

### 1. redis各种应用场景

- a. 更多的数据结构；
- b. 可持久化；
- c. 计数器；
- d. 发布-订阅功能；
- e. 事务功能；
- f. 过期回调功能；
- g. 队列功能；
- h. 排序、聚合查询功能。

### 2. redis持久化机制

- a. RDB：快照形式是直接把内存中的数据保存到一个 dump 文件中，定时保存，保存策略。（会丢数据）
- b. AOF：把所有的对Redis的服务器进行修改的命令都存到一个文件里，命令的集合。（影响性能）

### 3. mysql调优

- a. explain select语句；
- b. 当只要一条数据时使用limit 1；
- c. 为搜索字段建索引；
- d. 避免select \*；
- e. 字段尽量使用not null；
- f. 垂直分割；
- g. 拆分大的delete和insert语句：delete和insert会锁表；
- h. 分表分库分区。

### 4. 有没了解Docker，Docker和虚拟机有什么区别？

- 1、虚拟机：我们传统的虚拟机需要模拟整台机器包括硬件，每台虚拟机都需要有自己的操作系统，虚拟机一旦被开启，预分配给他的资源将全部被占用。，每一个虚拟机包括应用，必要的二进制和库，以及一个完整的用户操作系统。
- 2、Docker：容器技术是和我们的宿主机共享硬件资源及操作系统可以实现资源的动态分配。  
容器包含应用和其所有的依赖包，但是与其他容器共享内核。容器在宿主机操作系统中，在用户空间以分离的进程运行。
- 3、对比：
  - 1. docker启动快速属于秒级别。虚拟机通常需要几分钟去启动。
  - 2. docker需要的资源更少，docker在操作系统级别进行虚拟化，docker容器和内核交互，几乎没有性能损耗，性能优于通过Hypervisor层与内核层的虚拟化。；
  - 3. docker更轻量，docker的架构可以共用一个内核与共享应用程序库，所占内存极小。同样的硬件环境，Docker运行的镜像数远多于虚拟机数量。对系统的利用率非常高
  - 4. 与虚拟机相比，docker隔离性更弱，docker属于进程之间的隔离，虚拟机可实现系统级别隔离；
  - 5. 安全性：docker的安全性也更弱。Docker的租户root和宿主机root等同，一旦容器内的用户从普通用户权限提升为root权限，它就直接具备了宿主机的root权限，进而可进行无限制的操作。虚拟机租户root权限和宿主机的root虚拟机权限是分离的，并且虚拟机利用如Intel的VT-d和VT-x的ring-1硬件隔离技术，这种隔离技术可以防止虚拟机突破和彼此交互，而容器至今还没有任何形式的硬件隔离，这使得容器容易受到攻击。
  - 6. 可管理性：docker的集中化管理工具还不算成熟。各种虚拟化技术都有成熟的管理工具，例如VMware vCenter提供完备的虚拟机管理能力。
  - 7. 高可用和可恢复性：docker对业务的高可用支持是通过快速重新部署实现的。虚拟化具备负载均衡，高可用，容错，迁移和数据保护等经过生产实践检验的成熟保障机制，VMware可承诺虚拟机99.999%高可用，保证业务连续性。
  - 8. 快速创建、删除：虚拟化创建是分钟级别的，Docker容器创建是秒级别的，Docker的快速迭代性，决定了无论是开发、测试、部署都可以节约大量时间。
  - 9. 交付、部署：虚拟机可以通过镜像实现环境交付的一致性，但镜像分发无法体系化；Docker在Dockerfile中记录了容器构建过程，可在集群中实现快速分发和快速部署；
- 5. 同一个宿主机中多个Docker容器之间如何通信？多个宿主机中Docker容器之间如何通信？
  - 1、这里同主机不同容器之间通信主要使用Docker桥接（Bridge）模式。
  - 2、不同主机的容器之间的通信可以借助于 pipework 这个工具。

## 1、简历：

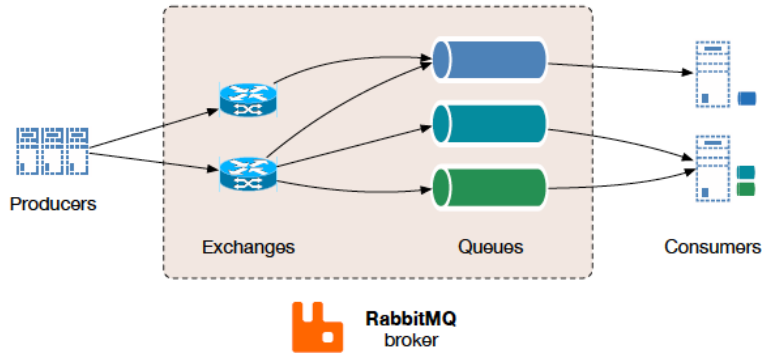
- 1. 介绍简历上主要项目，画框架图，说流程。

2. 针对简历上的技能进行提问。

2、队列：

3. 说说rabbitmq的结构。

a. 消息处理过程：



b. 四种交换机：

- 直连交换机，Direct exchange：带路由功能的交换机，根据routing\_key（消息发送的时候需要指定）直接绑定到队列，一个交换机也可以通过过个routing\_key绑定多个队列。
- 扇形交换机，Fanout exchange：广播消息。
- 主题交换机，Topic exchange：发送到主题交换机上的消息需要携带指定规则的routing\_key，主题交换机会根据这个规则将数据发送到对应的(多个)队列上。
- 首部交换机，Headers exchange：首部交换机是忽略routing\_key的一种路由方式。路由器和交换机路由的规则是通过Headers信息来交换的，这个有点像HTTP的Headers。将一个交换机声明成首部交换机，绑定一个队列的时候，定义一个Hash的数据结构，消息发送的时候，会携带一组hash数据结构的信息，当Hash的内容匹配上的时候，消息就会被写入队列。

4. rabbitmq队列与消费者的关系？

- 一个队列可以绑定多个消费者；
- 消息分发：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）。

5. rabbitmq交换器种类。

- fanout交换器：它会把所有发送到该交换器的消息路由到所有与该交换器绑定的队列中；
- direct交换器：direct类型的交换器路由规则很简单，它会把消息路由到哪些BindingKey和RoutingKey完全匹配的队列中；
- topic交换器：匹配规则比direct更灵活。
- headers交换器：根据发送消息内容的headers属性进行匹配（由于性能很差，不实用）。

6. 项目中哪里用到了kafka，kafka特性？

- 场景：
  - 大数据部门流数据处理；
  - elk；
- 特性：
  - 它被设计为一个分布式系统，易于向外扩展；
  - 它同时为发布和订阅提供高吞吐量；
  - 它支持多订阅者，当失败时能自动平衡消费者；
  - 它将消息持久化到磁盘，因此可用于批量消费，例如ETL，以及实时应用程序。

