节点中，只有leader才有修改数据的权利，而follewer节点即使收到了修改数据的请求，也会将请求转发给leader，由leader向所有的follwer广播，让他们执行某一项操作。等超过半数的follwer执行完毕后，leader就会判定此操作已经完成，然后向所有的follwer广播该操作已经生效。

**Zookeeper选举机制：？？？**

有两种情况，一种是在初始化集群的时候，需要选出来一个leader。一种是leader宕机后，需要选出来一个leader。

**zookeeper节点的四种状态：**

Leading：表名该节点已经处于领导状态，是master，他有写权限其他没有。

Looking：表名该节点正在寻找leading

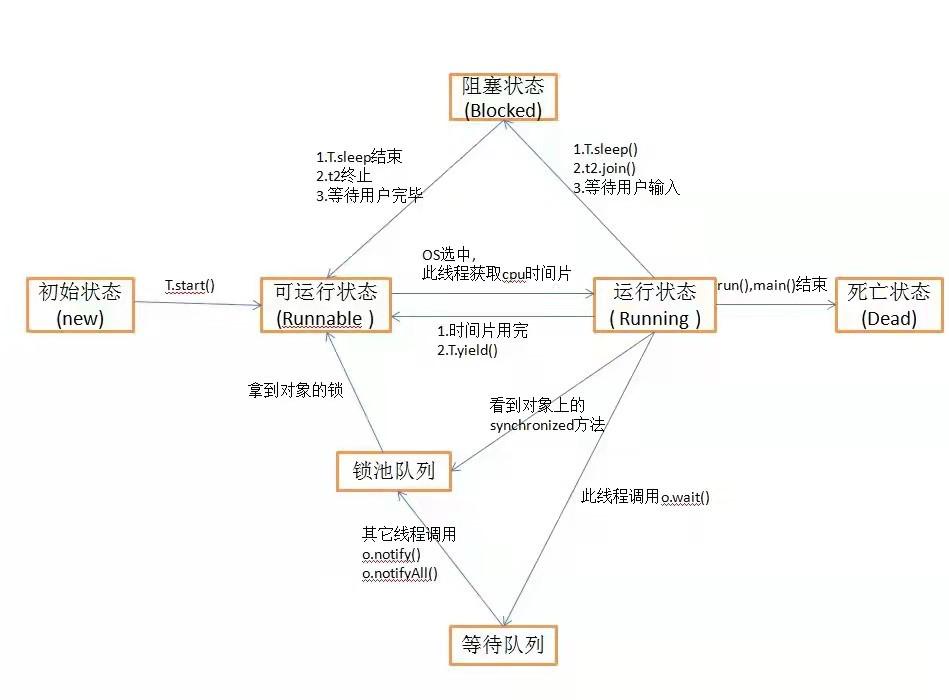
Following:表名该节正在跟随领导。可以想leader发送请求，接收leader消息，接收客户端请求。如果是写请求，会交给leader处理，然后响应给客户端。

Observing:和跟随者差不多，但是不参加选举。

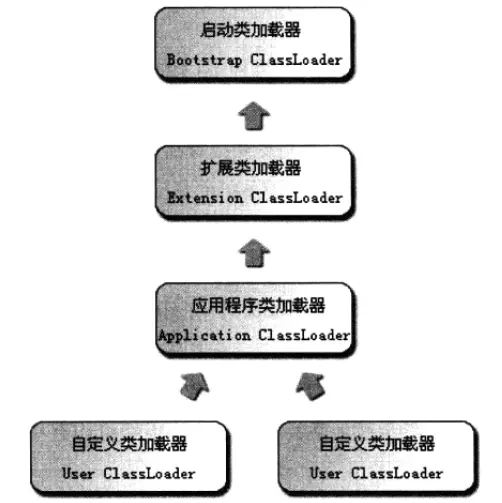
**为什么是zookeeper节点是奇数？**

提高容错率，zk容错机制在于，剩下的个数要大于宕机的个数。

# Jvm

**双亲委派机制**

了解双亲委派，需要先了解下JAVA的类加载器ClassLoader，java的类加载器主要有以下几种



 atomicinteger

什么是类加载器？

类加载器是jre的一部分，负责动态将类添加到Java虚拟机。

类加载分类

1、启动类加载器 bootstrap classloader ：加载jre/lib/rt.jar

2、扩展类加载器 extension classloader ：加载jre/lib/ext/\*.jar

3、应用程序类加载器 application classloader：加载classpath上指定的类库

**双亲委派机制**

　　双亲委派机制是指当一个类加载器收到一个类加载请求时，该类加载器首先会把请求委派给父类加载器。每个类加载器都是如此，只有在父类加载器在自己的搜索范围内找不到指定类时，子类加载器才会尝试自己去加载。

#### **双亲委派模型工作工程：**

　　1.当Application ClassLoader 收到一个类加载请求时，他首先不会自己去尝试加载这个类，而是将这个请求委派给父类加载器Extension ClassLoader去完成。

　　2.当Extension ClassLoader收到一个类加载请求时，他首先也不会自己去尝试加载这个类，而是将请求委派给父类加载器Bootstrap ClassLoader去完成。

　　3.如果Bootstrap ClassLoader加载失败(在<JAVA\_HOME>\lib中未找到所需类)，就会让Extension ClassLoader尝试加载。

　　4.如果Extension ClassLoader也加载失败，就会使用Application ClassLoader加载。

　　5.如果Application ClassLoader也加载失败，就会使用自定义加载器去尝试加载。

　　6.如果均加载失败，就会抛出ClassNotFoundException异常。

例子：

　　当一个Hello.class这样的文件要被加载时。不考虑我们自定义类加载器，首先会在AppClassLoader中检查是否加载过，如果有那就无需再加载了。如果没有，那么会拿到父加载器，然后调用父加载器的loadClass方法。父类中同理会先检查自己是否已经加载过，如果没有再往上。注意这个过程，直到到达Bootstrap classLoader之前，都是没有哪个加载器自己选择加载的。如果父加载器无法加载，会下沉到子加载器去加载，一直到最底层，如果没有任何加载器能加载，就会抛出ClassNotFoundException。

**jvm内存结构**

**虚拟机栈**，存储方法，局部变量和对象的引用。每当有一个方法被调用时，该方法就会压栈，当一个方法执行结束后，该方法就会出栈。是线程私有的。

**堆**，存储对象实例和数组。是线程之间共享的。

**方法区：**存储静态变量，常量，类信息。

**本地方法栈**：存储native方法

**程序计数器：**存储代码执行的位置

栈是私有的，堆是共享的。在堆中会产生垃圾。所以有垃圾回收机制。垃圾回收机制是分代回收策略。堆内存分为为年轻代和年老代。年轻代分为伊甸园区和幸存区，幸存区有幸存一区和幸存2区。他们之间大小比例是1:1：8。 当一个对象被new出来后，存入到伊甸园区，当**伊甸园区**满了后，进行yong gc，使用复制算法，把幸存下来的对象放进幸存1区中。当对象被回收15次后仍然存活，会将对象放进年老区中。如果年老区内存满了，会执行full gc，对整个堆进行一次垃圾回收，使用的是标记清除算法。如果经过多次垃圾回收后，年轻区和年老区都满了，jvm会报出堆内存溢出异常。

