后台开发面试

1.自我介绍

2.项目介绍

2.1什么是MD5

> 分为四步：：处理原文，设置初始值，循环加工，拼接结果。
>
> 首先，我们计算出原文长度(bit)对 512 求余的结果，如果不等于 448，就需要填充原文使得原文对 512 求余的结果等于 448。填充的方法是第一位填充 1，其余位填充 0。填充完后，信息的长度就是 512*N+448。
>
> 之后，用剩余的位置（512-448=64 位）记录原文的真正长度，把长度的二进制值补在最后。这样处理后的信息长度就是 512*(N+1)。
>
> 第二步:设置初始值
>
> MD5 的哈希结果长度为 128 位，按每 32 位分成一组共 4 组。这 4 组结果是由 4 个初始值 A、B、C、D 经过不断演变得到。

2.2介绍Shiro框架(底层实现)

> Shiro是apache旗下一个开源框架，它将软件系统的安全认证相关的功能抽取出来，实现用户身份认证，权限授权、加密、会话管理等功能，组成了一个通用的安全认证框架。

2.2.1 shiro的底层实现

> 1. 创建Subject实例对象currUser；
> 2. 判断当前currUser是否认证过；
> 3. 如果没有认证过，那么应当调用currUser的login(token)方法，token也就是令牌，用以封装用户输入的登录信息；
> 4. 实现自定义Realm，用以完成和数据库的交互，由Shiro底层来完成用户输入信息和数据库信息的比对，完成认证

5. shiro会把请求信息保存到session中：然后判断是否已经登录，如果没有登录，就会跳到登录页面，用户输入凭证之后就会交给FormAuthenticationFilter这个类来处理；如果登录成功之后就会调用一下方法重定向

1.利用用户的登陆信息生成Token

```
UsernamePasswordToken token = new UsernamePasswordToken(username, password);
```

token可以理解为用户令牌，登录的过程被抽象为Shiro验证令牌是否具有合法身份以及相关权限。

2.执行登陆动作

```
SecurityUtils.setSecurityManager(securityManager); // 注入SecurityManager
Subject subject = SecurityUtils.getSubject(); // 获取Subject单例对象
subject.login(token); // 登陆
```

3.接口和抽象类的区别

1.抽象类:如果一个类中包含了抽象方法,那么这个类就是抽象类.在Java中可以通过把某些方法声明abstract(abstract只能用来修饰类或者方法不能用来修饰属性)来表示一个类是抽象类。

2.接口就是一个方法的集合，接口中所有的方法都是没有方法体的，通过关键字interface来实现

相同点：

1）都不能被实例化

2）接口的实现类或者抽象类的子类都只有实现了接口或抽象类中的方法后才能被实例化

不同点：

1.在Java8之前接口只有定义，其方法不能在接口中实现（Java8 之后接口方法可以有默认实现），只有实现接口的类才能实现接口中定义的方法。抽象类可以定义和实现，即其方法可以在抽象类中实现 。抽象类可以有非抽象方法

2.一个类可以实现(implements)多个接口，但是最多只能实现(extends)一个抽象类。因此接口可以间接的达到多重继承的目的

3.接口强调的是特定功能的实现,设计理念是一种"has - a"的关系,而抽象类强调的所属关系,其设计理念是"is - a"的关系(ps: has -a 这种事物(羊毛)属于那种事物的一部分(绵羊)! is - a 这种事物(绵羊)属于那种事物(羊)的一个种类)

4.接口中的成员(实例)变量默认为public static final 只能够有静态的不能被修改的数据成员,而且必须给其赋初值,接口中的方法只能用public 和 abstract修饰.但是抽象类不一定

接口是一种特殊形式的抽象类,使用接口完全有可能实现与抽象类相同的操做.通常接口用于比实现比较常用的功能,便于日后维护或者添加删除方法;而抽象类更倾向于充当公共类的角色,不适用于日后重新对里面的代码进行修改.此外接口可以继承接口,抽象类可以实现接口,抽象类也可以继承具体的类,抽象类中可以有静态的main方法

4.多态

> 多态表示当同一个操作作用在不同对象时，会有不同的语义，从而产生不同的结果。3+4和“3”+“4”
>
> Java的多态性可以概括成"一个接口,两种方法"分为两种编译时的多态和运行时的多态。编译时的多态主要是指方法的重载（overload），运行时的多态主要是指方法的覆盖（override），接口也是运行时的多态
>
> 运行时的多态的三种情况：
>
> > 1、父类有方法，子类有覆盖方法：编译通过，执行子类方法。
> >
> > 2、父类有方法，子类没覆盖方法：编译通过，执行父类方法（子类继承）。
> >
> > 3、父类没方法，子类有方法：编译失败，无法执行。
>
> 方法带final、static、private时是编译时多态，因为可以直接确定调用哪个方法。

5.重载和覆盖

> 重载：发生在一个类中，方法名必须相同，参数类型不同，个数不同，顺序不同，方法返回值和访问修饰符可以不同，发生在编译时
>
> 重写：发生在父子类中，方法名，参数列表必须相同，返回值必须小于等于父类，抛出异常的范围必须小于等于父类，访问修饰符范围必须大于等于父类，父类为private则不能重写
>
> 函数时不能以返回值来区分的，返回值时函数运行之后的一个状态。保持调用这与被调用这之间的通信

6.spring

1.动态代理

> 代理类在程序运行时创建的代理方式被称为动态代理。代理类并不是在Java代码中定义的，而是在运行时根据我们在Java代码中的“指示”动态生成的。方法实现前后加入对应的公共功能

基于接口

> jdk的动态代理时基于Java 的反射机制来实现的，是Java 原生的一种代理方式。他的实现原理就是让代理类和被代理类实现同一接口，代理类持有目标对象来达到方法拦截的作用。通过接口的方式有两个弊端一个就是必须保证被代理类有接口，另一个就是如果相对被代理类的方法进行代理拦截，那么就要保证这些方法都要在接口中声明。接口继承的是java.lang.reflect.InvocationHandler;

基于继承

> cglib动态代理使用的ASM这个非常强大的Java字节码生成框架来生成class，比jdk动态代理ide效率高。基于继承的实现动态代理，可以直接通过super关键字来调用被代理类的方法.子类可以调用父类的方法

2.AOP

> 面向切面编程。（Aspect-Oriented Programming）。AOP可以说是对OOP的补充和完善。OOP引入封装、继承和多态性等概念来建立一种对象层次结构，用以模拟公共行为的一个集合。实现AOP的技术，主要分为两大类： 一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用静态织入 的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码，属于静态代理。
>
> > 1.面向切面编程提供声明式事务管理 2.spring支持用户自定义的切面

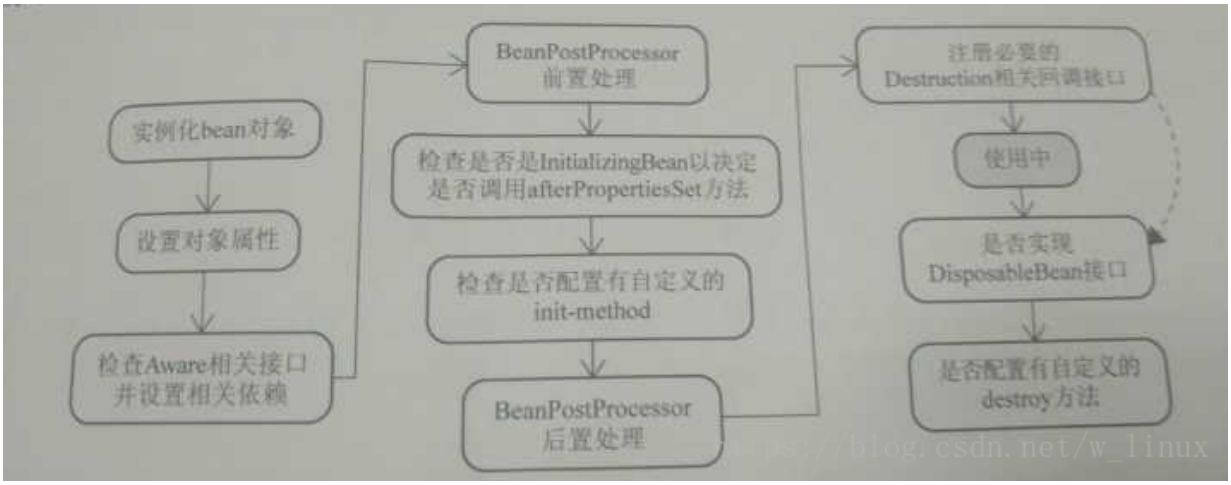
> 面向切面编程（aop）是对面向对象编程（oop）的补充， 面向对象编程将程序分解成各个层次的对象，面向切面编程将程序运行过程分解成各个切面。 AOP从程序运行角度考虑程序的结构，提取业务处理过程的切面，oop是静态的抽象，aop是动态的抽象， 是对应用执行过程中的步骤进行抽象，，从而获得步骤之间的逻辑划分。
>
> aop框架具有的两个特征： 1.各个步骤之间的良好隔离性 2.源代码无关性

2.1AOP

> springAOP 的具体加载步骤：
>
> 1、当 spring 容器启动的时候，加载了 spring 的配置文件
>
> 2、为配置文件中的所有 bean 创建对象
>
> 3、spring 容器会解析 aop:config 的配置
>
> 1、解析切入点表达式，用切入点表达式和纳入 spring 容器中的 bean 做匹配
>
> 如果匹配成功，则会为该 bean 创建代理对象，代理对象的方法=目标方法+通知
>
> 如果匹配不成功，不会创建代理对象
>
> 4、在客户端利用 context.getBean() 获取对象时，如果该对象有代理对象，则返回代理对象；如果没有，则返回目标对象
>
> 说明：如果目标类没有实现接口，则 spring 容器会采用 cglib 的方式产生代理对象，如果实现了接口，则会采用 jdk 的方式

3.IOC

> 控制反转也叫依赖注入。IOC利用java反射机制，AOP利用代理模式。IOC 概念看似很抽象，但是很容易理解。说简单点就是将对象交给容器管理，你只需要在spring配置文件中配置对应的bean以及设置相关的属性，让 spring 容器来生成类的实例对象以及管理对象。在spring容器启动的时候，spring会把你在配置文件中配置的bean都初始 化好，然后在你需要调用的时候，就把它已经初始化好的那些bean分配给你需要调用这些bean的类
>
> XML——读取---> resoure----解析------->BeanDefinition———注入----->BeanFactory
>
> 详细解析
>
> > 依赖注入（Dependecy Injection）和控制反转（Inversion of Control）是同一个概念，具体的讲：当某个角色需要另外一个角色协助的时候，在传统的程序设计过程中，通常由调用者来创建被调用者的实例。但在spring中创建被调用者的工作不再由调用者来完成，因此称为控制反转。创建被调用者的工作由spring来完成，然后注入调用者 因此也称为依赖注入。 spring以动态灵活的方式来管理对象 ， 注入的两种方式，设置注入和构造注入。 设置注入的优点：直观，自然 构造注入的优点：可以在构造器中决定依赖关系的顺序。
>
> Bean的生命周期
>
> > 可以简述为以下九步
> >
> > 实例化bean对象(通过构造方法或者工厂方法) 设置对象属性(setter等)（依赖注入） 如果Bean实现了BeanNameAware接口，工厂调用Bean的setBeanName()方法传递Bean的ID。（和下面的一条均属于检查Aware接口） 如果Bean实现了BeanFactoryAware接口，工厂调用setBeanFactory()方法传入工厂自身 将Bean实例传递给Bean的前置处理器的postProcessBeforeInitialization(Object bean, String beanname)方

> 法 调用Bean的初始化方法 将Bean实例传递给Bean的后置处理器的postProcessAfterInitialization(Object bean, String beanname)方法 使用Bean容器关闭之前，调用Bean的销毁方法
>
> 
>