7. rabbitmq、RocketMq、kafka对比。

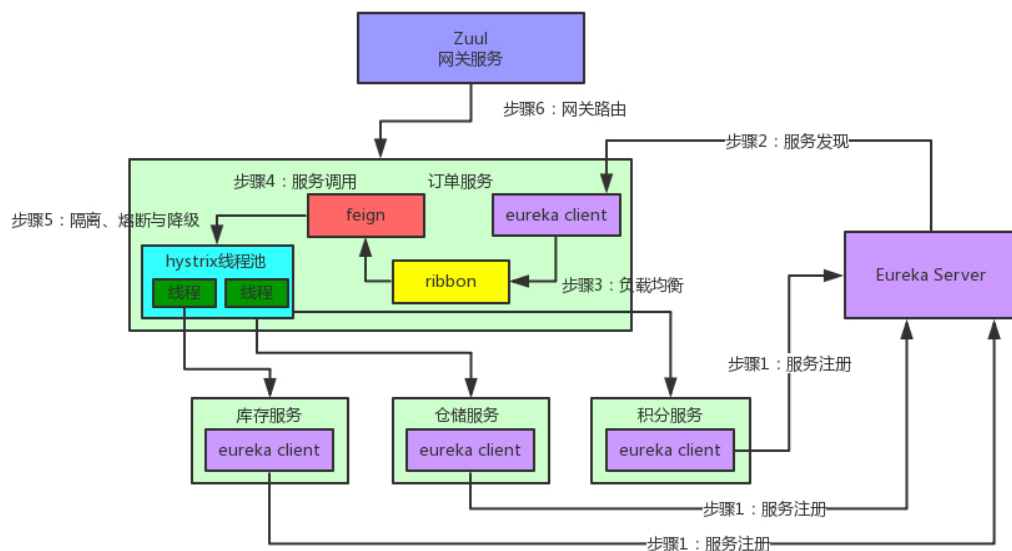
| 特性    | ActiveMQ                        | RabbitMQ                              | RocketMQ      | kafka  |
|-------|---------------------------------|---------------------------------------|---------------|--|
| 开发语言  | java                            | erlang                                | java          | scala  |
| 单机吞吐量 | 万级                              | 万级                                    | 10万级          | 10万级   |
| 时效性   | ms级                             | us级                                   | ms级           | ms级以内  |
| 可用性   | 高(主从架构)                         | 高(主从架构)                               | 非常高(分布式架构)    | 非常高(分布式架构)   |
| 功能特性  | 成熟的产品，在很多公司得到应用；有较多的文档；各种协议支持较好 | 基于erlang开发，所以并发能力很强，性能极好，延时很低；管理界面较丰富 | MQ功能比较完备，扩展性佳 | 只支持主要的MQ功能，像一些消息查询，消息回溯等功能没有提供，毕竟是为大数据准备的，在大数据领域应用广。 |

1. 中小型公司首选RabbitMQ：管理界面简单，高并发。
2. 大型公司可以选择RocketMQ：更高并发，可对rocketmq进行定制化开发。
3. 日志采集功能，首选kafka，专为大数据准备。

### 3、SpringCloud：

#### 8. 介绍springcloud核心组件及其作用，以及springcloud工作流程。

1.



springcloud由以下几个核心组件构成：

**Eureka**：各个服务启动时，Eureka Client都会将服务注册到Eureka Server，并且Eureka Client还可以反过来从Eureka Server拉取注册表，从而知道其他服务在哪里

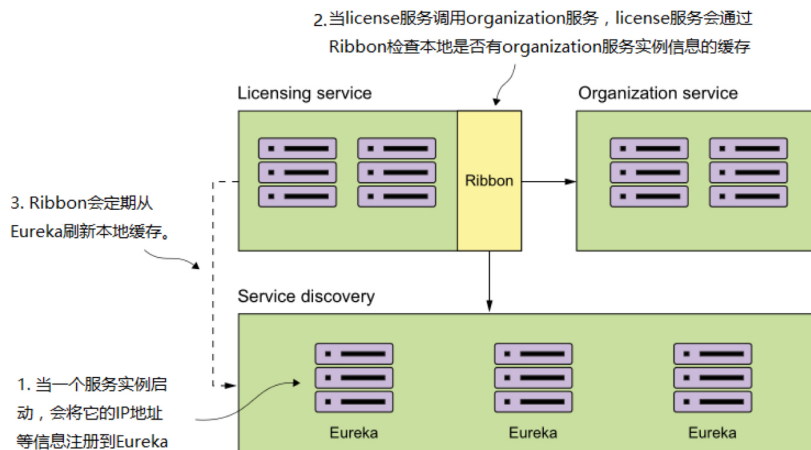
**Ribbon**：服务间发起请求的时候，基于Ribbon做负载均衡，从一个服务的多台机器中选择一台

**Feign**：基于Feign的动态代理机制，根据注解和选择的机器，拼接请求URL地址，发起请求

**Hystrix**：发起请求是通过Hystrix的线程池来走的，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题

**Zuul**：如果前端、移动端要调用后端系统，统一从Zuul网关进入，由Zuul网关转发请求给对应的服务

#### 9. 介绍springcloud心跳机制，以及消费端如何发现服务端（Ribbon）？

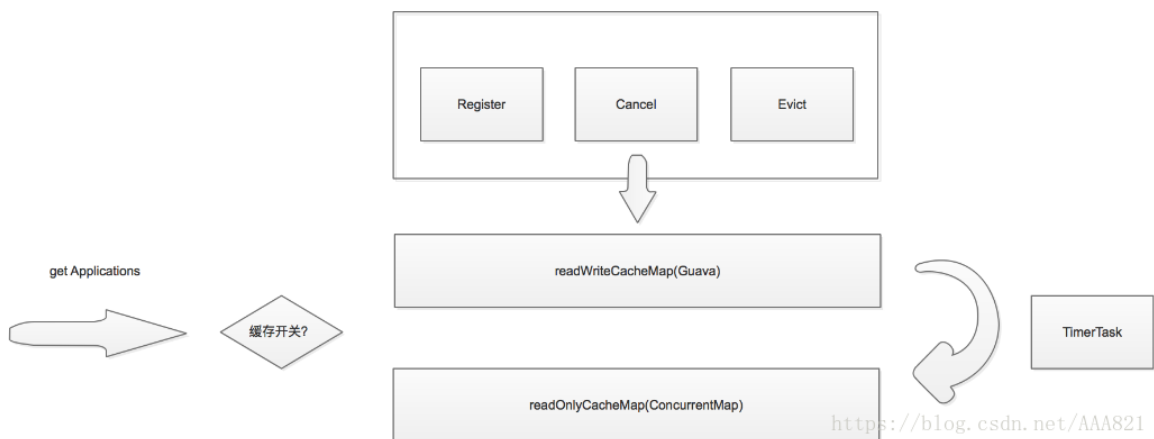


- a. 当一个服务实例启动，会将它的ip地址等信息注册到eureka；
- b. 当a服务调用b服务，a服务会通过Ribbon检查本地是否有b服务实例信息的缓存；
- c. Ribbon会定期从eureka刷新本地缓存。