如何解决堆内存溢出异常？。两个方面，首先分析一下代码。看看有没有频繁new对象的代码，造成了频繁垃圾回收。另一方面是调整一下堆内存大小。

**Jvm调堆参数**

Xmx最大堆内存。Xms最小堆内存 Xmn年轻代大小。年老代大小=堆内存-年轻代大小。

# Spring事务：

Spring事务分为编程式事务和声明式事务。编程式式事务在开发中用到的较少。用的较多的是声明式事务。

如何使用声明式事务，首先在xml中配置事务的配置信息，tx：annocation-deriven

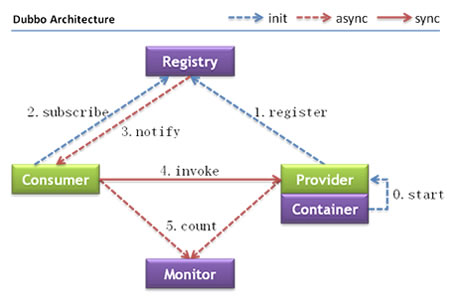
然后在需要支持事务的类上或是方法上写上@Transactional标签。标签中默认事务的传播行为是required。即如果该方法有事务，就使用该方法的事务，如果该方法没有事务，就自己创建一个事务。传播行为有7中，我们使用的的最多的就是默认的传播事务。

spring是如何管理事务的？在执行支持事务的方法时，如果方法没有抛出异常，正常执行就会执行commit，如果方法抛出了异常，就会执行回滚。我们可以在标签中，设置robackfor的值指定回滚时机。rollbackfor：用于指定触发事务回滚的异常类型，用的最多是runtimeexception，还有theowable。值是.class

Spring事务实现原理：spring事务使用aop代理实现的，在代码运行时生成一个代理对象。然后根据transactional中的配置，由拦截器transactionalinterceptor拦截。当目标对象被拦截后，会在目标方法之前创建并加入事务，接着执行目标方法的逻辑，最后根据执行情况执行回滚或提交。

#### Dubbo原理：

首先用动态代理生成一个代理实例，然后该实例执行save（name）方法，然后会进入到invoke函数中。在invoke函数中可以拿到接口名，方法名和方法的参数。通过方法名可以在zookeeper中获取到服务端的i端口。在invoke中建立一个socket连接，将接口名，方法名和方法参数封装为javabean，通过socket发送到服务端的ip端口中。服务端在接收到消息后解析，获取到接口名，方法名和方法参数。通过接口名可以获取到真正的实例对象，然后服务端用方法名，方法参数通过反射获取到需要执行的方法。方法执行完后，将结果封装到socket中写回消费者，消费者解析结果，完成。



# Springmvc执行流程descript

# Springmvc常用注解：

springmvc常用注解

1.@Controller 控制器

2.@RestController（@RestController = @Controller + @ResponseBody）将返回的数据结构转换为 JSON 格式

3.@RequestMapping  配置处理请求的路径

4. @RequestBody将前端传过来的json格式的参数映射到对象里

5.@ResponseBody  返回json格式的数据

6.@PathVariable  要用来获取 Restful风格请求的URL 参数,跟@RequestMapping（"show/{id}/{name}"）结合起来用

7.@RequestParam  前端传过来的参数名字为name，后端想用nm接收这个参数，需要在nm前加上这个注解

8.@CookieValue  获取cookie值

9.@RequestHeader  获取请求头

10.@PostMapping  只接收post请求

11.@GetMapping 只接收get请求

# 数据库事务特性：acid

数据库事务特性：

1.  原子性：原子性是指一个事务中的操作，要么全部成功，要么全部失败，如果失败，就回滚到事务开始前的状态

2.  一致性：一个事务执行之前和执行之后都必须处于一致性状态。拿转账举栗子，A账户和B账户之间相互转账，无论如何操作，A、B账户的总金额都必须是不变的。

3.  隔离性：隔离性是当多个用户 并发的 访问数据库时，如果操作同一张表，数据库则为每一个用户都开启一个事务，且事务之间互不干扰，也就是说事务之间的并发是隔离的。再举个栗子，现有两个并发的事务T1和T2，T1要么在T2开始前执行，要么在T2结束后执行，如果T1先执行，那T2就在T1结束后在执行。

4.  持久性：持久性就是指如果事务一旦被提交，数据库中数据的改变就是永久性的，即使断电或者宕机的情况下，也不会丢失提交的事务操作。

引出数据库隔离级别：

# 关于事务的隔离性等级：

1.脏读：

脏读是指一个事务在处理数据的过程中，读取到另一个未提交事务的数据。

 2.不可重复读：

不可重复读是指对于数据库中的某个数据，一个事务范围内的多次查询却返回了不同的结果，这是由于在查询过程中，数据被另外一个事务修改并提交了

3.幻读

比如说事务A查询出了五条数据。这时候事务B删除了一条id为1的数据，然后提交了。事务A查询时还是五条数据，想要删除id为1的数据时，删除失败。这就发生了幻读。

***四种隔离级别解决了上述问题***

1.读未提交（Read uncommitted）：

这种事务隔离级别下，select语句不加锁。

    2.读已提交（Read committed）：

可避免 ***脏读*** 的发生。

  3.可重复读（Repeatable read）：

MySql默认隔离级别。

4.串行化（Serializable ）：

可避免 ***脏读、不可重复读、幻读***的发生。

Serializable 是一致性最好的，性能最差的。一般不会使用 Serializable 和Read uncommitted 这两种隔离级别。

# 三范式：

第一范式：主键不能为空，字段不可再分（原子性）

第二范式：表中字段不能产生部份依赖（主键可以是复合主键，但是不建议使用，因为这样更有可能产生部份依赖）

第三范式：不能产生传递依赖。非主键字段不能传递依赖于主键，如：

主键1 字段2 字段3 字4

其中字段2依赖主键1，字段3依赖字段2，字段4依赖字段3.这就是部份依赖。应该让非主键字段直接依赖于主键字段

多对多，三张表，两个主键一张表

例：学生对教师。

学生表一张表，教师表一张表。单独再设一张关系表，表示多对多的关系：学生-教师表

学生-教师表的字段是两张表的主键，作为此表的外键。

一对多，两张表，多的表，加外键。

例：学生对班级表，一个班级有多个学生。在学生表中加入外键，外键为班级的主键。

# 乐观锁，悲观锁：

悲观锁：在select语句后加for update。这样在处理数据的过程中，数据是被锁定的，只有自己可以对数据进行操作，其他事务可以查询但不可以修改，但查询的不是最新的，这种情况是非一致性锁定读。

乐观锁是基于数据版本机制实现的。就是为数据增加一个版本号或者是时间戳。读取数据时，将版本号一同读取，在修改数据时，将版本号+1，在提交的时候先看一下版本号是否一致，如果一致就提交，如果不一致就是过期数据。

# Mybatis标签

resuptMap：处理查询的结果集合实体类属性之间的映射关系。

Collection是处理一对多的映射

association：是处理多对一的映射

sqlmap：将重复的sql语句放进sqlmap中，然后使用的时候直接引用。

动态sql：需要拼接sql语句的时候使用，增加sql语句的灵活性。

标签：if标签，判断sql中的条件是否满足，满足就拼接在where后面，不满足就扔掉。

Where：代替where1=1的作用

foreach标签，代替sql中的in。

foreach标签的属性含义如下： 标签用于遍历集合，它的属性：

•collection：代表要遍历的集合元素，注意编写时不要写#{}

•open：代表语句的开始部分

•close：代表结束部分

•item：代表遍历集合的每个元素，生成的变量名

•sperator：代表分隔符



# redis缓存：

缓存击穿

    缓存击穿是指在缓存中没有数据（数据过期了），但是在数据库中有数据。这时由于并发访问的用户数较多，在缓存中查询不到数据后同时去数据库中查询，造成数据库压力较大。

解决：1.设置热点数据永不过期

  2.加互斥锁。击穿的原因之一是并发查询，此时加个互斥锁，让用户排 队查询。

缓存穿透

缓存穿透是指缓存和数据库中都没有的数据。用户此时发起请求，但是因为缓存和数据库中都没有数据，导致用户一直不断请求这个数据，使数据库压力过大。

解决：用户查询没有结果的数据后返回一个代表空值的值。

缓存雪崩

缓存雪崩是指缓存数据大批量同时到期，而查询的数据量过大，导致数据都从数据库中查询，使数据库压力过大。和缓存击穿不同的是，雪崩是大批量不同的数据同时过期了，击穿是并发查询同一条数据。