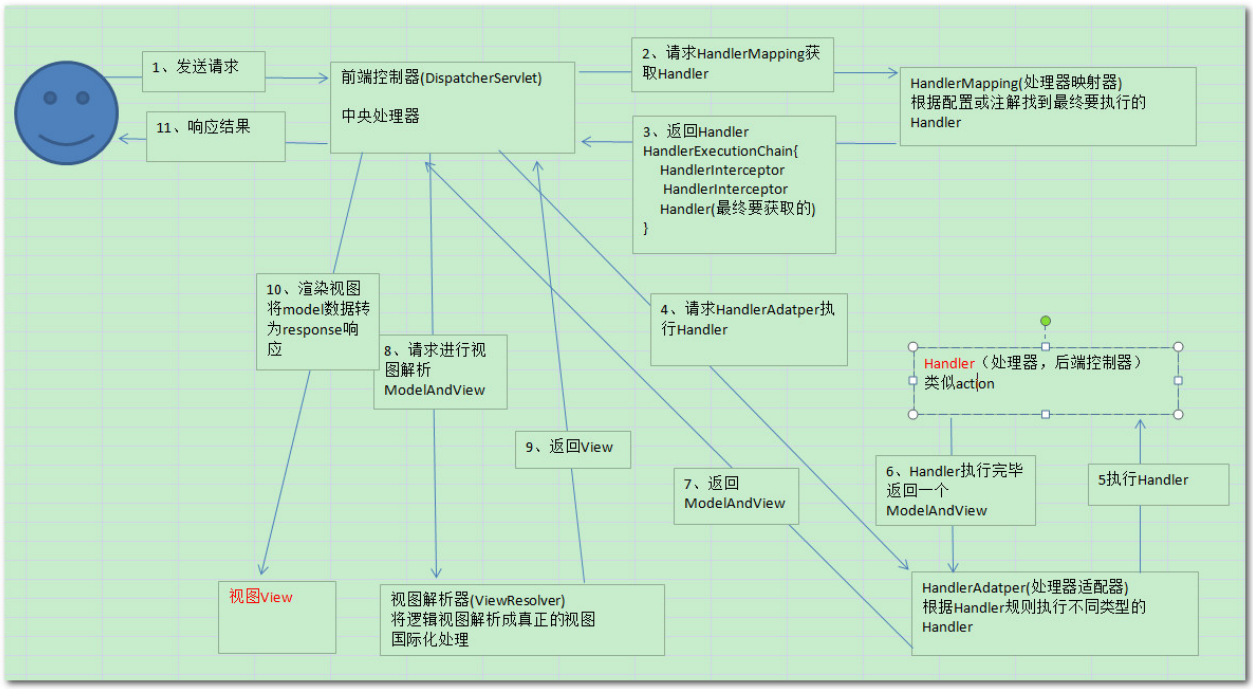
4.spring 中Bean的单例和多例模式的使用条件

> spring生成的对象默认都是单例(singleton)的.可以通过scope改成多例. 对象在整个系统中只有一份，所有的请求都用一个对象来处理，如service和dao层的对象一般是单例的。
>
> 为什么使用单例：因为没有必要每个请求都新建一个对象的时候，因为这样会浪费CPU和内存。
>
> prototype 多例模式：对象在整个系统中可以有多个实例，每个请求用一个新的对象来处理，如action。
>
> 为什么使用多例：防止并发问题；即一个请求改变了对象的状态，此时对象又处理另一个请求，而之前请求对对象的状态改变导致了对象对另一个请求做了错误的处理；

5.SSM(顺丰)

> 5.1SSM各层关系
>
> 5.2 为什么注入的是接口(接口多继承)
>
> 5.3 Spring的优点
>
> > 1.降低了组件之间的耦合性 ，实现了软件各层之间的解耦 2.可以使用容易提供的众多服务，如事务管理，消息服务等 3.容器提供单例模式支持 4.容器提供了AOP技术，利用它很容易实现如权限拦截，运行期监控等功能 5.容器提供了众多的辅助类，能加快应用的开发 6.spring对于主流的应用框架提供了集成支持，如hibernate，JPA，Struts等 7.spring属于低侵入式设计，代码的污染极低 8.独立于各种应用服务器 9.spring的DI机制降低了业务对象替换的复杂性 10.Spring的高度开放性，并不强制应用完全依赖于Spring，开发者可以自由选择spring的部分或全部

Spring MVC 的处理过程



（1）客户端通过url发送请求

（2-3）核心控制器Dispatcher Servlet接收到请求，通过系统或自定义的映射器配置找到对应的handler，并将url映射的控制器controller返回给核心控制器。

（4）通过核心控制器找到系统或默认的适配器

（5-7）由找到的适配器，调用实现对应接口的处理器，并将结果返回给适配器，结果中包含数据模型和视图对象，再由适配器返回给核心控制器

（8-9）核心控制器将获取的数据和视图结合的对象传递给视图解析器，获取解析得到的结果，并由视图解析器响应给核心控制器

（10）核心控制器将结果返回给客户端

7.数据库

7.1 数据库的三范式以及内外连接

1.数据库的三范式

第一范式(1NF):

> 指的是数据库表的中的每一列都是不可分割的基本数据项,同一列中不能有多个值。第一范式要求属性值是不可再分割成的更小的部分。第一范式简而言之就是强调的是列的原子性，即列不能够再分成其他几列。例如有一个列是电话号码一个人可能有一个办公电话一个移动电话。第一范式就需要拆开成两个属性。

第二范式（2NF）：

> 第二范式首先是第一范式，同时还需要包含两个方面的内容，一是表必须要有一个主键；二是没有包含主键中的列必须完全依赖主键，而不能只是依赖于主键的一部分。

> 例如在一个订单中可以订购多种产品，所以单单一个 OrderID 是不足以成为主键的，主键应该是（OrderID，ProductID）。显而易见 Discount（折扣），Quantity（数量）完全依赖（取决）于主键（OderID，ProductID），而 UnitPrice，ProductName 只依赖于 ProductID。所以 OrderDetail 表不符合 2NF。不符合 2NF 的设计容易产生冗余数据。 可以把【OrderDetail】表拆分为【OrderDetail】（OrderID，ProductID，Discount，Quantity）和【Product】（ProductID，UnitPrice，ProductName）来消除原订单表中 UnitPrice，ProductName多次重复的情况。

第三范式（3NF）：

> 首先是第二范式，例外非主键列必须依赖于主键，不能存在传递。也就是说不能存在非主键列A依赖于非主键列B，然后B依赖于主键列
>
> 考虑一个订单表【Order】（OrderID，OrderDate，CustomerID，CustomerName，CustomerAddr，CustomerCity）主键是（OrderID）。 其中 OrderDate，CustomerID，CustomerName，CustomerAddr，CustomerCity 等非主键列都完全依赖于主键（OrderID），所以符合 2NF。不过问题是 CustomerName，CustomerAddr，CustomerCity 直接依赖的是 CustomerID（非主键列），而不是直接依赖于主键，它是通过传递才依赖于主键，所以不符合 3NF。
>
> 通过拆分【Order】为【Order】（OrderID，OrderDate，CustomerID）和【Customer】（CustomerID，CustomerName，CustomerAddr，CustomerCity）从而达到 3NF。
>
> 二范式（2NF）和第三范式（3NF）的概念很容易混淆，区分它们的关键点在于，2NF：非主键列是否完全依赖于主键，还是依赖于主键的一部分；3NF：非主键列是直接依赖于主键，还是直接依赖于非主键列。

2.内外连接

1.内连接

> 内连接也叫自然连接，只有两个表相匹配的行才能在结果集中出现。返回的结果集选取两个表中所匹配的数据，舍弃不匹配的数据

2.外连接

> 内连接保证两个表中的所有行都满足条件，而外连接则不然，外连接不仅仅包含符合连接条件的行，而且还包括左表（左外连接），右表（右外连接），或者两个边表（全外连接）中的所有数据行

> 内连接只显示符合连接条件的记录，外连接除了显示符合连接条件的记录外，还显示表中的记录。

3.最左前缀原则

> 就是最左边优先，就类似于通关类游戏，过了第一关，才能过第二关，过了第一关和第二关，才能过第三关
>
> 建立索引a，b，c下列查询a b， a c，b c谁会走这个索引及原因?
>
> 根据最左前缀原则 只有ab会走这个索引

4.事务

> 事务是数据库中的一个单独的执行单元(unit)

4.1事务的四大特性(ACID)

> 1. 原子性:即事务是一个不可分割的整体,数据修改时要么都操作一遍要么都不操作
> 2. 一致性:一个事务执行前后数据库的数据必须保持一致性状态
> 3. 隔离性:当两个或者以上的事务并发执行时,为了保证数据的安全性,将一个事务的内部的操作与事务操作隔离起来不被其他事务看到

> 4. 持久性:更改是永远存在的

4.2事务的隔离级别

> 读未提交：事务中的修改，即使没有提交，其他事务也可以看得到，脏读。如果一个事务已经开始写数据，则另外一个事务则不允许同时进行写操作，但允许其他事务读此行数据。该隔离级别可以通过“排他写锁”实现。一个在写事务另一个虽然不能写但是能读到还没有提交的数据
>
> 读已提交：可以避免脏读但是可能出现不可重复读。允许写事务，读取数据的事务允许其他事务继续访问该行数据，但是未提交的写事务将会禁止其他事务访问该行。事务T1读取数据，T2紧接着更新数据并提交数据，事务T1再次读取数据的时候，和第一次读的不一样。即虚读
>
> 可重复读：禁止写事务，读事务会禁止所有的写事务，但是允许读事务，避免了不可重复读和脏读，但是会出现幻读，即第二次查询数据时会包含第一次查询中未出现的数据
>
> 序列化：禁止任何事务，一个一个进行；提供严格的事务隔离。它要求事务序列化执行，事务只能一个接着一个地执行，但不能并发执行。如果仅仅通过“行级锁”是无法实现事务序列化的，必须通过其他机制保证新插入的数据不会被刚执行查询操作的事务访问到。

5.引擎

MyISAM和innoDB

> count 运算上的区别 : 因为MyISAM缓存有表meta-data (行数等) , 因此在做COUNT(*)时对于一个结构很好的查询是不需要消耗多少资源的。对于innodb是没有这种缓存
>
> MyISAM强调的是性能，每次查询具有原子性，其执行速度比innodb类型更快，但是不提供事务支持。innodb提供事务支持事务，外部建等高级功能
>
> MyISAM不支持，而innodb支持
>
> 总的来说MyISAM更适合读密集的表，而Innodb更适合写密集的表，在数据库主从分离的情况下，经常选择MyISAM做主库存储引擎。

6.索引

6.1优缺点：

> 优点： 可以快速检索，减少I/O次数，加快检索速度；根据索引分组和排序，可以加快分组和排序
>
> 缺点： 索引本省也是表会占用内存，索引表占用的空间是数据表的1.5倍；索引表的创建和维护需要时间成本，这个成本随着数据量的增大而增大。

6.2索引的底层实现原理：

哈希索引：

> 只有memory（内存）存储引擎支持哈希索引，哈希索引用索引列的值计算该值的hashCode，然后在hashCode相应的位置存执该值所在行数据的物理位置，因为使用散列算法，因此访问速度非常快，但是一个值只能对应一个hashCode，而且是散列的分布方式，因此哈希索引不支持范围查找和排序的功能。

Btree索引：

> B树是一个平衡多叉树，设树的度为2d，高度为h，那么B树需要满足每个叶子节点的高度都一样等于h，每个非叶子节点由n-1个key和n个point组成，d< = n<=2d 。所有叶子节点指针均为空，非叶子结点的key都是[key,data]二元组，其中key表示作为索引的键，data为键值所在行的数据。

B+Tree索引

> B+Tree是BTree的一个变种，设d为树的度数，h为树的高度，B+Tree和BTree的不同主要在于：
>
> B+Tree中的非叶子结点不存储数据，只存储键值； B+Tree的叶子结点没有指针，所有键值都会出现在叶子结点上，且key存储的键值对应data数据的物理地址；B+Tree的每个非叶子节点由n个键值key和n个指针point组成；
>
> 优点：查询速度更加稳定，磁盘的读写代价更低