#### 10. eureka的缺点。

- a. 某个服务不可用时，各个Eureka Client不能及时的知道，需要1~3个心跳周期才能感知，但是，由于基于Netflix的服务调用端都会使用Hystrix来容错和降级，当服务调用不可用时Hystrix也能及时感知到，通过熔断机制来降级服务调用，因此弥补了基于客户端服务发现的时效性的缺点。

#### 11. eureka缓存机制？



- a. 第一层缓存：readOnlyCacheMap，本质上是ConcurrentHashMap：这是一个JVM的CurrentHashMap只读缓存，这个主要是为了供客户端获取注册信息时使用，其缓存更新，依赖于定时器的更新，通过和readWriteCacheMap的值做对比，如果数据不一致，则以readWriteCacheMap的数据为准。readOnlyCacheMap缓存更新的定时器时间间隔，默认为30秒
- b. 第二层缓存：readWriteCacheMap，本质上是Guava缓存：此处存放的是最终的缓存，当服务下线，过期，注册，状态变更，都会来清除这个缓存里面的数据。然后通过CacheLoader进行缓存加载，在进行readWriteCacheMap.get(key)的时候，首先看这个缓存里面有没有该数据，如果没有则通过CacheLoader的load方法去加载，加载成功之后将数据放入缓存，同时返回数据。readWriteCacheMap缓存过期时间，默认为180秒。
- c. 缓存机制：设置了一个每30秒执行一次的定时任务，定时去服务端获取注册信息。获取之后，存入本地内存。

#### 12. rpc和http的区别，使用场景？

##### a. 区别：

- 传输协议
  - RPC，可以基于TCP协议，也可以基于HTTP协议
  - HTTP，基于HTTP协议
- 传输效率
  - RPC，使用自定义的TCP协议，可以让请求报文体积更小，或者使用HTTP2协议，也可以很好的减少报文的体积，提

高传输效率

- HTTP，如果是基于HTTP1.1的协议，请求中会包含很多无用的内容，如果是基于HTTP2.0，那么简单的封装以下是可以作为一个RPC来使用的，这时标准RPC框架更多的是服务治理
- 性能消耗，主要在于序列化和反序列化的耗时
  - RPC，可以基于thrift实现高效的二进制传输
  - HTTP，大部分是通过json来实现的，字节大小和序列化耗时都比thrift要更消耗性能
- 负载均衡
  - RPC，基本都自带了负载均衡策略
  - HTTP，需要配置Nginx，HAProxy来实现
- 服务治理（下游服务新增，重启，下线时如何不影响上游调用者）
  - RPC，能做到自动通知，不影响上游
  - HTTP，需要事先通知，修改Nginx/HAProxy配置

**b. 总结：**RPC主要用于公司内部的服务调用，性能消耗低，传输效率高，服务治理方便。HTTP主要用于对外的异构环境，浏览器接口调用，APP接口调用，第三方接口调用等。

### 13. 分布式事务如何保持一致性？

#### 1. 二阶段提交：

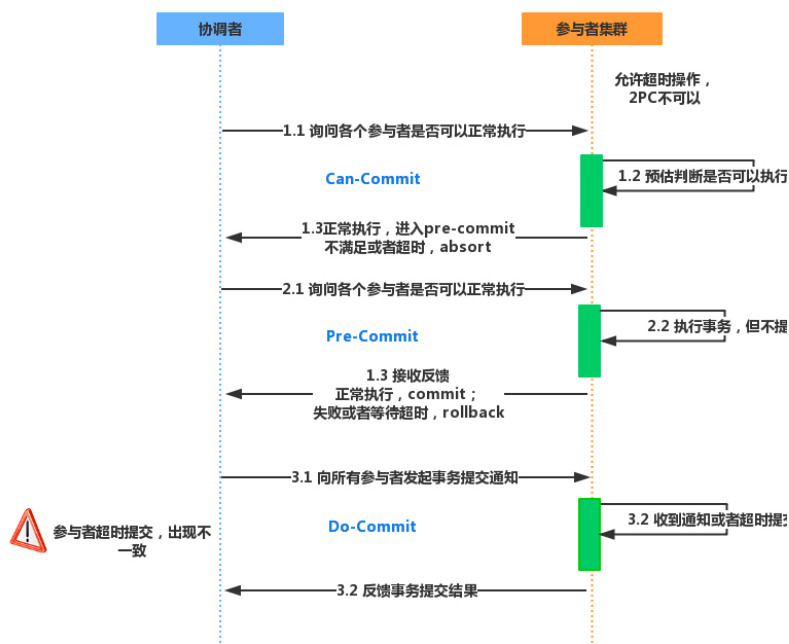
- 概念：参与者将操作成败通知协调者，再由协调者根据所有参与者的反馈情报决定各参与者是否要提交操作还是中止操作。
- 作用：主要保证了分布式事务的原子性；第一阶段为准备阶段，第二阶段为提交阶段；



- 缺点：不仅要锁住参与者的所有资源，而且要锁住协调者资源，开销大。一句话总结就是：2PC效率很低，对高并发很不友好。

#### 2. 三阶段提交：

- 概念：三阶段提交协议在协调者和参与者中都引入超时机制，并且把两阶段提交协议的第一个阶段拆分成了两步：询问，然后再锁资源，最后真正提交。这样三阶段提交就有CanCommit、PreCommit、DoCommit三个阶段。



b. 缺点：如果进入PreCommit后，Coordinator发出的是abort请求，假设只有一个Cohort收到并进行了abort操作，而其他对于系统状态未知的Cohort会根据3PC选择继续Commit，此时系统状态发生不一致性。

