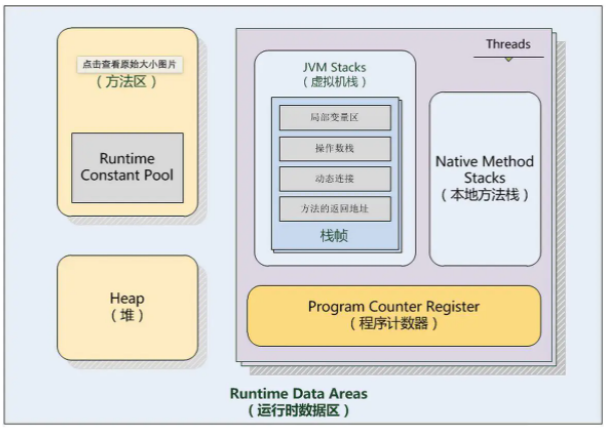
1、JVM内存模型：



栈(Stack)：存放的都是方法中的局部变量(方法的参数，或者方法{}内部的变量)，一旦超出作用域，立刻从栈内存中消失。（方法的运行一定要在栈当中）；

堆(Heap)：凡是new出来的东西，都在堆中。堆内存里面的东西都有一个地址值，堆内存里面的数据都有默认值；

方法区(Method Area)：存储.class相关信息，包含方法的信息；

本地方法栈(Native Method Stack)：与操作系统相关；

程序计数器(PC Register)：与CPU相关。

2、JVM垃圾回收算法和垃圾收集器

jvm 中，程序计数器、虚拟机栈、本地方法栈都是随线程而生随线程而灭，栈帧随着方法的进入和退出做入栈和出栈操作，实现了自动的内存清理，因此，我们的内存垃圾回收主要集中于 **堆和方法区**中，在程序运行期间，这部分内存的分配和使用都是动态的。

（1）垃圾收集算法：

1> 引用计数法：给对象添加引用计数器，当引用对象时计数器+1，引用失效时，计数器-1，当计数器=0时，对象失效，内存可以被回收。优点是：实现简单高效；缺点是：对象之间的互相循环引用问题不好解决；

2> 根搜索法（可达性分析法）：基本思想是，通过一系列的“GC Roots”对象作为起点进行搜索，如果在“GC Roots”和一个对象之间没有可达路径，则称该对象是不可达的。不过被判定为不可达的对象不一定就会成为可回收对象，被判定为不可达的对象要成为可回收对象必须至少经历两次标记过程，如果在这两次标记过程中仍然没有逃脱成为可回收对象的可能性，则基本上就真的成为可回收对象了。