聚簇索引与非聚簇索引

> 聚簇索引的解释是:聚簇索引的顺序就是数据的物理存储顺序
>
> 非聚簇索引的解释是:索引顺序与数据物理排列顺序无关

MyISAM——非聚簇索引

> MyISAM存储引擎采用的是非聚簇索引，非聚簇索引的主索引和辅助索引几乎是一样的，只是主索引不允许重复，不允许空值，他们的叶子结点的key都存储指向键值对应的数据的物理地址。 非聚簇索引的数据表和索引表是分开存储的。

innoDB——聚簇索引

> 聚簇索引的主索引的叶子结点存储的是键值对应的数据本身，辅助索引的叶子结点存储的是键值对应的数据的主键键值。因此主键的值长度越小越好，类型越简单越好。 聚簇索引的数据和主键索引存储在一起。

6.3 联合索引(顺丰)

> 利用最左前缀原则

7.数据库锁

> 锁是计算机协调多个进程或者纯线程并发访问某一资源的机制

7.1Mysql的锁种类

> Mysql的锁机制比较简单，不同的搜索引擎支持不同的锁机制
>
> 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率高，并发度最低
>
> 行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突概率最低，并发度也最高
>
> 页面锁：开销和加锁速度位于表锁和行锁之间，会出现死锁，锁定粒度也位于表锁和行锁之间，并发度一般

7.2Mysql表级锁的锁模式（MyISAM）

> Mysql表级锁有两种模式：表共享锁（Table Read Lock）和表独占锁（Table Write Lock）

8.having 和group by

8.计算机网络相关问题

8.1：HTTP的状态码

> 1XX信息提示 100继续。101切换协议
>
> 2XX表示成功 200：确定。客户端请求已成功
>
> 3XX表示重定向 300多种选择，也即是针对这个请求服务器可以做出多种操作
>
> 4XX表示请求出错 400表示服务器不理解语法 404表示服务器找不到请求页面
>
> 5XX表示服务器处理请求出错 500表示服务器内部出错

8.2TCP的三次握手和四次挥手

三次握手是为了建立可靠的连接，让双方都确认自己和对方的发送和接收是正常的

客 ---—带SYN标志的数据包—---------———-》服 服务器确认客服端发送正常

《---------带SYN/ACK的数据包------- 客确认自己发送接收正常对方发送正常接受正常 服确认自己接收正常，对方接收正常

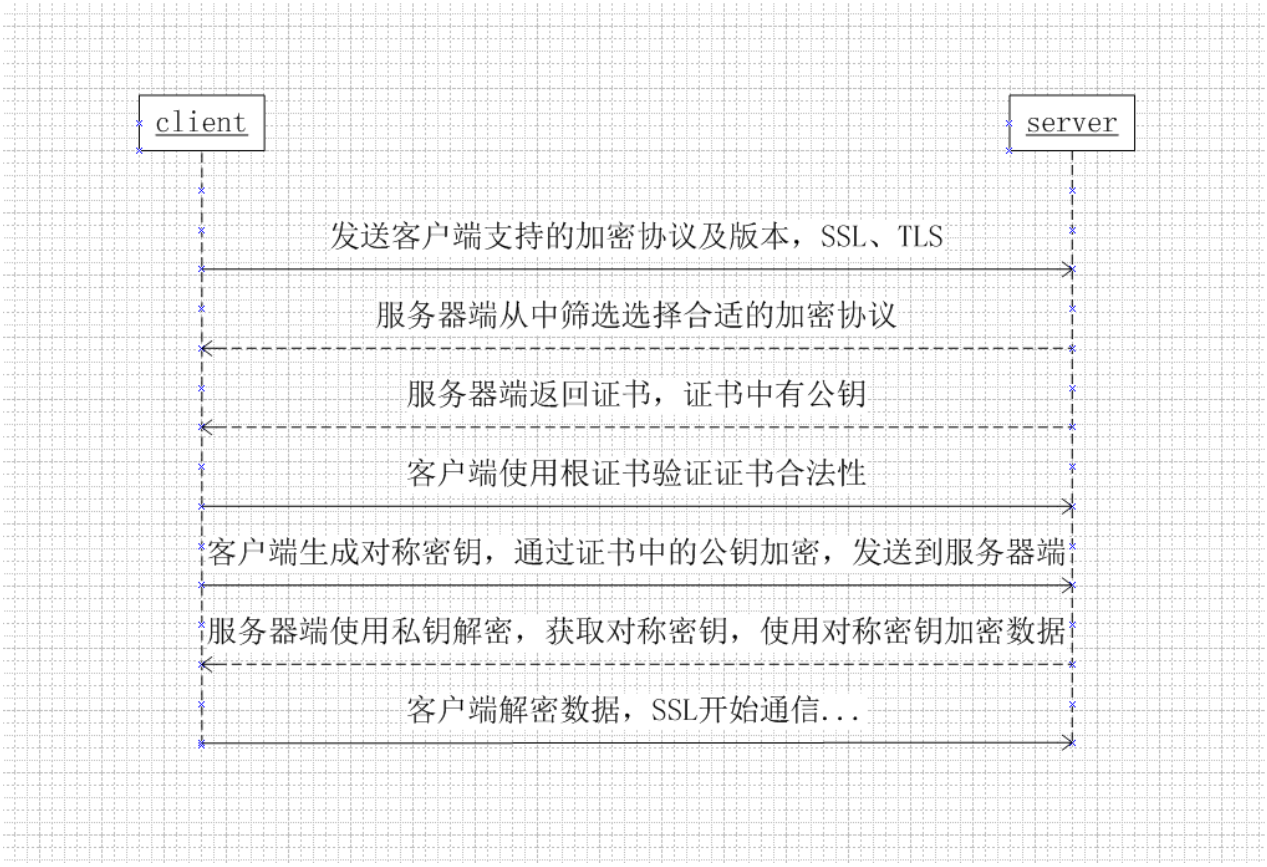
—-------带ACK的数据包-—------》 客，服务确认自己发送接收正常对方发送正常接受正常

四次挥手：任何一方都可以在数据传送结束后发出连接释放的通知,待对方确认后进入半关闭状态。当另一方也没有数据在发送的时候，则发出连接释放通知，对方确认之后就可以完全关闭TCP连接了

8.3四次挥手等待2MSL的原因：

> 先看正常情况。当客户端发出最后一个ACK报文后，网络协议上约定该报文再不济也会经过一个MSL到达服务器，随后服务器正常关闭。
>
> 客户端继续等一个MSL（若有问题，则问题都会在这个MSL水落石出），这里考虑正常情况，在这个MSL内客户端当然不会收到任何来自服务器的信息，所以客户端也正常关闭。
>
> 那么，异常情况，客户端发出最后一个ACK后，考虑在路上的任一时刻丢包。可能是刚发出去就丢了，也可能快到服务器的最后一跳丢了。客户端等完第一个MSL以后，此刻也不知道丢包没丢包，但是会假设：一个MSL已经足够长，如果路上丢了，服务器必然已经知道了丢包的事实，所以服务器会超时重传 从服务器发往客户端的最后一个报文 即书上从下往上数第二个报文，FIN=1，ACK=1,seq=w,ack=u+1。该报文再不济也会在下一个MSL内被客户端收到。

8.4 Https的建立过程



首先客户端享服务器发送请求,其中包含了客户端支持的加密协议版本以及ssl、tsl信息

服务器选择合适的加密协议，并且发送一个Ca证书给客户端，证书里面包含一个公钥

客户端验证证书的合法性，通过公钥加密一个共享密钥，将加密的共享密钥发送给服务端

服务端接受到加密的密钥之后通过私钥解密获得共享密钥，之后就可以利用这个共享密钥加密数据发送给客户端

客户端接收到加密的数据就可以利用共享密钥来解密，之后就可以开始ssl通信了

8.5 TCP粘包问题

udp不会产生粘包,因为udp是基于报文发送的,tcp是基于字节流发送的。

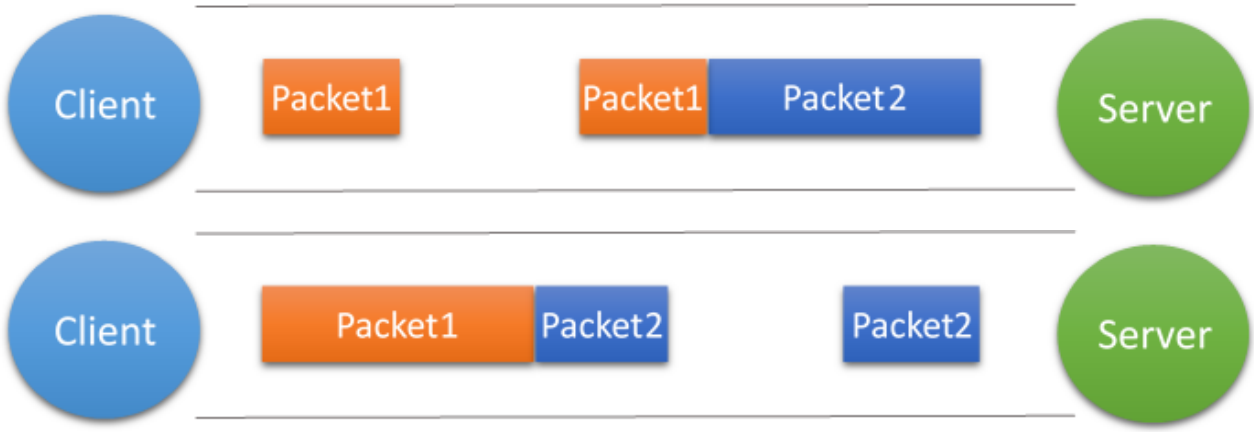
8.5.1 粘包、拆包的表现形势

现在假设客户端向服务端连续发送了两个数据包，用packet1和packet2来表示，那么服务端收到的数据可以分为三种

1、第一种情况，接收端正常收到两个数据包，即没有发生拆包和粘包的现象，此种情况不在本文的讨论范围内。

2、第二种情况，接收端只收到一个数据包，由于TCP是不会出现丢包的，所以这一个数据包中包含了发送端发送的两个数据包的信息，这种现象即为粘包。这种情况由于接收端不知道这两个数据包的界限，所以对于接收端来说很难处理。

3、第三种情况，这种情况有两种表现形式，如下图。接收端收到了两个数据包，但是这两个数据包要么是不完整的，要么就是多出来一块，这种情况即发生了拆包和粘包。这两种情况如果不加特殊处理，对于接收端同样是不好处理的。



8.5.2 粘包、拆包发生原因

粘包的发生原因

1. 要发送的数据小于TCP发送缓冲区的大小，TCP将多次写入缓冲区的数据一次发送出去，将会发生粘包。
2. 接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生粘包。

拆包发生的原因

1. 要发送的数据大于TCP发送缓冲区剩余空间大小，将会发生拆包。
2. 待发送数据大于MSS（最大报文长度），TCP在传输前将进行拆包。

8.5.3 粘包的解决办法

解决问题的关键在于如何给每个数据包添加边界信息，常用的方法有如下几个：

1. 发送端给每个数据包添加包首部，首部中应该至少包含数据包的长度，这样接收端在接收到数据后，通过读取包首部的长度字段，便知道每一个数据包的实际长度了。
2. 发送端将每个数据包封装为固定长度（不够的可以通过补0填充），这样接收端每次从接收缓冲区中读取固定长度的数据就自然而然的把每个数据包拆分开来。
3. 可以在数据包之间设置边界，如添加特殊符号，这样，接收端通过这个边界就可以将不同的数据包拆分开。

9.乐观锁和悲观锁(顺丰)