### 3. 柔性事务：

a. 概念：所谓柔性事务是相对强制锁表的刚性事务而言。流程如下：服务器A的事务如果执行顺利，那么事务A就先行提交，如果事务B也执行顺利，则事务B也提交，整个事务就算完成。但是如果事务B执行失败，事务B本身回滚，这时事务A已经被提交，所以需要执行一个补偿操作，将已经提交的事务A执行的操作作反操作，恢复到未执行前事务A的状态。

b. 缺点：业务侵入性太强，还要补偿操作，缺乏普遍性，没法大规模推广。

### 4. 消息最终一致性解决方案之RabbitMQ实现：

a. 实现：发送方确认+消息持久化+消费者确认。

## 14. 什么情况下用到分布式开发？

### a. 优点：

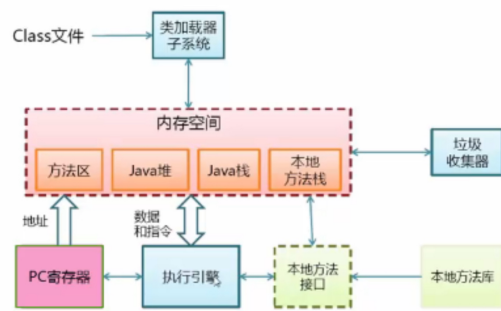
- 模块解耦：把模块拆分，使用接口通信，降低模块之间的耦合度。
- 项目拆分，不同团队负责不同的子项目：把项目拆分成若干个子项目，不同的团队负责不同的子项目。
- 提高项目扩展性：增加功能时只需要再增加一个子项目，调用其他系统的接口就可以。
- 分布式部署：可以灵活的进行分布式部署。
- 提高代码的复用性：比如service层，如果不采用分布式rest服务方式架构就会在手机wap商城，微信商城，pc，android，ios每个端都要写一个service层逻辑，开发量大，难以维护一起升级，这时候就可以采用分布式rest服务方式，公用一个service层。

### b. 缺点：

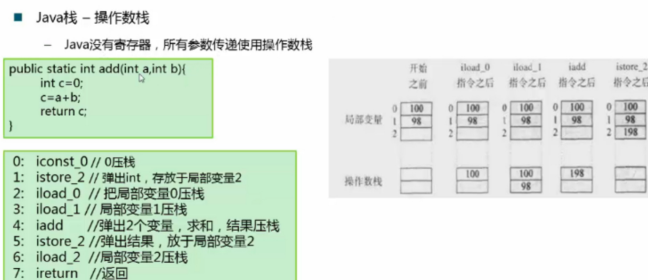
- 系统之间的交互要使用远程通信，接口开发增大工作量；
- 网络请求有延时；
- 事务处理比较麻烦，需要使用分布式事务。

## 4. jvm：

### 15. jvm内存模型，各个部分的特点？



1. PC寄存器：
  - a. 每个线程拥有一个pc寄存器；
  - b. 指向下一条指令的地址。
2. 方法区：
  - a. 保存装载的类的元信息：类型的常量池，字段、方法信息，方法字节码；
- jdk6时，String等常量信息置于方法区，jdk7移到了堆中；
  - b. 通常和永久区（Perm）关联在一起；
3. 堆：
  - a. 应用系统对象都保存在java堆中；
  - b. 所有线程共享java堆；
  - c. 对分代GC来说，堆也是分代的；
4. 栈：
  - a. 线程私有；
  - b. 栈由一系列帧组成（因此java栈也叫做帧栈）；
  - c. 帧保存一个方法的局部变量（局部变量表）、操作数栈、常量池指针；



- d. 每一次方法调用创建一个帧，并压栈。

## 16. 类加载器，双亲委派模型？

1. Bootstrap ClassLoader 启动ClassLoader
2. Extension ClassLoader 扩展ClassLoader
3. App ClassLoader 应用ClassLoader/系统ClassLoader
4. Custom ClassLoader 自定义ClassLoader

除了Bootstrap ClassLoader，每个ClassLoader都有一个Parent作为父亲。

1. 自底向上检查类是否已经加载；

## 2. 自顶向下尝试加载类。

5、双亲委派机制：当一个类收到了类加载请求，他首先不会尝试自己去加载这个类，而是把这个请求委派给父类去完成，每一个层次类加载器都是如此，因此所有的加载请求都应该传送到启动类加载器中，只有当父类加载器反馈自己无法完成这个请求的时候（在它的加载路径下没有找到所需加载的Class），子类加载器才会尝试自己去加载。



## 17. 类加载机制。

1. 概念：虚拟机把描述类的数据文件（字节码）加载到内存，并对数据进行验证、准备、解析以及类初始化，最终形成可以被虚拟机直接使用的java类型（`java.lang.Class`对象）。

2. 类生命周期：

类加载过程：读取二进制字节流到jvm—>验证格式语义等—>为静态变量分配内存空间—>常量池引用解析—>执行static标识的代码

a. 加载过程：通过一个类的全限定名来获取定义此类的二进制字节流，将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。在内存中(方法区)生成一个代表这个类的`java.lang.Class`对象，作为方法区这个类的各种数据的访问入口；

b. 验证过程：为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，文件格式验证、元数据验证、字节码验证、符号引用验证；

c. 准备过程：正式为类属性分配内存并设置类属性初始值的阶段，这些内存都将在方法区中进行分配；

准备阶段，`static`对象会被设置默认值，`static final`对象会被赋予给予的值。

d. 解析阶段：虚拟机将常量池内的符号引用替换为直接引用的过程。

i. 符号引用：字符串，引用对象不一定被加载；

ii. 直接引用：指针或者地址偏移量，引用对象一定在内存中。

e. 初始化阶段：类初始化阶段是类加载过程的最后一步。初始化阶段就是执行类构造器`<clinit>()`方法的过程。

f. 使用阶段：

g. 卸载阶段：

18. `java`堆的结构，一个`bean`被`new`出来之后，在内存空间的走向？



1、JVM中堆空间可以分成三个大区，新生代、老年代、永久代

2、新生代可以划分为三个区，Eden区，两个Survivor区，在HotSpot虚拟机Eden和Survivor的大小比例为8:1

## 19. 如何让栈溢出，如何让方法区溢出？

1. 运行时产生大量的类去填满方法区，直到溢出。

## 20. 写出几个jvm优化配置参数。

1. 设定堆内存大小，这是最基本的。

2. `-Xms`：启动JVM时的堆内存空间。

3. `-Xmx`：堆内存最大限制。

4. 设定新生代大小。

5. 新生代不宜太小，否则会有大量对象涌入老年代。



- 6. -XX:NewRatio: 新生代和老年代的占比。
- 7. -XX:NewSize: 新生代空间。
- 8. -XX:SurvivorRatio: 伊甸园空间和幸存者空间的占比。
- 9. -XX:MaxTenuringThreshold: 对象进入老年代的年龄阈值。

#### 10. 设定垃圾回收器

年轻代: -XX:+UseParNewGC。

老年代: -XX:+UseConcMarkSweepGC。

CMS可以将STW时间降到最低,但是不对内存进行压缩,有可能出现“并行模式失败”。比如老年代空间还有300MB空间,但是有一些10MB的对象无法被顺序的存储。这时候会触发压缩处理,但是CMS GC模式下的压缩处理时间要比Parallel GC长很多。

G1采用“标记-整理”算法,解决了内存碎片问题,建立了可预测的停顿时间类型,能让使用者指定在一个长度为M毫秒的时间段内,消耗在垃圾收集上的时间不得超过N毫秒。

#### 21. 有哪几种GC机制?

##### 1. 引用计数法(没有被java采用):

- a. 原理: 对于一个对象A,只要有任意一个对象引用了A,则A的引用计数器就加1,当引用失效时,引用计数器就减1,只要对象A的引用计数器的值为0,则对象A就会被回收。
- b. 问题:
  - i. 引用和去引用伴随加法和减法,影响性能;
  - ii. 很难处理循环引用。