解决：1.将缓存过期时间设置随机，避免同时过期的情况。

          2.如果缓存数据库是分布式部署，将热点数据均匀分布在不同的缓存数据库中。

          3.设置热点数据永远不过期。

# 商城项目

执行流程，

用户在进入网站后可以浏览商品的信息，同时可以将商品放进购物车中，但是不能支付。

当用户登录后可以将商品放进购物车中，同时可以下单支付等操作。

如何实现的？

注意点：

1.未登录时可以浏览，添加购物车等等

2.登录后可以添加购物车，下单，支付。

实现原理：

1. 未登录时**购物车的实现**：未登录时，购物车使用cooki存放商品，cookie的key存游客的名字，游客的名字随意起。cookie的value存放商品，value是一个map，map的key是商品id，value是商品数量。
2. **登录后的购物车**使用redis存放。redis使用hash结构，hash有三个字段，分别是key，field和value。key是用户id，field和value是cookie中的value，使用redisTemplate中的putAll方法可以直接将map放进去。在把商品放进redis之前需要有两个判断，第一个判断 ：该用户购物车在没登录时，是否有商品。如果有商品，就把cookie中的商品存进redis中，此时有另一个判断，该用户在登录后是否有购物车。如果没有就创建一个购物车存放。如果有就把新放入的商品增加到购物车。
3. **下订单：**用户在点击支付后，首先会生成一个订单，订单中有订单号和其他必要的参数。当订单生成以后，商品就从库存中减掉了。当用户点击提交订单后，会进入支付流程。如果订单在一定时间没有支付，就会被取消。我们在后端写了一个定时任务，每过10秒就扫描信一次订单，把当前时间减去订单生成的时间，如果大于十分钟就把订单取消，同时把减掉的库存再加回去。
4. **支付**：支付我们单独写了个网关项目，因为其他的项目也会用到支付的代码，为了实现代码可以复用，将支付功能写成了网关。支付网中有两种支付方式，支付宝支付和吧支付。

**支付宝支付流程**：网关在收到支付请求时，会收到商家传过来的参数，网关把参数封装在treemap中，因为treemap是会根据字典排序的。把订单号，appid，商品总金额等参数封装完成后，将treemap转换为String形式，使用商家的私钥进行签名，这时将签名的结果再次封装到treemap中作为一个参数。此时把签名的参数通过调用支付宝接口，发起支付请求。当支付宝收到请求后首先进行验签，验签通过后，支付宝给网关返回一个付款页面，网关把页面返回给商家后，商家传到前台。用户付款成功后支付宝进行异步回调，分别回调商家的前台和网关后台。商家前台展示一个支付成功的页面以及其他的后续操作。回调网关后，网关进行验签，确认是支付宝回调后，修改网关的订单状态，然后网关再回调商家。商家收到回调后自改商家的订单状态，然后发货。这样就支付完成了。

**微信支付流程**：封装参数，签名，调接口返回mweburl，请求mweburl拉起支付页面，异步回调，验签，改订单，发货。。。

**网关**：网关中有订单表，订单表的功能是记录**那个系统访问了网关**还有**对账**功能。因为有很多系统都调用网关进行支付，所以网关也要知道是谁来调用了。订单表中有订单id，时间，订单状态，金额等等。

**对账**：网关会定时指定对账任务，时间自定义，一般是网站访问量小的时候对账，比如晚上1.0等。网关先调起支付宝的对账接口，获取所有订单后和自己记录的订单比较，如果不同，以支付宝的订单为准，将没记录的订单添加到网关的订单表中。然后商家也要和网关对账，此时网关的订单已经是准确的了，所以以网关的订单为准。

遇到的问题：减库存问题，网关回调失败问题，表单重复提交问题，接口幂等性问题

**减库存**，当库存中有一件商品，但是有两个用户都购买时，结果应该是有一个用户失败的。（如何实现？）利用数据库的行级锁，我们在减库存时，不能先查出来库存够不够，因为查出来数据后有可能已经过期了。所以直接减，不查。在update语句中直接减，同时在update的where里面加个判断条件，判断库存够不够减的。

**回调失败**：补偿机制。在网关中增加一个回调异常表，每当有网关回调商家失败的情况，就失记录下来。字段：时间，回调url，订单号，回调状态123。设置定时回调任务3个，分别对应123.三次回调失败后人工解决。

**表单重复提交**：使用token，后台生成token，发前台同时后台存进redis，redis中key是生成的token值，value随意。提交时根据生成的随机字符串（token）删掉。再次提交再删，但是之前删过，所以二次提交后删不掉了 ，返回值0，提交失败。

**接口幂等性**：可能有网关多次回调商家的情况，回调到商家接口后先判断订单状态，如果第一次回调就修改订单状态，如果已修改说明之前已经回调过了。

# 权限管理项目

这是一个基于角色权限的访问控制系统，不同角色的员工在登录系统后，有不同的权限。系统根据员工所拥有的的权限来显示不同的菜单和动作。（动作是什么？）动作就是url连接。

动作分为普通动作和授权动作。普通动作是加载页面所必须的url，还有就是所有员工公共的url。授权动作是管理员需要授权才可以有的动作，比如对员工的修改，增加等。只有管理员授权了这些动作，员工才有修改，增加员工的权利。

流程

员工在登陆以后，系统通过员工的id查询到员工的角色，然后根据角色查询到员工的菜单和动作列表，之后把这些菜单和动作封装到list中返回给前端。前端解析出员工的菜单和动作显示出出来。

实现

表关系

角色-用户-一对多

菜单-动作-一对多

动作-角色-多对多

角色-菜单-多对多

遇到的问题：

一个用户在地址栏中访问一个不属于他权限下的菜单时，能够访问得到。这样就造成了不安全。如何解决？设置一个拦截器，在拦截器中检查这个请求在不在用户的访问权限中。

# 单点登录：

1. 用户访问a系统，首先进入到a的拦截器，a的拦截器判断是否有cookie，是否有token，发现都没有。

2.系统带着自己的url跳转到SSO登录系统， SSO系统也没有登录，此时先保存传过来的url，然后跳转到用户登录页。

3.用户填写用户名、密码，SSO系统进行认证后，将登录状态写入SSO的session，浏览器（Browser）中写入SSO域下的Cookie。

4.SSO系统登录完成后会生成一个token存到redis中，key是token

的值，value随意。然后跳转到a系统，同时将token作为参数传递给a系统。

5.app系统拿到token后，从后台向发送请求，验证token是否有效。验证功能写为接口形式。

6.验证通过后，a系统将登录状态写入session并设置a网站域下的Cookie。

至此，跨域单点登录就完成了。以后我们再访问app系统时，app就是登录的。接下来，我们再看看访问app2系统时的流程。

1.用户访问b系统，首先进入到b的拦截器，b的拦截器判断是否有cookie，是否有token，发现都没有，带着自己的url跳到sso。

2.sso发现自己域下有cookie，说明之前有网站登录了，不需要重新登录认证。

3.SSO生成token存进redis，浏览器跳转到b系统，并将token作为参数传递给b。

4.b拿到token，后台访问SSO，验证token是否有效。

5.验证成功后，b将登录状态写入session，并在b网站域下写入Cookie。

# sql执行顺序

(1)from

(2) on

(3) join

(4) where （对从数据库表文件加载到内存中的原生数据过滤）

(5)group by(开始使用select中的别名，后面的语句中都可以使用)

(6) avg,sum....