悲观锁

> 悲观锁就是每次都假设最坏的情况,每次去拿数据的时候都认为别人会修改,所以每次在拿数据的时候都会上锁,这样别人想拿数据就会阻塞知道他拿到锁(共享资源每次只给一个线程使用,其他线程阻塞,用完之后再把资源转让给其他线程)。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁，读锁，写锁等，都在操作之前先上锁。Java中的synchronized和reentrantLock等独占锁就是悲观锁的思想实现

乐观锁

> 乐观锁就是假设最好的情况,每次拿数据的时候都认为别人不会修改,所以不会身上锁,但是在更新的时候会判断一下再次期间别人有没有去更新这个数据,可以使用版本号机制和CAS算法来实现,乐观锁适用于多读的应用类型,这样可以提高吞吐量（即冲突很少发生的情况，这样可以省去所得开销，加大系统的吞吐量），数据库中的write_condition机制,其实就是提供乐观锁。在Java中Java.util.concurrent.atomic包下面的原子变量类就是使用了乐观锁的一种实现方式CAS实现的。

乐观锁的两种实现机制

1.版本号机制

> 一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。
>
> **举一个简单的例子：** 假设数据库中帐户信息表中有一个 version 字段，当前值为 1 ； 而当前帐户余额字段（ balance ）为 $100 。
>
> 1. 操作员 A 此时将其读出（ version=1 ），并从其帐户余额中扣除 $50 （ $100-$50 ）。
> 2. 在操作员 A 操作的过程中，操作员B 也读入此用户信息（ version=1 ），并从其帐户余额中扣除 $20 （ $100-$20 ）。
> 3. 操作员 A 完成了修改工作，将数据版本号加一（ version=2 ），连同帐户扣除后余额（ balance=$50 ），提交至数据库更新，此时由于提交数据版本大于数据库记录当前版本，数据被更新，数据库记录 version 更新为 2 。
> 4. 操作员 B 完成了操作，也将版本号加一（ version=2 ）试图向数据库提交数据（ balance=$80 ），但此时比对数据库记录版本时发现，操作员 B 提交的数据版本号为 2 ，数据库记录当前版本也为 2 ，不满足 “ 提交版本必须大于记录当前版本才能执行更新 “ 的乐观锁策略，因此，操作员 B 的提交被驳回。
>
> 这样，就避免了操作员 B 用基于 version=1 的旧数据修改的结果覆盖操作员A 的操作结果的可能。

2.CAS算法

> compare and swap（比较与交换），是一种有名的无锁算法。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。CAS算法涉及到三个操作数
>
> 需要读写的内存值 V 进行比较的值 A 拟写入的新值 B 当且仅当 V 的值等于 A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试。

9.1 在static方法上加锁和非static方法上加锁的区别

> 1.对象锁钥匙只能有一把才能互斥，才能保证共享变量的唯一性
>
> 2.在静态方法上的锁，和 实例方法上的锁，默认不是同样的，如果同步需要制定两把锁一样。

> 3.关于同一个类的方法上的锁，来自于调用该方法的对象，如果调用该方法的对象是相同的，那么锁必然相同，否则就不相同。比如 new A().x() 和 new A().x(),对象不同，锁不同，如果A的单利的，就能互斥。
>
> 4.静态方法加锁，能和所有其他静态方法加锁的 进行互斥
>
> 5.静态方法加锁，和xx.class 锁效果一样，直接属于类的
>
> 6.(自己补的)照上边所说，如果同一个对象上的2个非static的方法上加锁，这2个方法虽然不是一个方法，但如果都加锁的话也会互斥，即同一个对象不同非static的方法加锁的话一个方法已经拿到锁了那另外一个线程用同一个对象调用另外一个线程时也会处于等待---总结就是如果锁非static的方法的话就如同锁对象，而且同一个对象只有一把锁。那锁不同的属性呢?

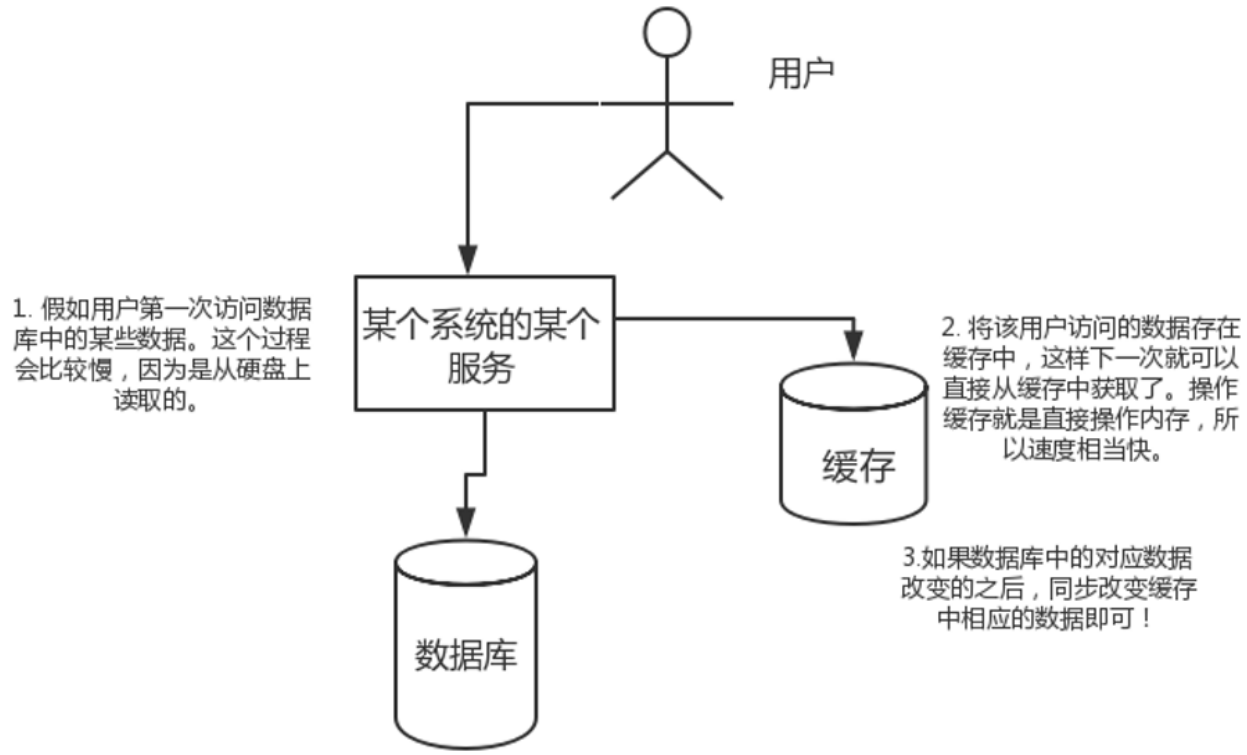
9.2 对类加锁和对对象加锁的区别

>

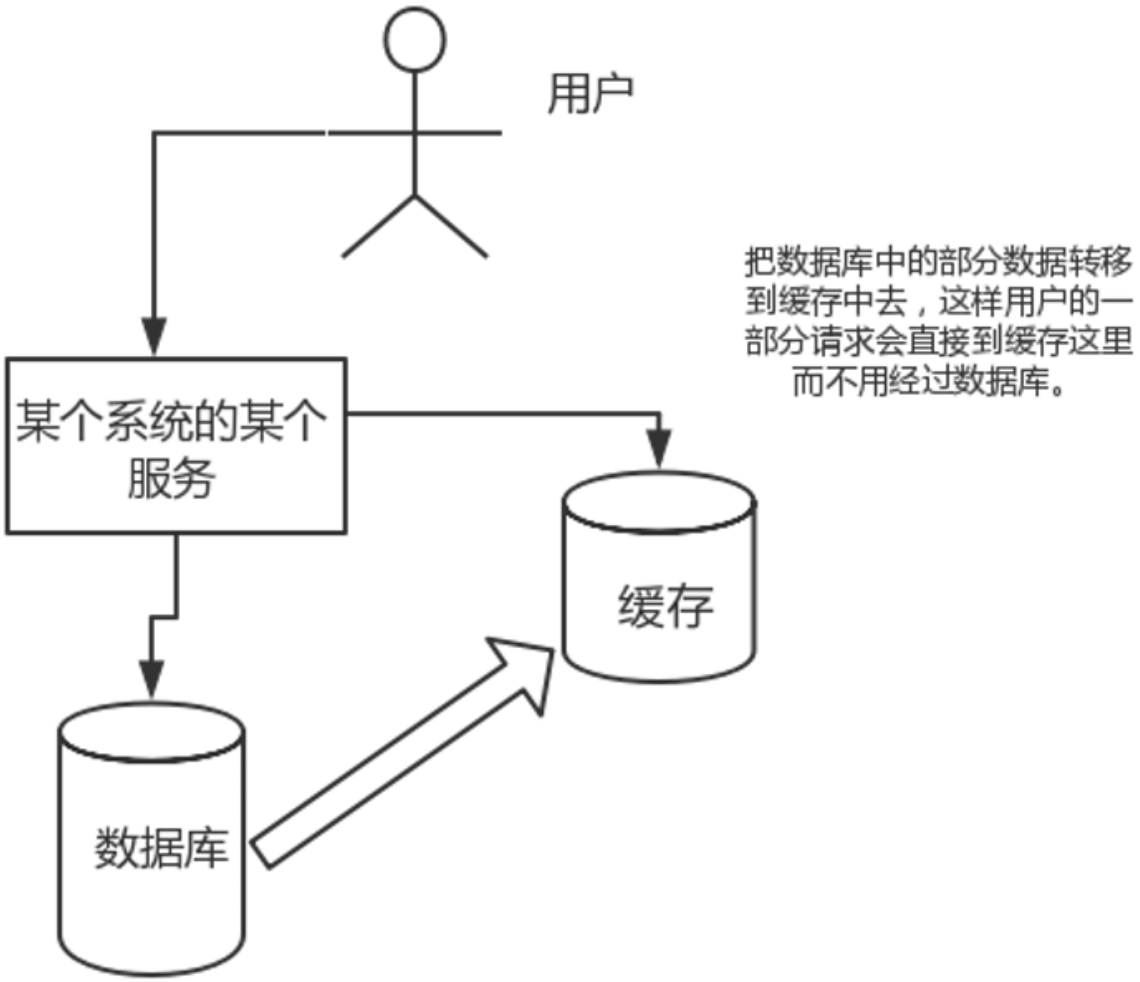
10.redis

10.1解决高并发和高性能的问题

高性能：直接处理缓存也就是处理内存很快



高并发：直接操作缓存能够承受的请求是远远大于直接访问数据库的，所以我们可以考虑把数据库中的部分数据转移到缓存中 去，这样用户的一部分请求会直接到缓存这里而不用经过数据库



10.2redis的常见数据结构即使用场景分析

1. String 常用命令: set,get,decr,incr,mget 等。 String数据结构是简单的key-value类型，value其实不仅可以是String，也可以是数字。 常规key-value缓存应用； 常规计数：微博数，粉丝数等。
2. Hash 常用命令： hget,hset,hgetall 等。 Hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象，后续操作的时候，你可以直接仅 仅修改这个对象中的某个字段的值。 比如我们可以Hash数据结构来存储用户信息，商品信息等等。比如下面我就用 hash 类型存放了我本人的一些信息：key=JavaUser293847 value={ “id”: 1, “name”: “SnailClimb”, “age”: 22, “location”: “Wuhan, Hubei” }
3. List 常用命令: lpush,rpush,lpop,rpop,lrange等list 就是链表，Redis list 的应用场景非常多，也是Redis 重要的数据结构之一，比如微博的关注列表，粉丝列表， 消息列表等功能都可以用Redis的 list 结构来实现。 Redis list 的实现为一个双向链表，即可以支持反向查找和遍历，更方便操作，不过带来了部分额外的内存开销。 另外可以通过 lrange 命令，就是从某个元素开始读取多少个元素，可以基于 list 实现分页查询，这个很棒的一个功 能，基于 redis 实现简单的高性能分页，可以做类似微博那种下拉不断分页的东西（一页一页的往下走），性能高。
4. Set 常用命令： sadd,spop,smembers,sunion 等 set 对外提供的功能与list类似是一个列表的功能，特殊之处在于 set 是可以自动排重的。 当你需要存储一个列表数据，又不希望出现重复数据时，set是一个很好的选择，并且set提供了判断某个成员是否在 一个set集合内的重要接口，这个也是list所不能提供的。可以基于 set 轻易实现交集、并集、差集的操作。 比如：在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis可以非常 方便的实现如共同关注、共同粉丝、共同喜好等功能。这个过程也就是求交集的过程，具体命令如下： sinterstore key1 key2 key3 将交集存在key1 内