##### 2. 标记清除法:

- a. 原理: 现代垃圾回收算法的思想基础。标记-清除算法将垃圾回收分为两个阶段: 标记阶段和清除阶段。一种可行的实现是,在标记节点,首先通过根节点,标记所有从根节点开始的可达对象。因此,未被标记的对象就是未被引用的垃圾对象。然后在清除阶段,清除所有未被标记的对象。
- b. 问题:
  - i. 标记和清除两个过程效率不高,产生内存碎片导致需要分配较大对象时无法找到足够的连续内存而需要触发一次GC操作。

##### 3. 标记压缩法:

- a. 原理: 适合用于存活对象较多的场合,如老年代。它在标记-清除算法的基础上做了一些优化。标记阶段一样,但之后,将所有存活对象压缩到内存的一端。之后,清除边界外所有的空间。
- b. 优点:
  - i. 解决了标记-清除算法导致的内存碎片问题和在存活率较高时复制算法效率低的问题。

##### 4. 复制算法:

- a. 原理: 将原有的内存空间分为两块,每次只使用其中一块,在垃圾回收时,将正在使用的内存中的存活对象复制到未使用的内存块中,之后清除正在使用的内存块中的所有对象,交换两个内存的角色,完成垃圾回收。
- b. 问题:
  - i. 不适用于存活对象比较多的场合,如老年代。

##### 5. 分代回收法:

- a. 原理: 根据对象存活周期的不同将内存划分为几块,一般是新生代和老年代, **新生代基本采用复制算法,老年代采用标记整理算法。**

#### 5、spring:

##### 22. springboot启动过程。



1. 通过 SpringFactoriesLoader加载 META-INF/spring.factories文件，获取并创建 SpringApplicationRunListener对象

2. 然后由 SpringApplicationRunListener来发出 starting 消息

3. 创建参数，并配置当前 SpringBoot 应用将要使用的 Environment

4. 完成之后，依然由 SpringApplicationRunListener来发出 environmentPrepared 消息

5. 创建 ApplicationContext

6. 初始化 ApplicationContext，并设置 Environment，加载相关配置等

7. 由 SpringApplicationRunListener来发出 contextPrepared消息，告知SpringBoot 应用使用的 ApplicationContext已准备OK

8. 将各种 beans 装载入 ApplicationContext，继续由 SpringApplicationRunListener来发出 contextLoaded 消息，告知 SpringBoot 应用使用的 ApplicationContext已装填OK

9. refresh ApplicationContext，完成IoC容器可用的最后一步

10. 由 SpringApplicationRunListener来发出 started 消息

11. 完成最终的程序的启动

12. 由 SpringApplicationRunListener来发出 running 消息，告知程序已运行起来了

## 23. 说说几个常用的注解？

**@RestController**：@ResponseBody和@Controller的合集。

**@EnableAutoConfiguration**：尝试根据你添加的jar依赖自动配置你的Spring应用。

**@ComponentScan**：表示将该类自动发现（扫描）并注册为Bean，可以自动收集所有的Spring组件，包括@Configuration类。

**@ImportResource**：用来加载xml配置文件。

**@Configuration**：相当于传统的xml配置文件，如果有些第三方库需要用到xml文件，建议仍然通过@Configuration类作为项目的配置主类——可以使用@ImportResource注解加载xml配置文件。

**@SpringBootApplication**：相当于@EnableAutoConfiguration、@ComponentScan和@Configuration的合集。

## 24. spring事件的实现原理，写出常用的几个事件。

1. 事件实现原理：理解这篇即可：<https://www.jianshu.com/p/dcbe8f0afbdc>

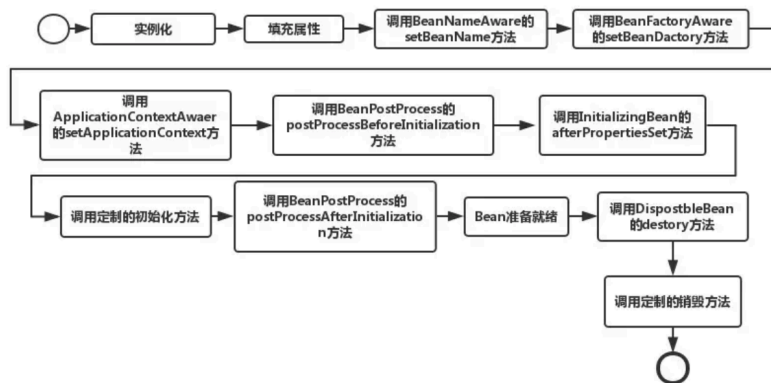
2. 常用事件：

(1)ContextRefreshedEvent:当ApplicationContext初始化或者刷新时触发该事件

(2)ContextClosedEvent:ApplicationContext被关闭时触发该事件.容器被关闭时,其管理的所有单例Bean都被销毁

- (3) RequestHandleEvent: 在Web应用中, 当一个Http请求结束时触发该事件
- (4) ContextStartedEvent: 当容器调用start()方法时触发
- (5) ContextStopEvent: 当容器调用stop()方法时触发

## 25. spring的bean的生命周期?



## 26. BeanFactory和FactoryBean的区别。

详见“6.1、框架—Spring”

## 27. spring中使用到了FactoryBean的哪个方法?

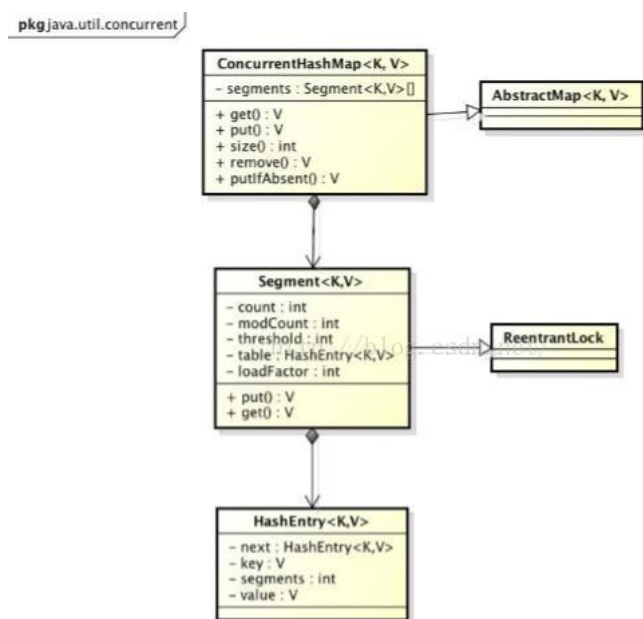
详见“6.1、框架—Spring”