(7)having（对分组之后的数据进行过滤）

(8) select

(9) distinct去重

(10) order by

# 左右连接，内连接

**左连接**的意思是，无论是否符合ON语句后面的表连接条件都会把左边那张表的记录全部查询出来，右边的那张表只匹配符合条件的数据行。

**右连接**则与之相反

**内连接**指的是把表连接时表与表之间匹配的数据行查询出来，就是两张表之间数据行匹配时，要同时满足ON语句后面的条件才行。

# 进程和线程的区别

”进程是资源分配的最小单位，线程是CPU调度的最小单位“

做个简单的比喻：进程=火车，线程=车厢

* 线程在进程下行进（单纯的车厢无法运行）
* 一个进程可以包含多个线程（一辆火车可以有多个车厢）
* 不同进程间数据很难共享（一辆火车上的乘客很难换到另外一辆火车，比如站点换乘）
* 同一进程下不同线程间数据很易共享（A车厢换到B车厢很容易）
* 进程要比线程消耗更多的计算机资源（采用多列火车相比多个车厢更耗资源）
* 进程间不会相互影响，一个线程挂掉将导致整个进程挂掉（一列火车不会影响到另外一列火车，但是如果一列火车上中间的一节车厢着火了，将影响到所有车厢）
* 进程可以拓展到多机，线程最多适合多核（不同火车可以开在多个轨道上，同一火车的车厢不能在行进的不同的轨道上）
* 进程使用的内存地址可以上锁，即一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。（比如火车上的洗手间）－"互斥锁"
* 进程使用的内存地址可以限定使用量（比如火车上的餐厅，最多只允许多少人进入，如果满了需要在门口等，等有人出来了才能进去）－“信号量”

# servlet生命周期

在servlet生命周期中主要有三个方法：

init方法，初始化阶段调用

service方法处理请求阶段调用

destory方法终止阶段调用

初始化阶段：

Servlet容器加载Servlet，加载完成后，Servlet容器会创建一个Servlet实例并调用init()方法，init()方法**只会调用一次。**

创建servlet实例的三种时机：

1. **Servlet容器启动时，自动装载某些servlet，实现这个需要在web.xml文件中添加<loadstartup>1</load-on-startup>**
2. 当客户端**首次**向http服务器发送请求时

处理请求阶段：

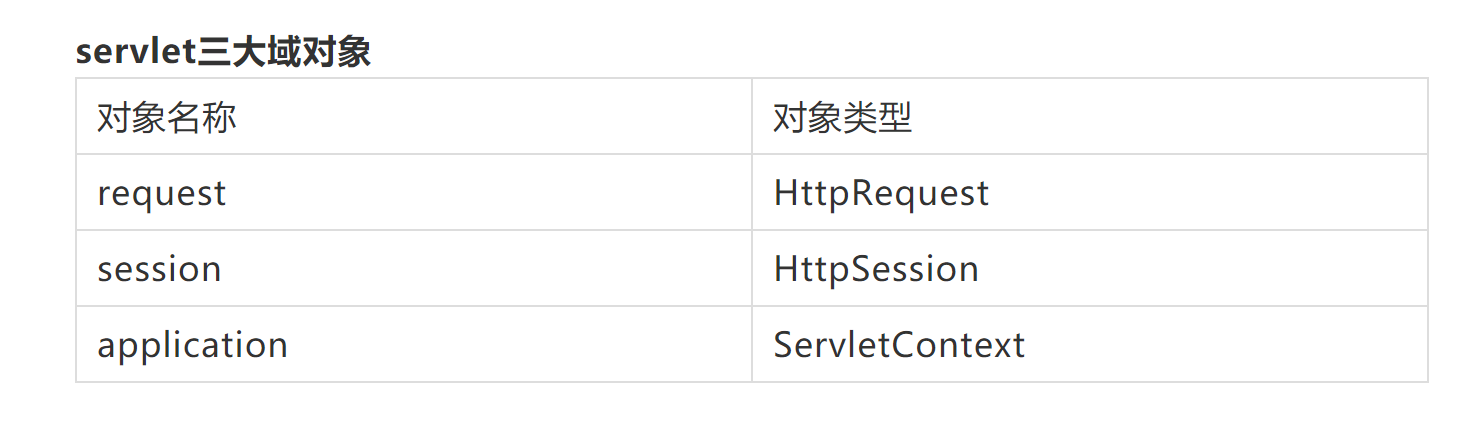
当一个请求发送到http服务器后，服务器就会产生一个新的线程去处理，http服务器把请求对象封装为ServletRequest对象。然后去servlet容器中找对应的servlet，调用service方法处理请求，调用完成后，返回SerrvletResponse对象，http服务器再将其封装为resopnse对象返回给客户端。

p终止阶段：

web服务器被终止的时候

重新加载的时候

# servlet三大域对象



## **request**

##### **生命周期：**

**创建**：客户端向服务器发送一次请求,服务器就会创建request对象.

**销毁**：服务器对这次请求作出响应后就会销毁request对象.

**有效**：仅在当前请求中有效。

## **.session**

##### **生命周期：**

**创建**：服务器端第一次调用getSession();(保存在服务器内存中)

**销毁**：

1.非正常关闭服务器(正常关闭session会序列化，再次启动服务器session会被反序列化)；

2.session过期了，默认30分钟.

3.手动调用session.invalidate();