> 5. Sorted Set 常用命令： zadd,zrange,zrem,zcard等 和set相比，sorted set增加了一个权重参数score，使得集合中的元素能够按score进行有序排列。 举例： 在直播系统中，实时排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息（可以理解为按消息维 度的消息排行榜）等信息，适合使用 Redis 中的SortedSet 结构进行存储。

2.Hash： 适合用于存储对象，后续操作的时候，你可以直接仅仅修改这个对象中的某个字段的值。可以用Hash来存储用户的信息，商品的信息等等。hget ， hset ， hgetall

3.List：list就是链表，是最重要的数据结构，比如微博的关注列表，粉丝列表，消息列表都是用redis的list结构来实现的。

lrange可以用来实现分页查看的功能。lpush ， lpop ， rpush，rpop

4.Set：set对外提供的功能和list类似，但是set可以自动排重。可以轻易实现交集，并集，差集的操作。比如微博的共同关注，共同粉丝和共同喜好。

5.Sorted Set： 和set相比，sorted set 增加了一个权重参数score ， 使得集合中的元素能够按score进行有序排列。举例，在直播系统中，实施排行信息包含直播间在线用户列表，各种礼物排行榜，弹幕消息等信息，适用于Redis中的SortedSet结构进行存储。

10.3 redis这是过期时间

> 有些数据是有时间限制的例如一些登陆信息，尤其是短信验证码都是有时间限制的。
>
> 定期删除+惰性删除
>
> 定期删除要点：默认每隔1000ms就随机抽取一些设置了过期时间的key。
>
> 惰性删除：定期删除会导致很多过期的key到了时间并没有被删除掉。假如过期的key靠定期删除没有删除掉，还停留在内存中，除非你的系统去查一下那个key，才会被redis删除

10.4reddis的持久化机制

> 1.RDB
>
> 也就是快照持久化,通过创建快照来获得存储在内存里面的数据在某个时间节点上的副本。redis创建快照后可以对快照进行备份，可以将快照复制到其他服务器从而创建出具有相同数据的服务器副本（redis主从结构，主要用来提高redis的性能），还可以将快照留在原地以便重启服务器的时候使用
>
> 2.AOF只追加文件
>
> 与快照相比AOF的实时性更好，开启AOF持久化后每执行一条会更改Redis中的数据的命令，Redis就会将该命令写入硬盘中的AOF文件

10.5 缓存雪崩和缓存穿透

> 缓存穿透：一般是黑客故意去请求缓存中不存在的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量 请求而崩掉。 解决办法： 有很多种方法可以有效地解决缓存穿透问题，常见的则是采用布隆过滤器，将所有可能存在的数据哈 希到一个足够大的bitmap中，一个一定不存在的数据会被 这个bitmap拦截掉，从而避免了对底层存储系统的查询压 力。另外也有一个更为简单粗暴的方法（我们采用的就是这种），如果一个查询返回的数据为空（不管是数 据不存 在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，长不超过五分钟。
>
> 缓存雪崩：缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩 掉。

> 生雪崩的原因之一，比如在写本文的时候，马上就要到双十二零点，很快就会迎来一波抢购，这波商品时间比较集中的放入了缓存，假设缓存一个小时。那么到了凌晨一点钟的时候，这批商品的缓存就都过期了。而对这批商品的访问查询，都落到了数据库上，对于数据库而言，就会产生周期性的压力波峰。

10.6 redis与数据库数据更新的问题流程,以及数据一致性(顺丰)

10.6.1 写完数据库后是否需要马上更新缓存还是直接删除缓存?

> 1. 如果写数据库的值与更新到缓存值是一样的，不需要经过任何的计算，可以马上更新缓存，但是如果对于那种写数据频繁而读数据少的场景并不合适这种解决方案，因为也许还没有查询就被删除或修改了，这样会浪费时间和资源
> 2. 如果写数据库的值与更新缓存的值不一致，写入缓存中的数据需要经过几个表的关联计算后得到的结果插入缓存中，那就没有必要马上更新缓存，只有删除缓存即可，等到查询的时候在去把计算后得到的结果插入到缓存中即可。
>
> 所以一般的策略是当更新数据时，先删除缓存数据，然后更新数据库，而不是更新缓存，等要查询的时候才把最新的数据更新到缓存
>
> 更新的时候先更新数据库在跟新缓存，读的时候先读缓存，要是缓存里面没有就再读数据库，同时将数据放入缓存并返回响应
>
> 这样会引起数据一致性问题，如果先更新了数据库，删除缓存的时候失败了怎么办？那么数据库中是新数据，缓存中是老数据，数据出现不一致了。
>
> 所以改进为：
>
> 先删除缓存，后更新数据库。因为即使后面更新数据库失败了，缓存是空的，读的时候会从数据库中重新拉，虽然都是旧数据，但数据是一致的。
>
> 更新的时候先删除缓存再跟新数据库，读的时候先读缓存，要是缓存里面没有就再读数据库，同时将数据放入缓存并返回响应（我的项目只缓存了签到数据，每天早上十点将签到数据缓存到redis，便于查询）

11.线程安全与线程不安全

11.1概念

> 线程安全：线程安全就是多线程访问的时候采用了加锁机制，当一个线程访问数据时，进行了保护，其他线程不能进行访问直到该线程访问结束。不会出现脏数据
>
> 线程不安全： 就是在数据访问时不提供保护，有可能出现多个线程先后更改数据造成得到的数据是脏数据

11.2常见的线程安全和线程不安全的类

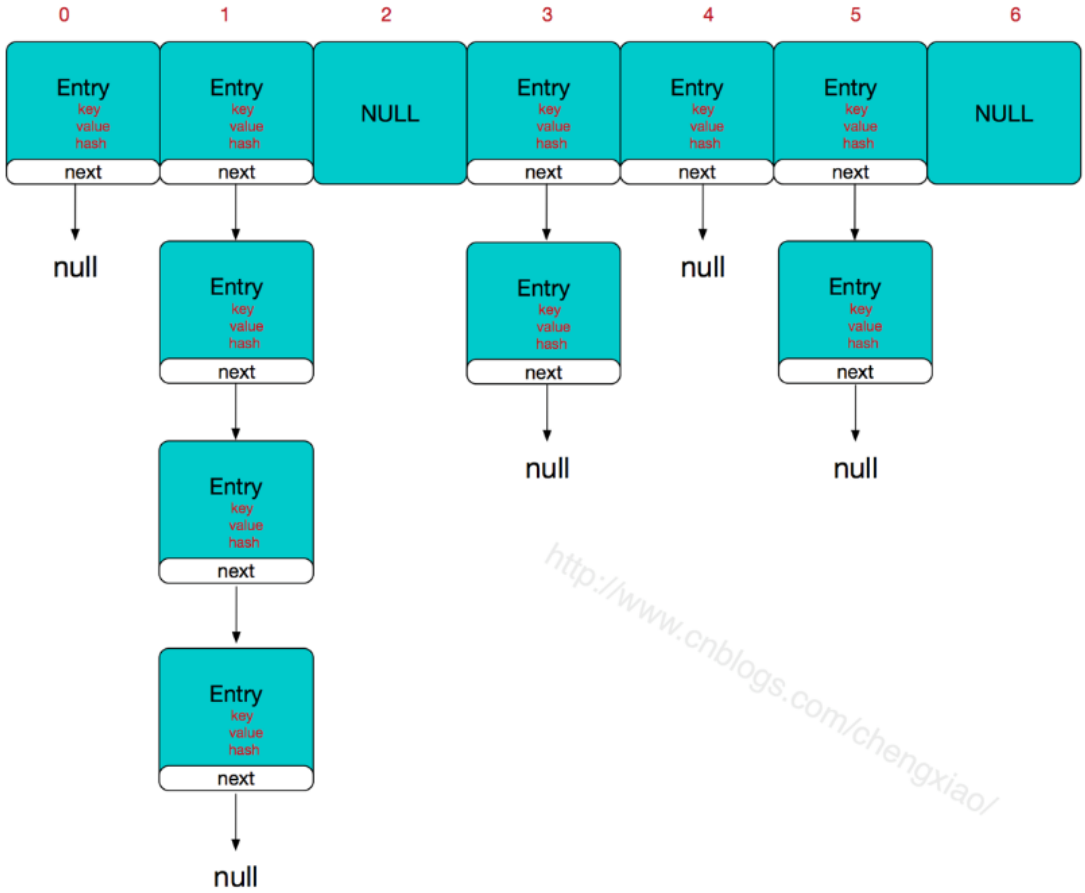
> ArrayList是非线程安全的，Vector是线程安全的；（没有一个ArrayList是同步的，大多数vctor都是直接或者间接同步的）
>
> HashMap是非线程安全的，HashTable是线程安全的；
>
> StringBuilder是非线程安全的，StringBuffer是线程安全的。