## 6、数据结构:

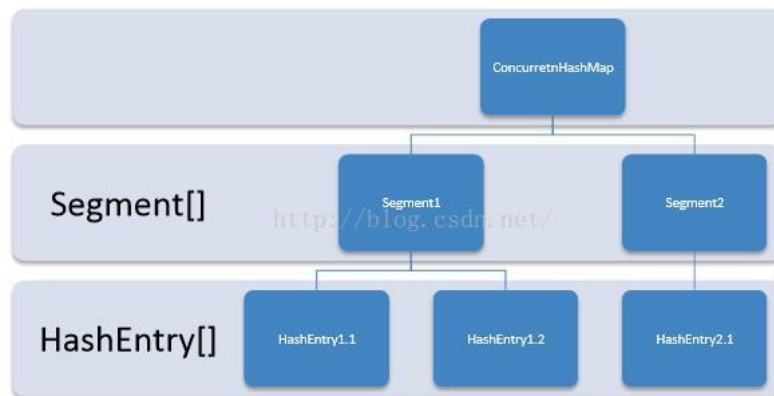
### 28. 说说HashMap、ConcurrentHashMap数据结构, 1.7与1.8的区别?

#### 1、数据结构:

以下是ConcurrentHashMap的类图:



ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁ReentrantLock, 在ConcurrentHashMap中扮演锁的角色, HashEntry则用于存储键值对数据。一个ConcurrentHashMap中包含一个Segment数组, Segment的结构和HashMap类似, 是一种数组和链表结构。一个Segment里面包含一个HashEntry数组, 每个HashEntry是一个链表的元素。每个Segment拥有一个锁, 当对HashEntry数组的数据进行修改时, 必须先获得对应的Segment锁, 如图所示:



2、concurrenthashmap 1.7和1.8区分：

- 去除 Segment + HashEntry + Unsafe的实现，  
改为 Synchronized + CAS + Node + Unsafe的实现  
其实 Node 和 HashEntry 的内容一样，但是HashEntry是一个内部类。  
用 Synchronized + CAS 代替 Segment，这样锁的粒度更小了，并且不是每次都要加锁了，CAS尝试失败了在加锁。
- put()方法中 初始化数组大小时，1.8不用加锁，因为用了个 sizeCtl变量，将这个变量置为-1，就表明table正在初始化。

29. 谈谈数据结构，比如TreeMap。

详见：<https://my.oschina.net/u/566591/blog/1548176>

30. B-tree、B+tree?

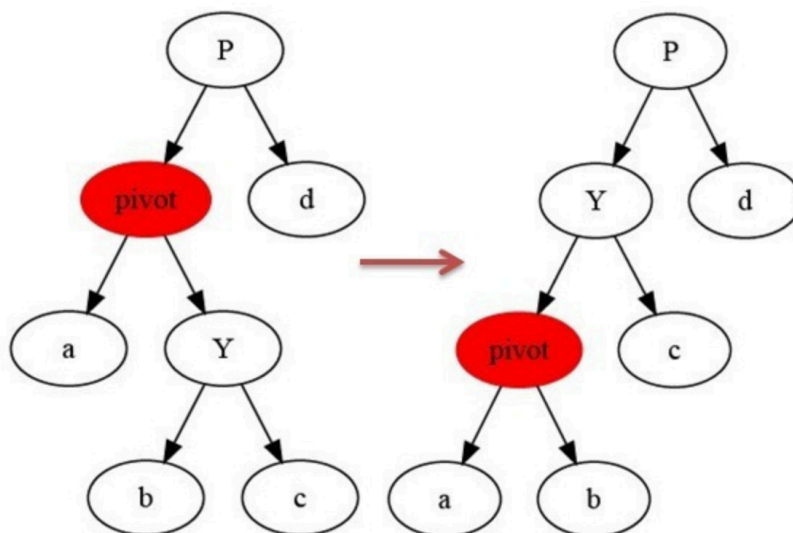
详见“面试题库/java基础”

31. 红黑树左旋与右旋的区别？

当在对红黑树进行插入和删除等操作时，对树做了修改可能会破坏红黑树的性质。为了继续保持红黑树的性质，可以通过对结点进行重新着色，以及对树进行相关的旋转操作，即通过修改树中某些结点的颜色及指针结构，来达到对红黑树进行插入或删除结点等操作后继续保持它的性质或平衡的目的。

树的旋转分为左旋和右旋，下面借助图来介绍一下左旋和右旋这两种操作。

1.左旋



如上图所示，当在某个结点pivot上，做左旋操作时，我们假设它的右孩子y不是NIL[T]，pivot可以为任何不是NIL[T]的左子结点。左旋以pivot到Y之间的链为“支轴”进行，它使Y成为孩子树的新根，而Y的左孩子b则成为pivot的右孩子。

```
1 LeftRotate(T, x)
```

```
2 y ← x.right
```

//定义y: y是x的右孩子

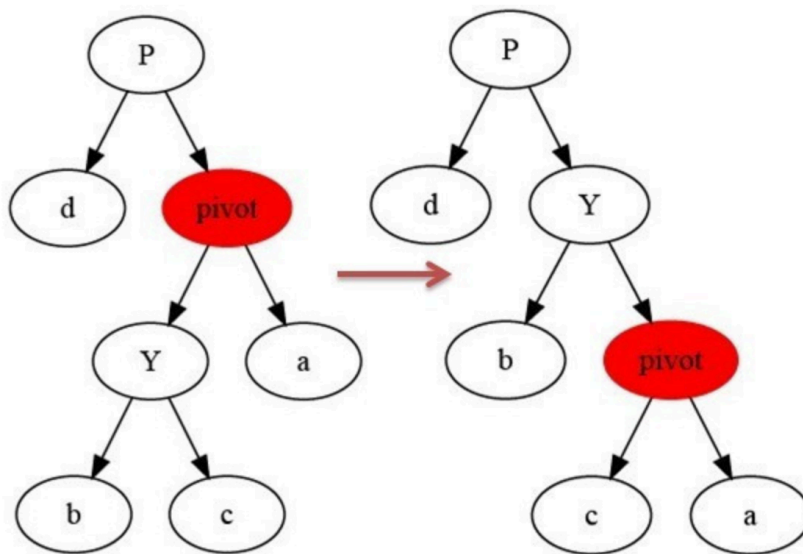
```

3 x.right ← y.left           //y的左孩子成为x的右孩子
4 if y.left ≠ T.nil
5     y.left.p ← x
6 y.p ← x.p                 //x的父结点成为y的父结点
7 if x.p = T.nil
8     then T.root ← y
9 else if x = x.p.left
10    then x.p.left ← y
11 else x.p.right ← y
12 y.left ← x               //x作为y的左孩子
13 x.p ← y