**注意**:关闭浏览器再次访问会找不到session的会话id而不是session被销毁了。session过期了30分钟后自动销毁

**有效**：用户打开浏览器会话开始，直到关闭浏览器会话才会结束。一次会话期间只会创建一个session对象

## **application**

##### **生命周期：**

**创建**：服务器启动的时候,服务器为每个WEB应用创建一个属于该web项目的对象ServletContext类.

**销毁**：服务器关闭或者项目从服务器中移除的时候.

**有效**：此信息在整个服务器上被保留。

## **区别**

**request:** 每一次请求都是一个新的request对象,如果在web组件之间需要共享同一个请求中的数据,只能使用请求转发.

**session:** 每一次会话都是一个新的session对象,如果需要在一次会话中的多个请求之间需要共享数据,只能使用session.

**application:** 应用对象,Tomcat启动到关闭,表示一个应用,在一个应用中有且只有一个application对象,作用于整个Web应用,可以实现多次会话之间的数据共享.

# JSP中九大内置对象

request    请求对象　  类型 javax.servlet.ServletRequest   作用域 Request

response   响应对象   类型 javax.servlet.SrvletResponse  作用域  Page

pageContext 页面上下文对象 类型 javax.servlet.jsp.PageContext  作用域  Page

session   会话对象    类型 javax.servlet.http.HttpSession  作用域   Session

application  应用程序对象  类型 javax.servlet.ServletContext   作用域

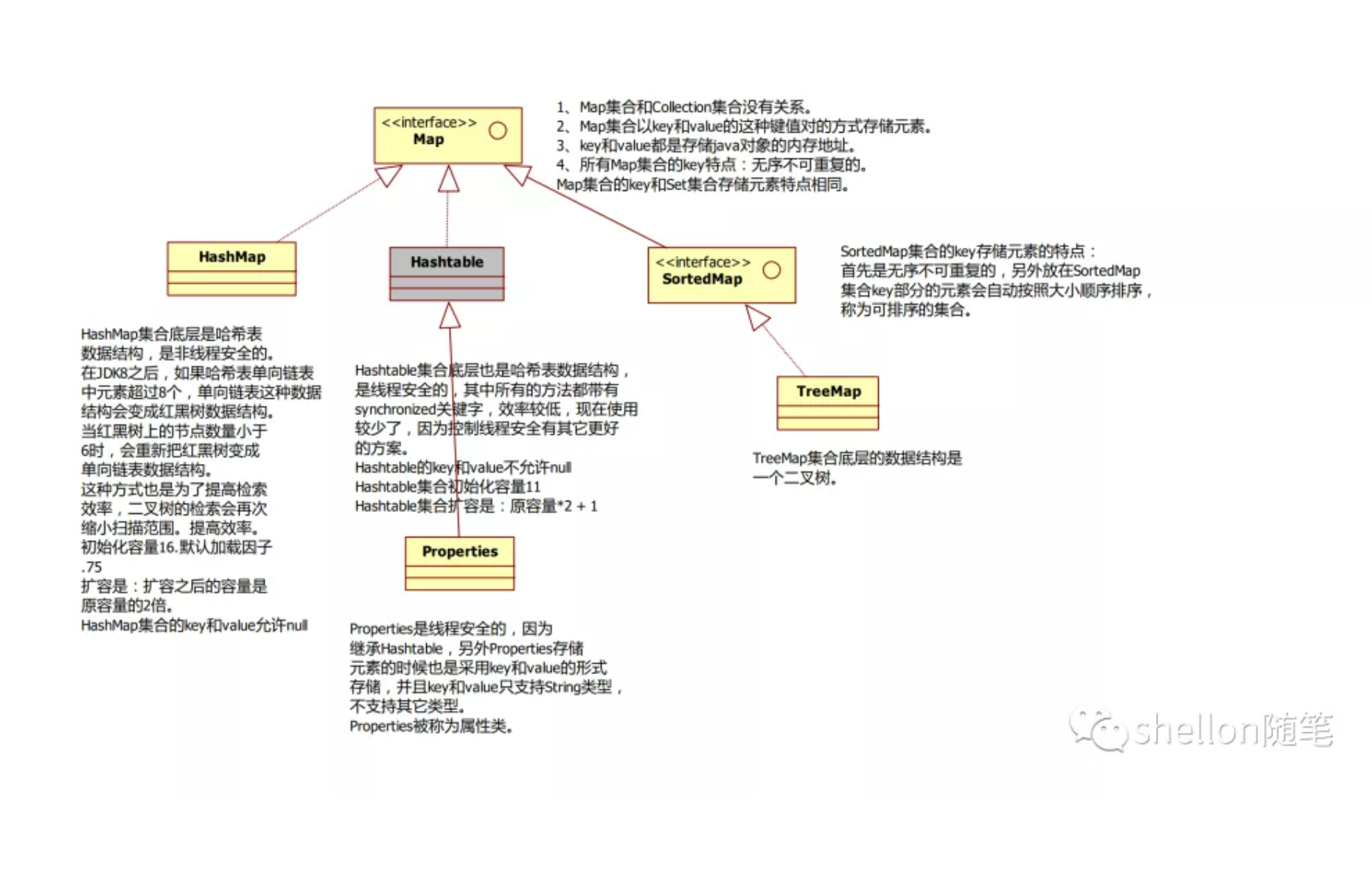
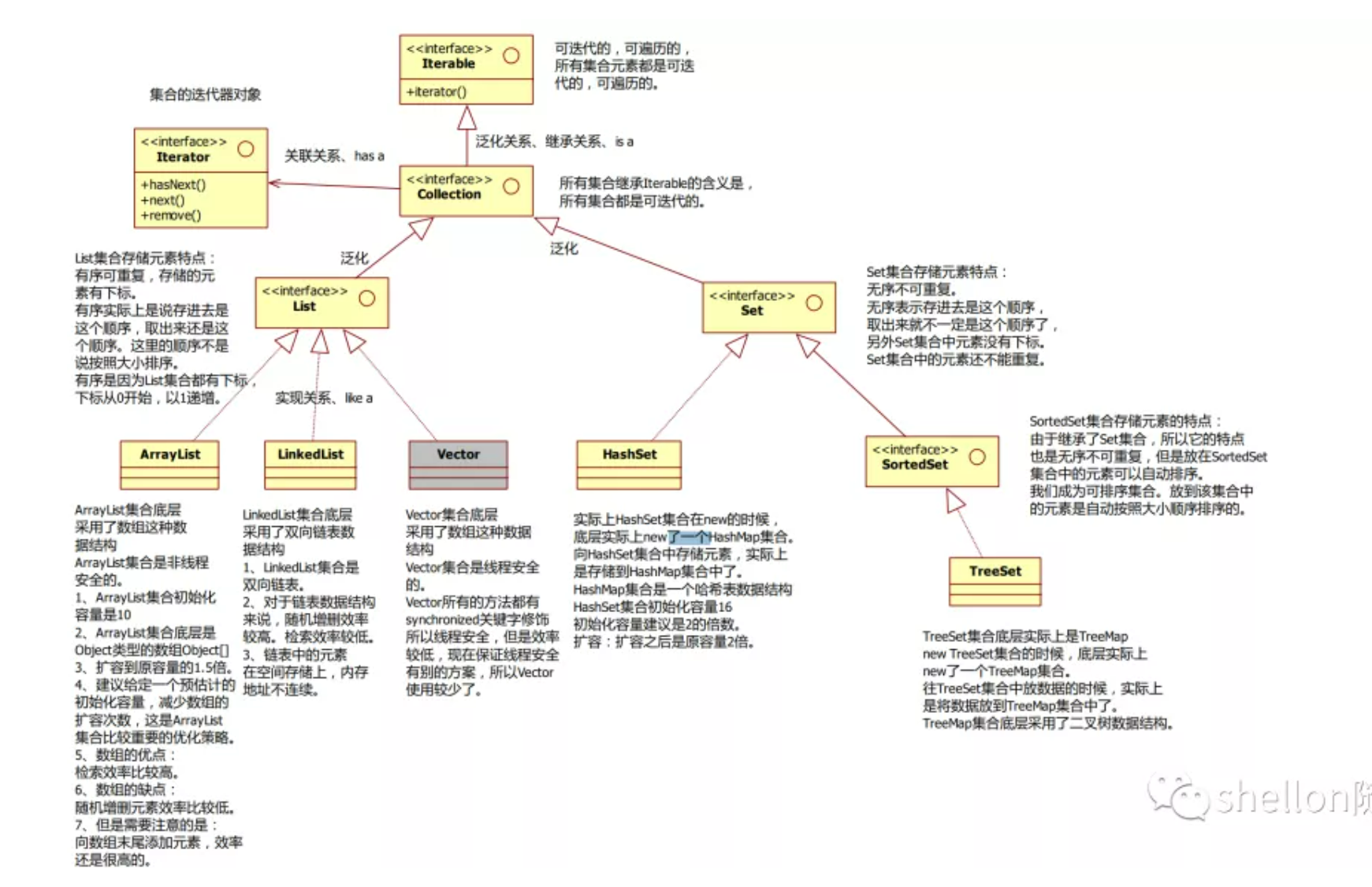
out      输出对象    类型 javax.servlet.jsp.JspWriter   作用域   Page