11.3线程安全的实现

> 线程安全是通过线程同步控制来实现的，也就是synchronized关键字来实现

11.4HashMap的get，put以及扩容原理

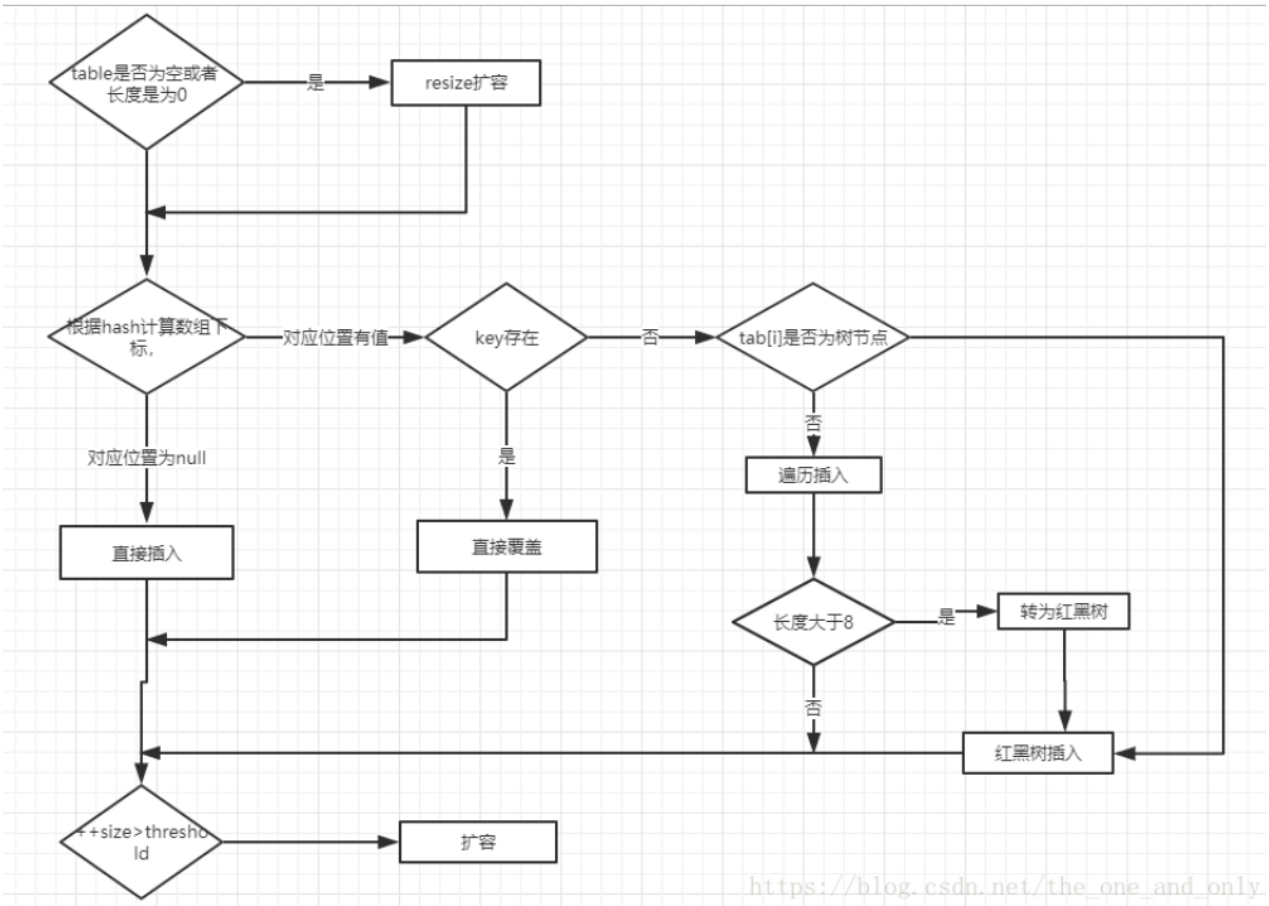
> HashMap的实现原理：使用了数组，链表和红黑树来实现的



简单来说，HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的，如果定位到的数组位置不含链表（当前entry的next指向null），那么对于查找，添加等操作很快，仅需一次寻址即可；如果定位到的数组包含链表，对于添加操作，其时间复杂度为O(n)，首先遍历链表，存在即覆盖，否则新增；对于查找操作来讲，仍需遍历链表，然后通过key对象的equals方法逐一比对查找。所以，性能考虑，HashMap中的链表出现越少，性能才会越好。

11.4.1 put

> 1.调用hash函数得到key的HashCode值 2.通过HashCode值与数组长度-1逻辑与运算得到一个index值 3.遍历索引位置对应的链表，如果Entry对象的hash值与hash函数得到的hash值相等，并且该Entry对象的key值与put方法传过来的key值相等则，将该Entry对象的value值赋给一个变量，将该Entry对象的value值重新设置为put方法传过来的value值。将旧的value返回。
>
> 4.添加Entry对象到相应的索引位置



11.4.2get

> 先前HashMap通过hash code来存放数据，那么get方法一样要通过hash code来获取数据。可以看到如果当前table没有数据的话直接返回null反之通过传进来的hash值找到对应节点（Node）first，如果first的hash值以及Key跟传入的参数匹配就返回对应的value反之判断是否是红黑树，如果是红黑树则从根节点开始进行匹配如果有对应的数据则结果否则返回Null，如果是链表的话就会循环查询链表，如果当前的节点不匹配的话就会从当前节点获取下一个节点来进行循环匹配，如果有对应的数据则返回结果否则返回Null

11.4.3扩容机制

> 扩容是成倍增长的利用resize()方法会在HashMap的键值对达到“阈值”后进行数组扩容，而扩容时会调用resize()方法。元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置

11.5HashMap和TreeMap之间的关系与区别(顺丰)

> TreeMap怎么实现的
>
> 怎么排序的

12.GC垃圾回收机制

12.1概念

> 主要作用是回收程序中不再使用的内存。主要完成三项任务：分配内存，确保被引用的对象的内存不被错误的回收以及回收不再被引用的对象的内存空间

12.2GC的四大算法

（1）引用计数算法

> 简单效率低。在堆中每个对象都有一个引用计数器；当对象被引用时，引用计算器加1，当引用被置为空或者离开作用域时，引用计数减1，但是不能解决相互引用的问题，所以jvm没有采用这个算法。

（2）追踪回收算法

> 追踪回收算法利用jvm维护的对象引用图，从根节点开始遍历对象引用图，同时标记遍历到的对象。当遍历结束后，未被标记的对象就是目前已不被使用的对象，可以被回收了。

（3）压缩回收算法

> 把堆中活动的对象移动到堆中另一端，这样就会在堆中另外一端留出很大的一块空闲区域，相当于对堆中的碎片进行了处理。虽然可以大大简化消除堆碎片的工作，但每次处理都会带来性能的损失

（4）复制回收算法

> 把堆分成大小相同的区域，在任何时刻，只有其中的一个区域被使用，直到这个区域的被消耗完成为止，此时垃圾回收器会中断程序执行，通过程序的遍历方式把所有活动的对象复制到另一个区域中，在复制的过程中他们是紧紧挨着布置的，从而可以消除内存碎片。当复制过程结束后程序会接着运行直到这块区域被使用完，然后在采用上面的方法继续进行垃圾回收

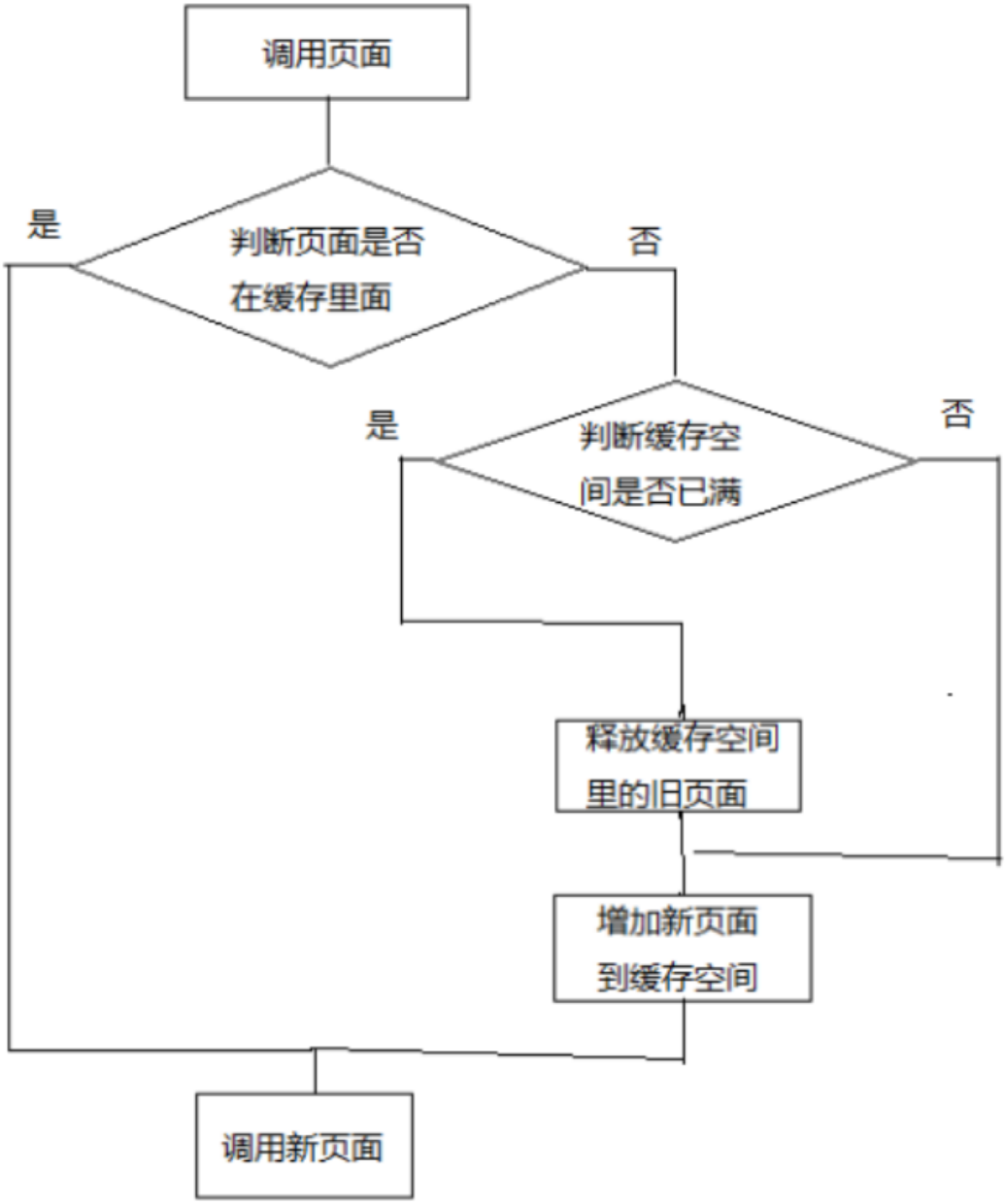
（5）迭代回收算法

> 复制回收算法每次执行时，所有处于活动状态的都要被复制，执行这样的效率很低。迭代算法：把堆分成两个或者多个子堆，每个子堆被视为一代，算法优先收集“年幼”的对象，如果一个对象经过多次收集依然“存活”，那么就把这个对象转移到高一级的堆里，减少对其的扫描次数。

12.3 G1 和 CMS (携程)

>

13缓存过期策略



14.Java设计模式

14.1 动态代理(顺丰)

动态代理怎么实现的

14.2 Java 反射机制怎么实现的(顺丰)

只要有类名就可以获取类的全部信息

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

> 要想解剖一个类,必须先要获取到该类的字节码文件对象。而解剖使用的就是Class类中的方法.所以先要获取到每一个字节码文件对应的Class类型的对象 。反射就是把java类中的各种成分映射成一个个的Java对象。
>
> java动态代理实现代理步骤：
>
> a定义被代理类：接口及接口实现类
>
> b定义代理类，代理类需要实现InvocationHandler接口的类并重写invoke方法
>
> c生成被代理的类的实例：调用 Proxy.newProxyInstance(被代理的类.getClass().getClassLoader(), 被代理类.getClass().getInterfaces(), InvocationHandler的实现类);
>
> 注意：newProxyInstance返回的是接口类型，所以java动态代理要求被代理类实现接口。
>
> d被代理的类的实例调用需要执行的方法

14.2.1反射的作用

> ①、在运行时判断任意一个对象所属的类 ②、在运行时构造任意一个类的对象 ③、在运行时判断任意一个类所具有的成员变量和方法（通过反射设置可以调用 private） ④、在运行时调用任意一个对象的方法

14.2.2 反射的实现

> 类名.class
>
> 类名.getClass()
>
> Class.forName(“全类名即包名+类名")

14.3cglib的动态代理为什么可以不基于接口实现

> 基于继承

15.多线程

15.1 线程池(顺丰)