```

2.右旋

右旋与左旋差不多，再此不做详细介绍。



树在经过左旋右旋之后，树的搜索性质保持不变，但树的红黑性质则被破坏了，所以，红黑树插入和删除数据后，需要利用旋转与颜色重涂来重新恢复树的红黑性质。

7、并发：

32. concurrent包下有哪些常用类？

1.CountDownLatch:api解释：一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。个人理解是CountDownLatch可以让一组线程同时执行，然后在这组线程全部执行完前，可以让另一个线程等待。

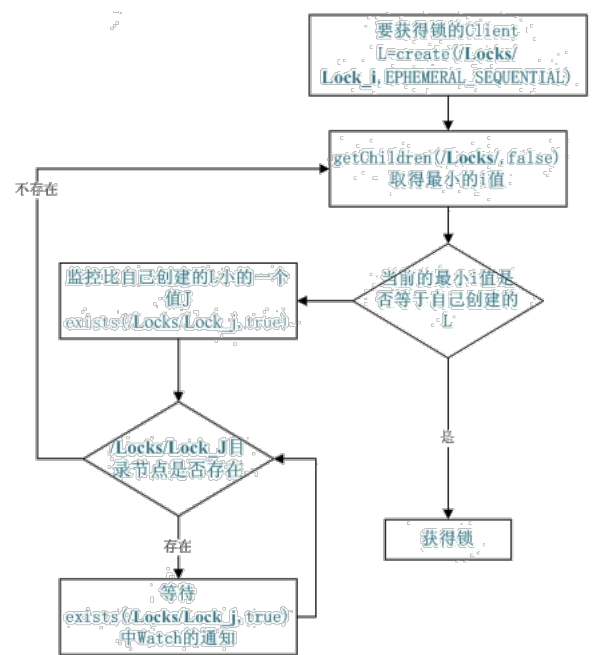
2.ReentrantLock：可重入互斥锁

3.Condition：此类是同步的条件对象，每个Condition实例绑定到一个ReentrantLock中，以便争用同一个锁的多线程之间可以通过Condition的状态来获取通知。

注意：使用Condition前，首先要获得ReentrantLock，当条件不满足线程1等待时，ReentrantLock会被释放，以能让其他线程争用，其他线程获得reentrantLock，然后满足条件，唤醒线程1继续执行。

33. 三种分布式锁。

1、Zookeeper：基于zookeeper瞬时有序节点实现的分布式锁，其主要逻辑如下（该图来自于IBM网站）。大致思想即为：每个客户端对某个功能加锁时，在zookeeper上的与该功能对应的指定节点的目录下，生成一个唯一的瞬时有序节点。判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。



## 2、优点

锁安全性高，zk可持久化，且能实时监听获取锁的客户端状态。一旦客户端宕机，则瞬时节点随之消失，zk因而能第一时间释放锁。这也省去了用分布式缓存实现锁的过程中需要加入超时时间判断的这一逻辑。

## 3、缺点

性能开销比较高。因为其需要动态产生、销毁瞬时节点来实现锁功能。所以不太适合直接提供给高并发的场景使用。

## 4、实现

可以直接采用zookeeper第三方库curator即可方便地实现分布式锁。

## 5、适用场景

对可靠性要求非常高，且并发程度不高的场景下使用。如核心数据的定时全量/增量同步等。

2、memcached: memcached带有add函数，利用add函数的特性即可实现分布式锁。add和set的区别在于：如果多线程并发set，则每个set都会成功，但最后存储的值以最后的set的线程为准。而add的话则相反，add会添加第一个到达的值，并返回true，后续的添加则都会返回false。利用该点即可很轻松地实现分布式锁。

## 2、优点

并发高效

## 3、缺点

memcached采用列入LRU置换策略，所以如果内存不够，可能导致缓存中的锁信息丢失。

memcached无法持久化，一旦重启，将导致信息丢失。

## 4、使用场景

高并发场景。需要 1) 加上超时时间避免死锁；2) 提供足够支撑锁服务的内存空间；3) 稳定的集群化管理。

3、redis: redis分布式锁即可以结合zk分布式锁锁高度安全和memcached并发场景下效率很好的优点，其实现方式和memcached类似，采用setnx即可实现。需要注意的是，这里的redis也需要设置超时时间。以避免死锁。可以利用jedis客户端实现。

```

1 ICacheKey cacheKey = new ConcurrentCacheKey(key, type);
2 return RedisDao.setnx(cacheKey, "1");

```

## 8、线程池：

### 34. 你知道哪些常用的阻塞队列？

JDK7 提供了 7 个阻塞队列。分别是

- ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列。
- LinkedBlockingQueue：一个由链表结构组成的有界阻塞队列。
- PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列。
- DelayQueue：一个使用优先级队列实现的无界阻塞队列。
- SynchronousQueue：一个不存储元素的阻塞队列。
- LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
- LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

### 35. newFixedThreadPool使用到了哪个阻塞队列？

看Executors源码可知：newFixedThreadPool使用了LinkedBlockingQueue。

```
1 public static ExecutorService newFixedThreadPool(int var0) {  
2     return new ThreadPoolExecutor(var0, var0, 0L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue()  
3 }
```

## 9、数据库：

### 36. 说说mysql存储引擎innodb和myisam的区别和使用场景。

#### a. InnoDB：

1. 支持事务处理
2. 支持外键
3. 支持行锁
4. 不支持FULLTEXT类型的索引（在Mysql5.6已引入）
5. 不保存表的具体行数，扫描表来计算有多少行
6. 对于AUTO\_INCREMENT类型的字段，必须包含只有该字段的索引
7. DELETE 表时，是一行一行的删除
8. InnoDB 把数据和索引存放在表空间里面
9. 跨平台可直接拷贝使用
10. 表格很难被压缩

#### b. MyISAM：

1. 不支持事务，回滚将造成不完全回滚，不具有原子性
2. 不支持外键
3. 支持全文搜索
4. 保存表的具体行数,不带where时，直接返回保存的行数
5. DELETE 表时，先drop表，然后重建表
6. MyISAM 表被存放在三个文件。frm 文件存放表格定义。数据文件是MYD (MYData)。索引文件是MYI (MYIndex)索引
7. 跨平台很难直接拷贝
8. AUTO\_INCREMENT类型字段可以和其他字段一起建立联合索引
9. 表格可以被压缩

c. 选择：因为MyISAM相对简单所以在效率上要优于InnoDB.如果系统读多，写少。对原子性要求低。那么MyISAM最好的选择。  
且MyISAM恢复速度快。可直接用备份覆盖恢复。如果系统读少，写多的时候，尤其是并发写入高的时候。InnoDB就是首选了。  
两种类型都有自己优缺点，选择那个完全要看自己的实际类弄。

### 37. 说说mysql查询优化。

- 1、选择最合适的字段属性：类型、长度、是否允许NULL等；尽量把字段设为not null，一面查询时对比是否为null；
- 2.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。
- 3.应尽量避免在 where 子句中对字段进行 null 值判断、使用!= 或 <> 操作符，否则将导致引擎放弃使用索引而进行全表扫描
- 4.应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全