config    配置对象    类型 javax.servlet.ServletConfig   作用域   Page

page     页面对象    类型 javax.lang.Object       作用域   Page

exception 错误处理对象 类型 java.lang.Throwable 作用域   Page

# 集合



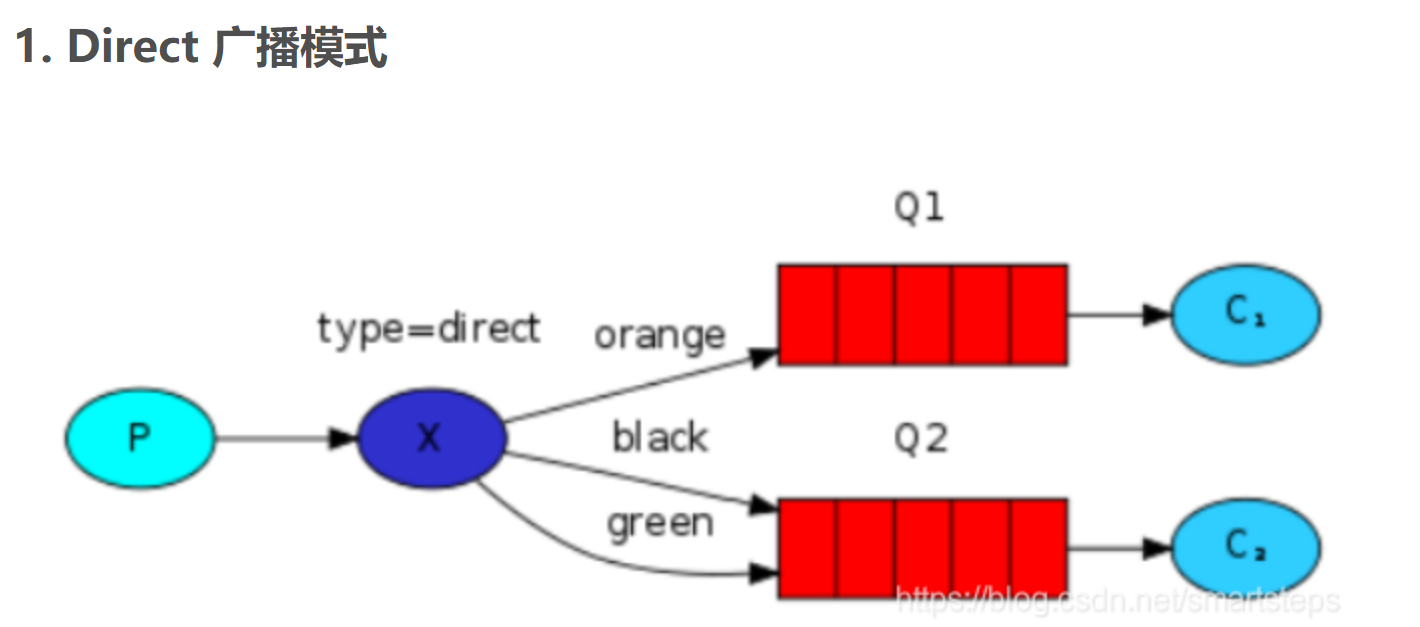
# redis事务

redis对事务的支持目前还比较简单。Redis只能保证一个client发起的事务中的命令可以连续的执行，而中间不会插入其他client的命令。当一个client在一个链接中发出多命令时，这个连接会进入一个事务中，该连接后续的命令不会立即执行，而是先放在一个队列当中，当执行exec命令的时候，redis会顺序的执行**队列**中的所有命令。

# rabbitmq工作模式

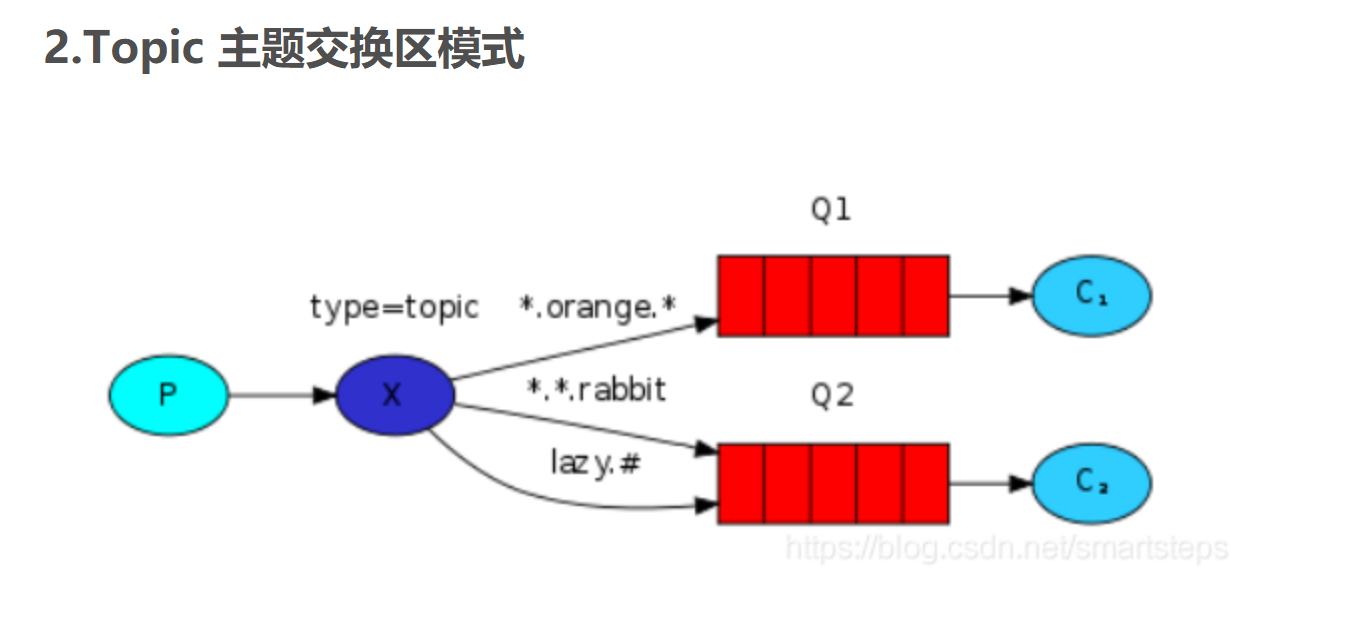
一共五种工作模式，前三种有交换机，后两种没有交换机。所谓没有交换机是指没有指定交换机，其实使用的是默认的交换机

**direct直连模式**

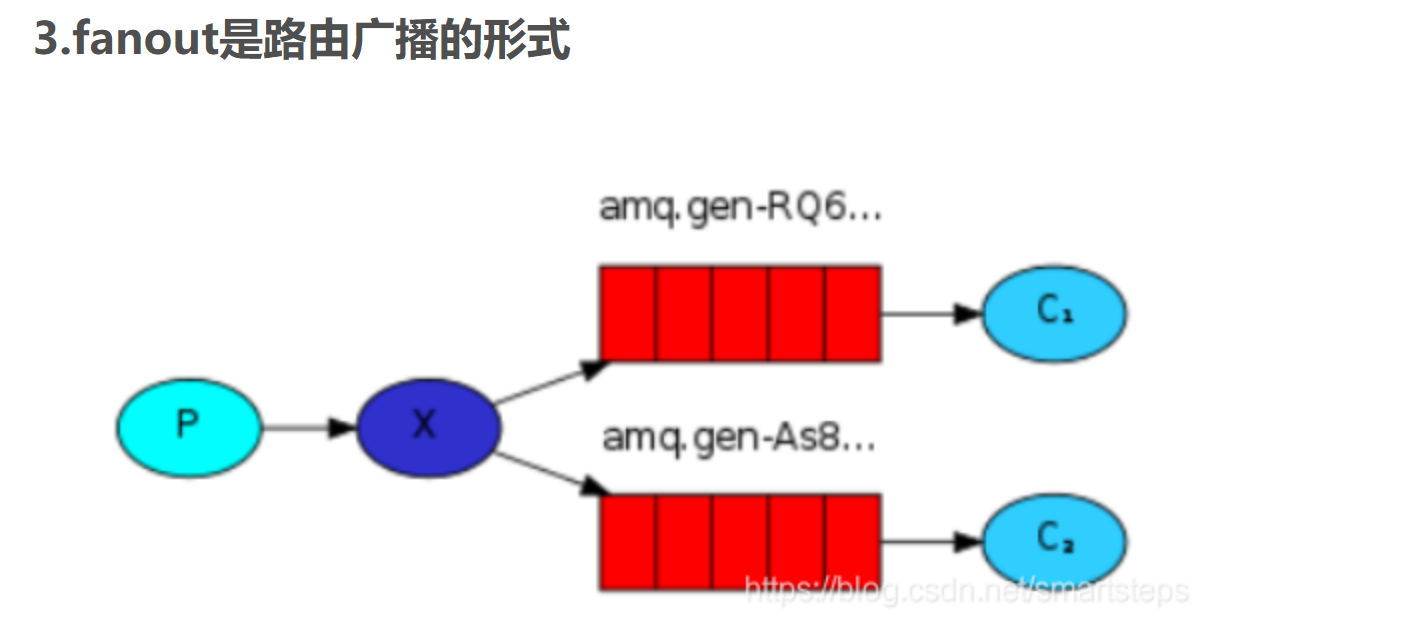


创建消息队列的时候,指定一个路由键（RoutingKey）.当发送者发送消息的时候,指定对应的Key.当Key和消息队列的RoutingKey一致的时候,消息将会被发送到该消息队列中.