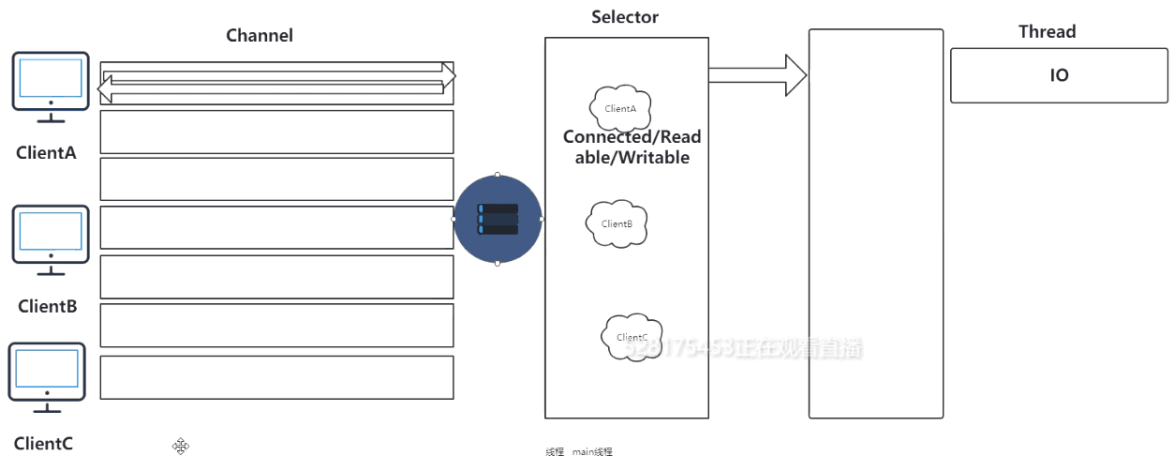
```
/**
@param corePoolSize 核心池的大小,在创建了线程池后，默认情况下，线程池中并没有任何线程，而  是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或prestartCoreThread()方 法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就    会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中;
@param maximumPoolSize  线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程
@param keepAliveTime 表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0;
@param unit 参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性
@param workQueue 一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响 主要ArrayBlockingQueue;LinkedBlockingQueue;SynchronousQueue;
@param threadFactory 线程工厂，主要用来创建线程
@param handler 表示当拒绝处理任务时的策略，有以下四种取值:
```

```
ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。
ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。
ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）
ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务
  */
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler) {
 if (corePoolSize < 0 ||
     maximumPoolSize <= 0 ||
     maximumPoolSize < corePoolSize ||
     keepAliveTime < 0)
     throw new IllegalArgumentException();
 if (workQueue == null || threadFactory == null || handler == null)
     throw new NullPointerException();
 this.acc = System.getSecurityManager() == null ?
         null :
         AccessController.getContext();
 this.corePoolSize = corePoolSize;
 this.maximumPoolSize = maximumPoolSize;
 this.workQueue = workQueue;
 this.keepAliveTime = unit.toNanos(keepAliveTime);
 this.threadFactory = threadFactory;
 this.handler = handler;
}
```

```
ThreadPoolExecutor extends AbstractExecutorService //继承抽象类
AbstractExecutorService implements ExecutorService //实现接口
ExecutorService extends Executor  //继承Executor接口
```

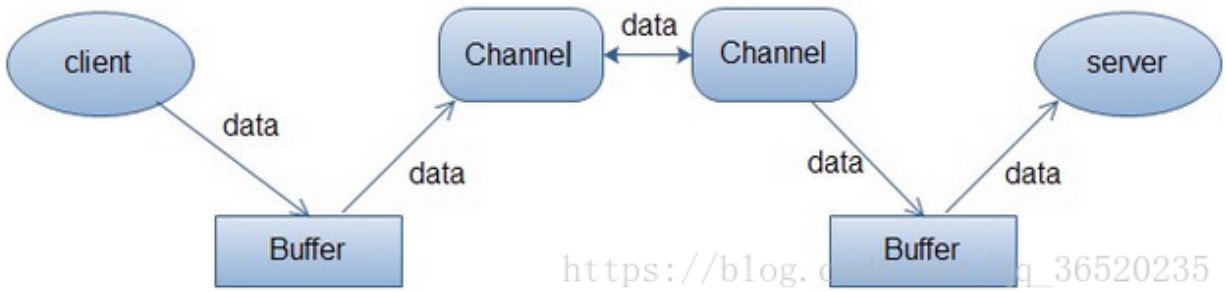
1. Executor是一个顶层接口，在它里面只声明了一个方法execute(Runnable)，返回值为void，参数为Runnable类型，从字面意思可以理解，就是用来执行传进去的任务的；
2. ExecutorService接口继承了Executor接口，并声明了一些方法：submit、invokeAll、invokeAny以及shutDown等；
3. 抽象类AbstractExecutorService实现了ExecutorService接口，基本实现了ExecutorService中声明的所有方法；
4. 然后ThreadPoolExecutor继承了类AbstractExecutorService。

16.Java Nio



主要有channel buffer 和selector实现的

Buffer顾名思义：缓冲区，实际上是一个容器，一个连续数组。Channel提供从文件、网络读取数据的渠道，但是读写的数据都必须经过Buffer。如下图：



数据--读-->channel--缓存-----> buffer----写---> channel-----> 文件

（1）创建 ServerSocketChannel，配置它为非阻塞模式；

（2）绑定监听，配置 TCP 参数，例如 backlog 大小；

（3）创建一个独立的 I/O 线程，用于轮询多路复用器 Selector；

（4）创建 Selector，将之前创建的 ServerSocketChannel 注册到 Selector 上，监听 SelectionKey.ACCEPT；

（5）启动 I/O 线程，在循环体中执行 Selector.select()方法，轮询就绪的 Channel；

（6）当轮询到了处于就绪状态的 Channel 时，需要对其进行判断，如果是 OP_ACCEPT 状态，说明是新的客户端接入，则调用 ServerSocketChannel.accept()方法接受新的客户端；

（7）设置新接入的客户端链路 SocketChannel 为非阻塞模式，配置其他的一些 TCP 参数；

（8）将 SocketChannel 注册到 Selector，监听 OP_READ 操作位；

（9）如果轮询的 Channel 为 OP_READ，则说明 SocketChannel 中有新的就绪的数据包需要读取，则构造 ByteBuffer 对象，读取数据包；

（10）如果轮询的 Channel 为 OP_WRITE，说明还有数据没有发送完成，需要继续发送；

17.cookies和session

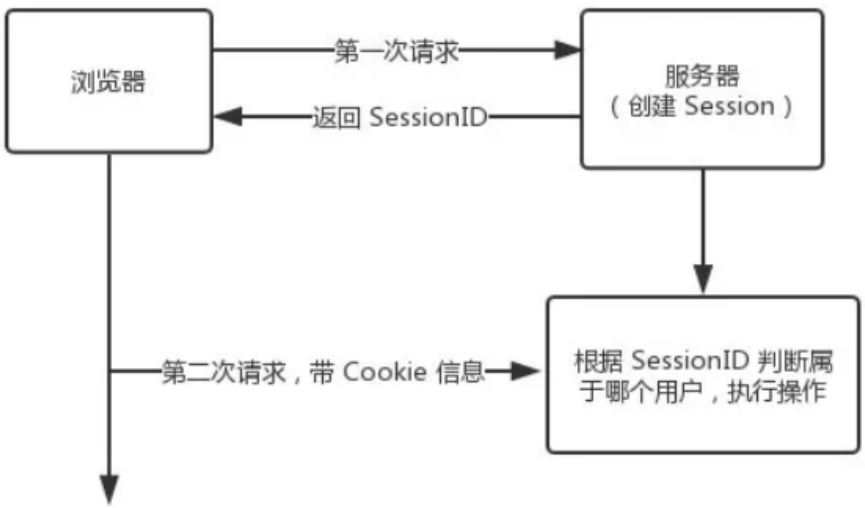
17.1 什么是cookie和session

> cookies(Http cookies、浏览器cookies、web cookie)是服务器发送到浏览器并保存在本地的一个数据块，他会在浏览器下一次向同一台服务器请求时被携带并发送到服务器上。通常是告知服务器两个请求是否来自于同一个浏览器，如保持用户的登陆状态。使得无状态的http协议记录状态变得可能
>
> Cookie 主要用于以下三个方面：
>
> - 会话状态管理（如用户登录状态、购物车、游戏分数或其它需要记录的信息）
> - 个性化设置（如用户自定义设置、主题等）
> - 浏览器行为跟踪（如跟踪分析用户行为等）
>
> Session 代表着服务器和客户端一次会话的过程。Session 对象存储特定用户会话所需的属性及配置信息。这样，当用户在应用程序的 Web 页之间跳转时，存储在 Session 对象中的变量将不会丢失，而是在整个用户会话中一直存在下去。当客户端关闭会话，或者 Session 超时失效时会话结束。

17.2Cookie 和 Session 有什么不同?

> - 作用范围不同，Cookie 保存在客户端（浏览器），Session 保存在服务器端。
> - 存取方式的不同，Cookie 只能保存 ASCII，Session 可以存任意数据类型，一般情况下我们可以在 Session 中保持一些常用变量信息，比如说 UserId 等。
> - 有效期不同，Cookie 可设置为长时间保持，比如我们经常使用的默认登录功能，Session 一般失效时间较短，客户端关闭或者 Session 超时都会失效。
> - 隐私策略不同，Cookie 存储在客户端，比较容易遭到不法获取，早期有人将用户的登录名和密码存储在 Cookie 中导致信息被窃取；Session 存储在服务端，安全性相对 Cookie 要好一些。
> - 存储大小不同， 单个 Cookie 保存的数据不能超过 4K，Session 可存储数据远高于 Cookie。

17.3为什么需要 Cookie 和 Session，他们有什么关联?

> 说起来为什么需要 Cookie ， 这就需要从浏览器开始说起，我们都知道浏览器是没有状态的(HTTP 协议无状态)，这意味着浏览器并不知道是张三还是李四在和服务端打交道。这个时候就需要有一个机制来告诉服务端，本次操作用户是否登录，是哪个用户在执行的操作，那这套机制的实现就需要 Cookie 和 Session 的配合。
>
> 那么 Cookie 和 Session 是如何配合的呢？我画了一张图大家可以先了解下。
>
> 
>

> 用户第一次请求服务器的时候，服务器根据用户提交的相关信息，创建创建对应的 Session ，请求返回时将此 Session 的唯一标识信息 SessionID 返回给浏览器，浏览器接收到服务器返回的 SessionID 信息后，会将此信息存入到 Cookie 中，同时 Cookie 记录此 SessionID 属于哪个域名。
>
> 当用户第二次访问服务器的时候，请求会自动判断此域名下是否存在 Cookie 信息，如果存在自动将 Cookie 信息也发送给服务端，服务端会从 Cookie 中获取 SessionID，再根据 SessionID 查找对应的 Session 信息，如果没有找到说明用户没有登录或者登录失效，如果找到 Session 证明用户已经登录可执行后面操作。
>
> 根据以上流程可知，SessionID 是连接 Cookie 和 Session 的一道桥梁，大部分系统也是根据此原理来验证用户登录状态。

17.4 服务端是根据 Cookie 中的信息判断用户是否登录，那么如果浏览器中禁止了 Cookie，如何保障整个机制的正常运转

> 第一种方案，每次请求中都携带一个 SessionID 的参数，也可以 Post 的方式提交，也可以在请求的地址后面拼接 xxx?SessionID=123456...。
>
> 第二种方案，Token 机制。Token 机制多用于 App 客户端和服务器交互的模式，也可以用于 Web 端做用户状态管理。
>
> Token 的意思是“令牌”，是服务端生成的一串字符串，作为客户端进行请求的一个标识。Token 机制和 Cookie 和 Session 的使用机制比较类似。
>
> 当用户第一次登录后，服务器根据提交的用户信息生成一个 Token，响应时将 Token 返回给客户端，以后客户端只需带上这个 Token 前来请求数据即可，无需再次登录验证。

17.5 考虑分布式 Session 问题

> 在互联网公司为了可以支撑更大的流量，后端往往需要多台服务器共同来支撑前端用户请求，那如果用户在 A 服务器登录了，第二次请求跑到服务 B 就会出现登录失效问题。
>
> 分布式 Session 一般会有以下几种解决方案：
>
> - Nginx ip_hash 策略，服务端使用 Nginx 代理，每个请求按访问 IP 的 hash 分配，这样来自同一 IP 固定访问一个后台服务器，避免了在服务器 A 创建 Session，第二次分发到服务器 B 的现象。
> - Session 复制，任何一个服务器上的 Session 发生改变（增删改），该节点会把这个 Session 的所有内容序列化，然后广播给所有其它节点。
> - 共享 Session，服务端无状态话，将用户的 Session 等信息使用缓存中间件来统一管理，保障分发到每一个服务器的响应结果都一致。

18.序列化,Java怎么做序列化的

18.1.实现序列化:

> 1)让类实现Serializable接口,该接口是一个标志性接口,标注该类对象是可被序列
>
> 2)然后使用一个输出流来构造一个对象输出流并通过writeObect(Obejct)方法就可以将实现对象写出
>
> 3)如果需要反序列化,则可以用一个输入流建立对象输入流,然后通过readObeject方法从流中读取对象

18.2 作用

> 1)序列化就是一种用来处理对象流的机制,所谓对象流也就是将对象的内容进行流化,可以对流化后的对象进行读写操作,也可以将流化后的对象传输与网络之间;
>
> 2)为了解决对象流读写操作时可能引发的问题(如果不进行序列化,可能会存在数据乱序的问题)
>
> 3）序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆

19.面向对象的设计原则(jd)

> 1. 单一职责原则：专注降低类的复杂度，实现类要职责单一；能够增加类的可读性高，进而可以提高系统的可维护性；就一个类而言，应该只有一个引起它变化的原因
> 2. 开闭原则: 所有面向对象原则的核心，设计要对扩展开发，对修改关闭；
> 3. 里式替换原则：实现开放关闭原则的重要方式之一，设计不要破坏继承关系；由于使用基类对象的地方都可以使用子类对象，因此尽量使用基类类型定义对象，而在运行时再确定其子类类型，用子类对象来替换父类对象。
> 4. 依赖倒置原则：系统抽象化的具体实现，要求面向接口编程，是面向对象设计的主要实现机制之一；
> 5. 接口隔离原则：要求接口的方法尽量少，接口尽量细化；
> 6. 迪米特法则：降低系统的耦合度，使一个模块的修改尽量少的影响其他模块，扩展会相对容易；
> 7. 组合复用原则：在软件设计中，尽量使用组合/聚合而不是继承达到代码复用的目的。

20.Java集合(jd)

20.1arraylist的扩容过程

> 添加元素时使用 ensureCapacityInternal() 方法来保证容量足够，如果不够时，需要使用 grow() 方法进行扩容，新容量的大小为 oldCapacity + (oldCapacity >> 1) ，也就是旧容量的 1.5 倍。扩容操作需要调用 Arrays.copyOf() 把原数组整个复制到新数组中，这个操作代价很高，因此最好在创建ArrayList 对象时就指定大概的容量大小，减少扩容操作的次数。

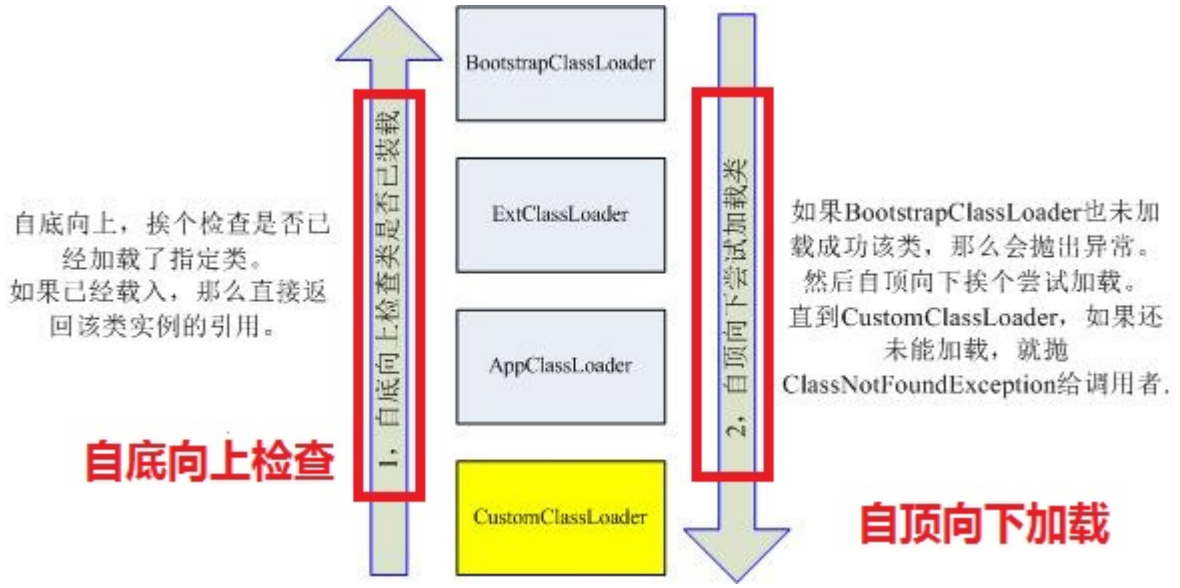
20.2 collection中线程安全的类

> vector 和 currentQueue

21.Java类加载机制(jd)

> 启动类加载器： 一般由c++实现，是虚拟机自身的一部分
>
> 扩展类加载器： 这个类加载器是由ExtClassLoader（sun.misc.Launcher$ExtClassLoader）实现的。它负责将 <JAVA_HOME>/lib/ext 或者被java.ext.dir 系统变量所指定路径中的所有类库加载到内存中，开发者可以直接使用扩展类加载器。
>
> 应用类加载器（系统类加载器）：这个类加载器是由 AppClassLoader（sun.misc.Launcher$AppClassLoader）实现的。由于这个类加载器是 ClassLoader 中的 getSystemClassLoader() 方法的返回值，因此一般称为系统类加载器。它负责加载用户类路径（ClassPath）上所指定的类库，开发者可以直接使用这个类加载器，如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。
>
> 自定义类加载器

21.1双亲委派原则

> 类加载器之间的层次关系，称为双亲委派模型（Parents Delegation Model）。该模型要求除了顶层的启 动类加载器外，其它的类加载器都要有自己的父类加载器。这里的父子关系一般通过组合关系（Composition）来实现，而不是继承关系（Inheritance）。
>
> JVM在加载类时默认采用的是双亲委派机制。通俗的讲，就是某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归 (本质上就是loadClass函数的递归调用)。因此，所有的加载请求最终都应该传送到顶层的启动类加载器中。如果父类加载器可以完成这个类加载请求，就成功返回；只有当父类加载器无法完成此加载请求时，子加载器才会尝试自己去加载。事实上，大多数情况下，越基础的类由越上层的加载器进行加载，因为这些基础类之所以称为“基础”，是因为它们总是作为被用户代码调用的API（当然，也存在基础类回调用户用户代码的情形）。 关于虚拟机默认的双亲委派机制，我们可以从系统类加载器和扩展类加载器为例作简单分析。
>
> 
>

22.操作系统

22.1死锁

> 1.互斥： 每个资源要么已经分配给了一个进程，要么就是可用的。
>
> 2.不可抢占原则： 已经分配给一个进程的资源不能强制性地被抢占，它只能被占有它的进程显式地释放。
>
> 3.请求与保持： 已经得到了某个资源的进程可以再请求新的资源。
>
> 4.循环等待： 有两个或者两个以上的进程组成一条环路，该环路中的每个进程都在等待下一个进程所占有的资源。

22.2 死锁的处理方法

> 1.鸵鸟策略： 对于一些发生死锁不会造成太大影响的情况下，选择不去处理，忽略他。
>
> 2.死锁检测与死锁恢复： 利用算法寻找是否存在环，存在环的话可以利用利用抢占恢复、利用回滚恢复、通过杀死进程恢复
>
> 3.死锁预防： 破坏四个条件

> 4.死锁避免：

22.2 进程通信

> 1.管道： 半双工，只能在父类中使用
>
> 2.FIFO： 命名管道，去除了管道只能在父子进程中使用的限制。
>
> 3.消息队列： 消息队列可以独立于读写进程存在，从而避免了 FIFO 中同步管道的打开和关闭时可能产生的困难；避免了 FIFO 的同步阻塞问题，不需要进程自己提供同步方法；读进程可以根据消息类型有选择地接收消息，而不像 FIFO 那样只能默认地接收。
>
> 4.信号量: 它是一个计数器，用于为多个进程提供对共享数据对象的访问。
>
> 5.共享内存: 允许多个进程共享一个给定的存储区。因为数据不需要在进程之间复制，所以这是最快的一种 IPC。 需要使用信号量用来同步对共享存储的访问。多个进程可以将同一个文件映射到它们的地址空间从而实现共享内存。另外 XSI 共享内存不是使用文件，而是使用内存的匿名段。
>
> 6.套节字： socket，它可用于不同机器间的进程通信。

22.3查看磁盘使用情况

> df -h [文件]
>
> ll -a
>
> 第一个字符代表这个文件的类型（目录，文件或链接文件）
>
> > 若为[d]则是目录 若为[-]则是文件 若为[l]则表示链接文件 若为[b]则表示为设备文件里面的可供存储的周边设备 若为[c]则表示为设备文件里面的串行端口设备，如键盘鼠标
>
> 第一个字母：d代表目录（若为-则代表文件），第2~4代表文件拥有者具有的权限，此处为rwx，即读写执行三种权限都具备；第5~7代表加入此用户组的账号的权限，此处为—，即没有权限；第8~10代表非本人且没有加入用户组的其他账号的权限，此处为—，即没有权限。

23 Integer 和 int

> int 是八大数据类型之一。Integer 是int类型的包装类，把int包装成object对象。int 是数值类型的，Integer是引用类型的是对象。int是基本数据类型Integer是int的一个包装类(wrapper)，他是类不是基本数据类型,他的内部其实包含一个int型的数据那为什么要用Integer呢，主要是因为面向对象的思想，因为Java语言是面向对象的，这也是它只所以流行的原因之一，对象封装有很多好处，可以把属性也就是数据跟处理这些数据的方法结合在一起，比如Integer就有parseInt()等方法来专门处理int型相关的数据，另一个非常重要的原因就是在Java中绝大部分方法或类都是用来处理类类型对象的，如ArrayList集合类就只能以类作为他的存储对象，而这时如果想把一个int型的数据存入list是不可能的，必须把它包装成类，也就是Integer才能被List所接受。所以Integer的存在是很必要的。

24 .排序(虾皮)

24.1 堆排序的时间复杂度

> 排序时间复杂度为O(nlogn)，查找的时间复杂度为O（logn）类是与二叉树的查找
>
> 堆排序的时间复杂度为O（nlogn） 构建堆的过程的时间复杂度为n，调堆的时间复杂度为logn

```
static int len ;//因为后面很多地方需要用到len
public static void heapSort(int[]nums){
    len =nums.length;
    for(int i =len/2;i>=0;i--){//建立最大堆的过程len=nums.length
        adjustMaxHeap(nums,i);
    }
    for(int i =len-1;i>=0;i--){
        int temp = nums[0];
        nums[0]=nums[i];
        nums[i]=temp;
        len--;
        adjustMaxHeap(nums,0);//传入的时候调整数组的长度在减少，所以len值在改变
    }
}
public static  void  adjustMaxHeap(int[] nums,int i) {
    int leftChild = 2 * i + 1;
    int rightChild = 2 * i + 2;
    int largest = i;
    if (leftChild < len&& nums[largest] < nums[leftChild]) {
        largest = leftChild;
    }
    if (rightChild < len && nums[largest] < nums[rightChild]) {
        largest = rightChild;
    }
    if (largest != i) {
        int temp = nums[i];
        nums[i] = nums[largest];
        nums[largest] = temp;
        adjustMaxHeap(nums, largest);
    }
}
```

24.2.TopK(往一个结点不断发送字符串,返回字符串字典序大（小）的十个)

> 采用最小（大）的堆法。先读入前十（m）个字符串创建大小为m的小（大）顶堆，建堆的时间复杂度为O（mlogm），然后遍历接下来的字符串，并与最小（大）堆的堆顶元素比较，如果比堆顶元素小（大）则继续读取，如果比堆顶元素大（小）则替换堆顶元素重新调整堆为最小（大）堆。直到数据（n个）全部发送完毕。最后的时间复杂度为O（nmlongm）空间复杂度为m。

> 如何和女朋友有效通信不用tcp
>
> 一致哈希

25.开发模式

> 敏捷开发
>
> 瀑布模型

26.数据结构

> LRU

> 堆排,快排,归并
>
> 最小编辑距离
>
> 两层丢鸡蛋最小实验测试出鸡蛋会碎的楼层
>
> 解数独
>
> 跳楼梯(变态跳楼梯)
>
> 机器人寻路
>
> 非严格的最小上升子序列
>
> TopK
>
> 链表的反转 , n个一体反转
>
> 链表的相交节点
>
> 队列实现栈和栈实现队列
>
> 数字和为sum的方法数
>
> 多线程的实现
>
> 还有一些忘记了