表扫描

- 5.in 和 not in 也要慎用，否则会导致全表扫描
- 6.模糊查询也将导致全表扫描，若要提高效率，可以考虑字段建立前置索引或用全文检索；
- 7.如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。
- 9.应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。
- 10.不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。
- 11.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。
- 12.不要写一些没有意义的查询，如需要生成一个空表结构：
- 13.Update 语句，如果只更改1、2个字段，不要Update全部字段，否则频繁调用会引起明显的性能消耗，同时带来大量日志。
- 14.对于多张大数据量（这里几百条就算大了）的表JOIN，要先分表再JOIN，否则逻辑读会很高，性能很差。
- 15.select count(\*) from table; 这样不带任何条件的count会引起全表扫描，并且没有任何业务意义，是一定要杜绝的。
- 16.索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能要重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。
- 17.应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。
- 18.尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。



19.尽可能的使用 `varchar/nvarchar` 代替 `char/nchar` , 因为首先变长字段存储空间小, 可以节省存储空间, 其次对于查询来说, 在一个相对较小的字段内搜索效率显然要高些。

20.任何地方都不要使用 `select * from t` , 用具体的字段列表代替“\*”, 不要返回用不到的任何字段。

21.尽量使用表变量来代替临时表。如果表变量包含大量数据, 请注意索引非常有限 (只有主键索引)。

22. 避免频繁创建和删除临时表, 以减少系统表资源的消耗。临时表并不是不可使用, 适当地使用它们可以使某些例程更有效, 例如, 当需要重复引用大型表或常用表中的某个数据集时。但是, 对于一次性事件, 最好使用导出表。

23.在新建临时表时, 如果一次性插入数据量很大, 那么可以使用 `select into` 代替 `create table`, 避免造成大量 `log` , 以提高速度; 如果数据量不大, 为了缓和系统表的资源, 应先`create table`, 然后`insert`。

24.如果使用到了临时表, 在存储过程的最后务必将所有的临时表显式删除, 先 `truncate table` , 然后 `drop table` , 这样可以避免系统表的较长时间锁定。

25.尽量避免使用游标, 因为游标的效率较差, 如果游标操作的数据超过1万行, 那么就应该考虑改写。

26.使用基于游标的方法或临时表方法之前, 应先寻找基于集的解决方案来解决问题, 基于集的方法通常更有效。

27.与临时表一样, 游标并不是不可使用。对小型数据集使用 `FAST_FORWARD` 游标通常要优于其他逐行处理方法, 尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时间允许, 基于游标的方法和基于集的方法都可以尝试一下, 看哪一种方法的效果更好。

28.在所有的存储过程和触发器的开始处设置 `SET NOCOUNT ON` , 在结束时设置 `SET NOCOUNT OFF` 。无需在执行存储过程和触发器的每个语句后向客户端发送 `DONE_IN_PROC` 消息。

29.尽量避免大事务操作, 提高系统并发能力。

30.尽量避免向客户端返回大数据量, 若数据量过大, 应该考虑相应需求是否合理。

**38. 说说脏读、不可重复读、幻读；**

1. ISOLATIONREADUNCOMMITTED 这是事务最低的隔离级别, 它允许另外一个事务可以看到这个事务未提交的数据。这种隔离级别会产生脏读, 不可重复读和幻读。
2. ISOLATIONREADCOMMITTED 保证一个事务修改的数据提交后才能被另外一个事务读取。另外一个事务不能读取该事务未提交的数据。这种事务隔离级别可以避免脏读出现, 但是可能会出现不可重复读和幻读。
3. ISOLATIONREPEATABLEREAD 这种事务隔离级别可以防止脏读, 不可重复读。但是可能出现幻读。它除了保证一个事务不能读取另一个事务未提交的数据外, 还保证了避免不可重复读。
4. ISOLATION\_SERIALIZABLE 这是花费最高代价但是最可靠的事务隔离级别。事务被处理为顺序执行。除了防止脏读, 不可重复读外, 还避免了幻读。

| 隔离级别                    | 脏读 (Dirty Read) | 不可重复读 (NonRepeatable Read) | 幻读 (Phantom Read) |
|-------------------------|-----------------|----------------------------|-------------------|
| 未提交读 (Read uncommitted) | 可能              | 可能                         | 可能                |
| 已提交读 (Read committed)   | 不可能             | 可能                         | 可能                |
| 可重复读 (Repeatable read)  | 不可能             | 不可能                        | 可能                |
| 可串行化 (Serializable )    | 不可能             | 不可能                        | 不可能               |

- 未提交读(Read Uncommitted): 允许脏读, 也就是可能读取到其他会话中未提交事务修改的数据
- 提交读(Read Committed): 只能读取到已经提交的数据。Oracle等多数数据库默认都是该级别 (不重复读)
- 可重复读(Repeated Read): 可重复读。在同一个事务内的查询都是事务开始时刻一致的, InnoDB默认级别。在SQL标准中, 该隔离级别消除了不可重复读, 但是还存在幻象读
- 串行读(Serializable): 完全串行化的读, 每次读都需要获得表级共享锁, 读写相互都会阻塞

**39. 说说事务的四种特性 (ACID) 。**

- 1)原子性(Atomic): 事务中各项操作, 要么全做要么全不做, 任何一项操作的失败都会导致整个事务的失败;
- 2)一致性(Consistent): 事务结束后系统状态是一致的;
- 3)隔离性(Isolated): 并发执行的事务彼此无法看到对方的中间状态;
- 4)持久性(Durable): 事务完成后所做的改动都会被持久化, 即使发生灾难性的失败。通过日志和同步备份可以在故障发生后重建数据。

**40. codis与redis集群的区别。**

redis cluster基于smart client和无中心的设计, client必须按key的哈希将请求直接发送到对应的节点。这意味着: 使用官方cluster必须要等对应语言的redis driver对cluster支持的开发和不断成熟; client不能直接像单机一样使用pipeline来提高效率, 想同时执行多个请求来提速必须在client端自行实现异步逻辑。而codis因其有中心节点、基于proxy的设计, 对client来说可以像对单机redis一样去操作proxy (除了一些命令不支持), 还可以继续使用pipeline并且如果后台redis有多个的话速度会显著快于单redis的pipeline。同时codis使用zookeeper来作为辅助, 这意味着单纯对于redis集群来说需要额外的机器搭zk, 不过对于很多已经在其他服务上用了zk的公司来说这不是问题)

**10、设计：**

**41. 要缓存网站登录的用户信息，你有几种方式？**



cookie+session、cookie+redis。

**42.** 让你设计一套分布式缓存，如何设计可以同时更新所有服务器的缓存？

**43.** 说说你在工作中遇到的困难或者挑战。