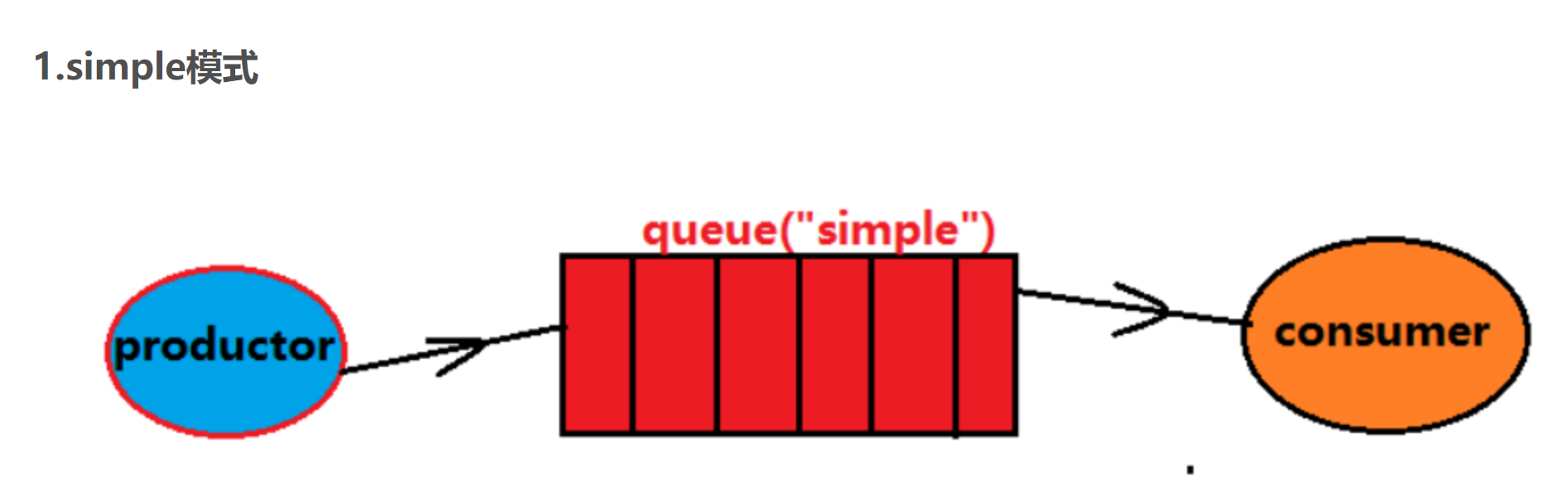
如果两个消息队列绑定同一个routingkey时，如何消费？使用轮询



topic转发信息主要是依据通配符,队列和交换机的绑定主要是依据一种模式(通配符+字符串),而当发送消息的时候,只有指定的Key和该模式相匹配的时候,消息才会被发送到该消息队列中.通配符：\* 表示一个词，# 表示零个或多个词

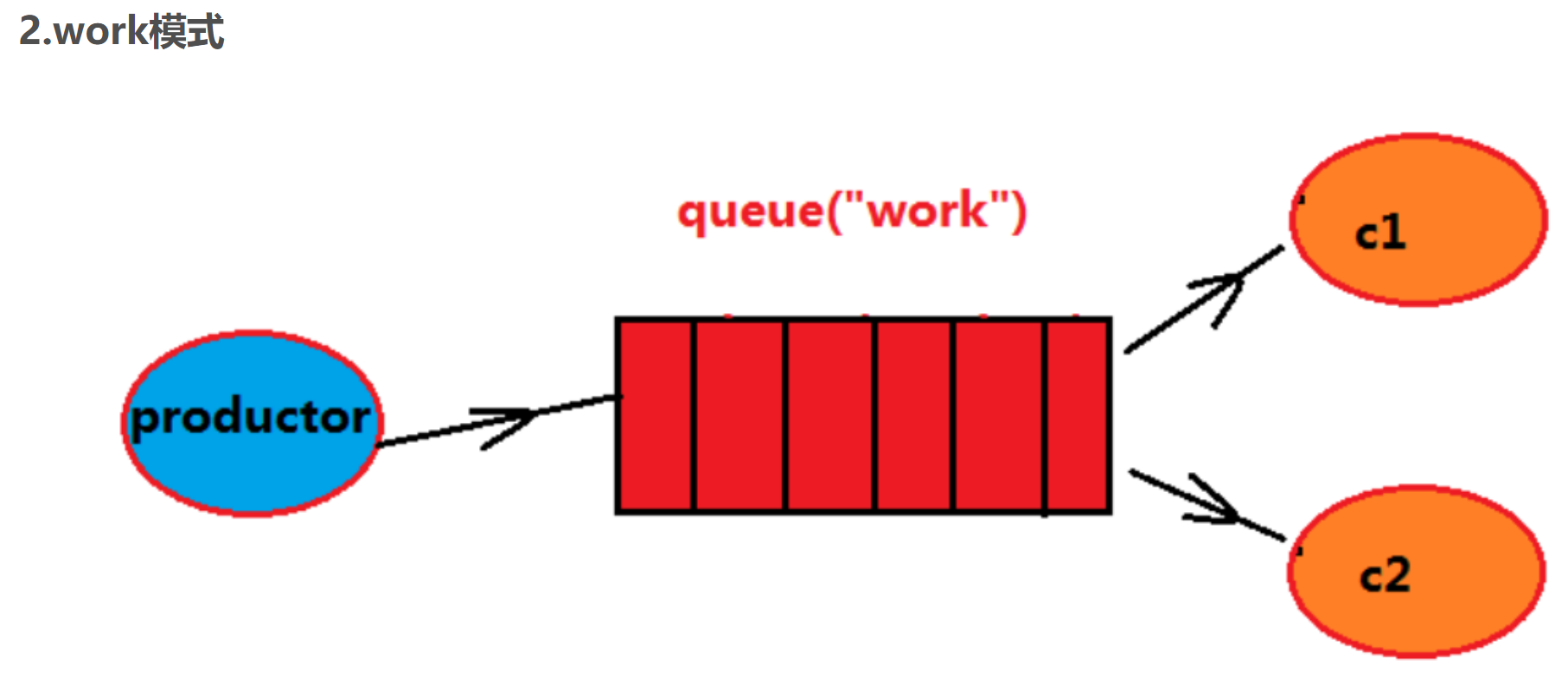


fanout是路由广播的形式,将会把消息发给绑定它的全部队列。



消息生产者p将消息放入队列

消费者监听队列,如果队列中有消息,就消费掉,消息被拿走后,自动从队列删除



生产者将消息放入队列

多个消费者同时监听同一个队列 , 消息如何被消费 ?

C1,C2 共同争抢当前消息队列的内容 , 谁先拿到消息 , 谁来负责消费

rabbitmq回调

# linux常用命令：

init 6 重启

init 0关机

cd /        切换到根目录

cd ..  切换到上一级目录

ls                查看当前目录下的所有目录和文件

ll 详细查看

mkdir  创建目录

touch 创建文件

rm -f  强制删除文件

rm -r 递归删除

mv 移动文件

cp 赋值

vi 编辑

wq 保存退出

wq！ 强制保存并退出

tar -zcvf 打包压缩后的文件名 要打包的文件（不是压缩）

wget 下载

tar [-zxvf] 压缩文件   解压

ps -ef | grep 8080 查看8080端口占用情况

ps -ef | grep tomcat 查看tomcat是否启动

su - 用户名 切换用户

pwd     查看当前目录路径

 kill -9 pid(强制杀死进程)

**ifconfig：查看网卡信息**

**ping**

# 封装继承多态的关系？

封装：

把一个对象的属性私有化，同时提供对外界访问属性的方法形成类。

继承：

在原有类的基础上，进行更具体更详细的定义。可以从原有类上拿到所有东西（），可以改造原有的类，可以在原有类基础上添加新东西。

多态：

多态是面向对象编程的一个特性，多态就是多种状态。什么是多种状态？不同的子类在继承父类的时候重写了父类的同一个方法，表现出一个方法有多种状态。同时，以多种方法实现接口也是多态。所以能看出，多态是在继承的基础上来的。

# ssm和springboot区别：

springboot

使用springboot我们可以不用或者很少使用配置就可以使项目运行起来。因为Spring框架需要进行大量的配置，Spring Boot而springboot使用习惯优于配置的理念，通过自动配置，让项目设置变得很容易。开发者只需要引入自己需要的第三方库即可。

Sping Boot的出现是为了使开发者更注重业务的逻辑，简化了对开发环境的配置。

优点：

**帮助我们快速启动一个web项目。**

**继承了原有的spring功能。**

**简化了spring使用流程。**

ssm

spring是一个轻量级的控制反转（IoC）和面向切面（AOP）的容器框架。

SpringMVC分离了控制器、模型对象、分派器以及处理程序对象的角色，这种分离让它们更容易进行定制。

MyBatis是一个支持普通SQL查询，存储过程和高级映射的持久层框架。

ssm优点：

spring ，核心是ioc、aop技术。ioc解耦，使得代码复用，可维护性大幅度提升，aop提供切面编程，同样的增强了生产力。

mybatis的sql可以由开发者去掌控和调优，对sql的控制更加直观。但是在业务场景比较复杂，sql好多联合关联的情况下，mybatis的sql语句编写就比较麻烦了。所以缺点就是对sql不熟悉的开发者就不太友好了。

# maven

maven是一个项目管理工具，他可以管理项目之间的jar包依赖关系。通过配置pom.xml文件可以轻松导入项目需要的jar包。

同时maven还是一个仓库，maven仓库有本地仓库，maven私服仓库和中央仓库。

本地仓库就是当我们导入项目需要的jar包时，maven会自动在本地创建一个仓库，存储我们用过的jar包，下次再使用时可以直接导入，不用下载。

maven私服，公司的仓库，其他人在写好jar包后可以上传到maven私服中，需要的时候任何人可以在私服下载。

maven中央仓库，在下载jar包时，maven首先查找本地仓库有没有，如果没有就去私服找，如果私服没有就去中央仓库找。

# mysql分区：

数据库分区是一种物理数据库设计技术，分区的意思是指将同一表中不同行的记录分配到不同的物理文件中，几个分区就有几个.idb文件。分区的目的是为了在特定的SQL操作中减少数据读写的总量以缩减sql语句的响应时间，同时对于应用来说分区完全是透明的。

MYSQL的分区主要有两种形式：水平分区和垂直分区。

水平分区是根据表的行进行分区，也就是把一张大表分成多个小表。原表有十万行数据，分表分两万行一个小表。

水平分区一定要通过某个属性列来分割。比如一张包括每年的历史大事的数据表，数据都存在一张表中，可以按年份字段进行分区，从而让每个分区的数据量适中，避免全表过大。

垂直分区：

对表进行垂直划分来减少原表的长度。比如说原来一张表中与15个列，通过垂直分区，把15个列分为5个列一张表。

实现分区的四种算法：

**range**：根据范围分区

partition by range(salary)

(partition p1 values less than (**1000**),

partition p2 values less than (**2000**));

**list**：罗列出值，根据值进行分区：

partition by list(

    partition name1 values in(1,3,5),

    partition name2 values in(2,4,6),..

)

**hash：**指定要分几个区，指定根据那个列值hash，至于如何hash是mysql自动完成的

partition by hash(**salary**)

partitions **4**;

**key：**跟hash差不多，key也是使用hash，使用是mysql自己的hash函数。hash可以自己指定一种hash函数**。**

partition by key(birthdate)

partitions **4**;

# Tomcat优化：

1. 内存优化

-Xms：Java虚拟机初始化时堆的最小内存，一般与 Xmx配置为相同值，这样的好处是GC不必再为扩展内存空间而消耗性能；

        -Xmx：Java虚拟机可使用堆的最大内存；

        -XX:PermSize：Java虚拟机永久代大小；

        -XX:MaxPermSize：Java虚拟机永久代大小最大值；

1. 线程优化：在Connector中设置最大线程数。tomcat默认线程数是200
2. tomcat做负载均衡

# 什么是微服务?

就是根据业务拆分成一个个的服务，彻底解耦。每一个服务提供单独的业务功能，一个服务做一件事。

# 谈谈VUE

**1.vue父组件向子组件传递数据？**

答：通过props

**2.子组件向父组件传递事件？**

答：$emit方法

**3.<keep-alive></keep-alive>的作用是什么?**

答:keep-alive 是 Vue 内置的一个组件，可以使被包含的组件保留状态，或避免重新渲染。

**4.说出几种vue当中的指令和它的用法？**

答：v-model双向数据绑定；

v-for循环；

v-if v-show 显示与隐藏；

v-on事件；v-once: 只绑定一次。

v-bind绑定一个value属性；

**5.axios及安装?**

答：请求后台资源的模块。npm install axios --save装好，

js中使用import进来，然后.get或.post。返回在.then函数中如果成功，失败则是在.catch函数中。

**其他：**

4. final 在 java 中有什么作用？

final 修饰的类叫最终类，该类不能被继承。

final 修饰的方法不能被重写。

final 修饰的变量叫常量，常量必须初始化，初始化之后值就不能被修改。

如何决定使用 HashMap 还是 TreeMap？

对于在Map中插入、删除和定位元素这类操作，HashMap是最好的选择。然而，假如你需要对一个有序的key集合进行遍历，TreeMap是更好的选择。基于你的collection的大小，也许向HashMap中添加元素会更快，将map换为TreeMap进行有序key的遍历。

1、 HashMap和Hashtable是Map接口下的两个实现类，因为Map对象是键值对的，所以此两类也是键值对的。

2、 HashMap是线程非安全的，Hashtable是线程安全的，所以HashMap的效率高于Hashtable。

3、 HashMap允许键或值为null，键最多只可以有一个为null，值不受限制。而Hashtable键或值都不许为null。

注意：Hashtable中的“t”是小写的。

ArrayList和Vector使用数组存储元素；LinkedList使用链表存储元素

2. ArrayList和Vector插入删除数据时，需要搬运数据，效率较差；LinkedList使用链表，不需要搬运数据，效率高

3. ArrayList和Vectory查询时，按数组下标查询，不需要遍历，效率高；LinkedList需要遍历，查询效率底