哪些对象可以视为GC roots？

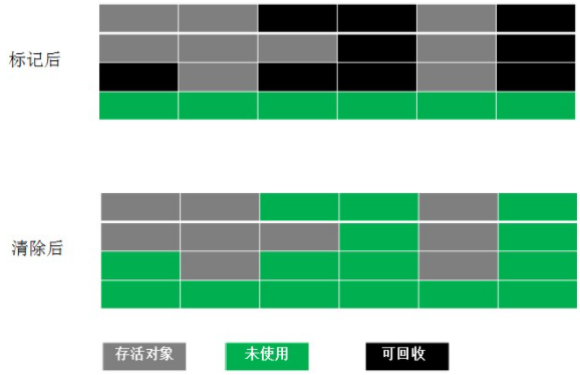
a. 虚拟机栈中（即栈帧中的本地变量）的引用对象；

b. 本地方法栈中的引用对象；

c. 方法区（永久代）中的静态变量引用的对象和常量池中引用的对象

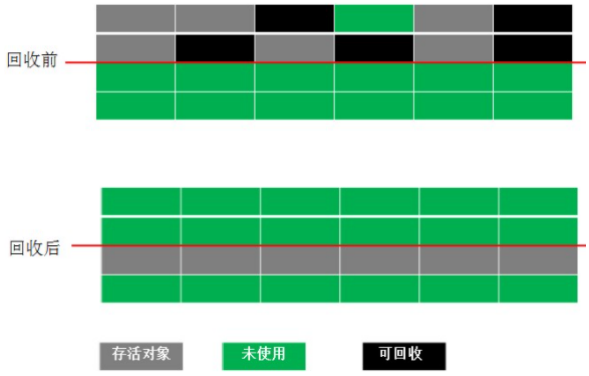
（2）垃圾回收算法：

1>标记-清除（Mark-Sweep）算法：分为两个阶段，标记阶段和清除阶段。标记阶段的任务是标记出所有需要被回收的对象，清除阶段就是回收被标记的对象所占用的空间。



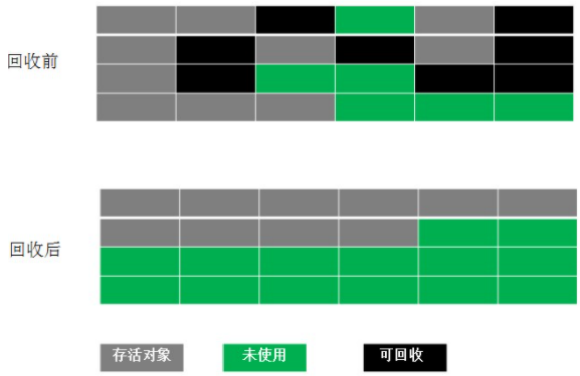
标记-清除算法实现起来比较容易，但是有一个比较严重的问题就是容易产生内存碎片，碎片太多可能会导致后续过程中需要为大对象分配空间时无法找到足够的空间而提前触发新的一次垃圾收集动作。

2> 复制（Coping）算法：为了解决Mark-Sweep算法的缺陷，Coping算法就被提了出来，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块；当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再将已使用的内存空间一次清理掉，这样一来就不容易出现内存碎片的问题。



这种算法虽然实现简单，运行高效，且不容易产生内存碎片，但是能够使用的内存缩减到原来的一半，Coping算法的效率跟存活对象的数目有很大的关系，如果存活对象很多，Coping算法的效率将会大大降低。

3> 标记-整理算法（Mark-Compact）：为了解决Coping算法的缺陷，充分利用内存空间，提出了Mark-Compact算法，该算法标记阶段和Mark-Sweep一样，但是在完成标记之后，它不是直接清理可回收对象，而是将存活对象都向一端移动，达到清除内存碎片的问题。



4> 分代收集算法（Generational Collection）：目前大部分JVM的垃圾收集器采用的算法，它的核心思想是根据对象存活的生命周期将内存划分为若干个不同区域，一般情况下，将堆区划分为老年代和新生代，老年代的特点是每次垃圾收集时，只有少量对象需要被回收，而新生代的特点是，每次垃圾回收时都有大量的对象需要被回收，那么就可以根据不同代的特点，采取最合适的收集算法。（执行GC多次后，依然存活的对象会被转移至老年代。）（持久代：或者称为方法区，通常持久代用来保存类常量以及字符串常量。持久代主要回收两种：常量池中的常量，无用的类信息）

目前，大部分垃圾收集器对于新生代都采取**复制算法**，因为新生代中每次垃圾回收都要回收大部分的对象，也就是说需要复制的操作次数较少。老年代一般使用的是**标记-整理算法。**

（3）典型的垃圾收集器

1> Serial/Serial Old收集器，单线程收集器，在它进行垃圾收集时，必须暂停所有用户线程。Serial收集器是针对新生代的收集器，采用的是Coping算法，Serial Old收集器是针对老年代的收集器，采用的是Mark-Compact算法，它的优点是实现简单高效，缺点是会给用户带来停顿。

2> ParNew收集器，是Serial收集器的多线程版本，使用多个线程进行垃圾收集。

3> Parallel Scavenge收集器，是新生代的多线程收集器，它在回收期间不需要暂停其他用户，其采用的是Coping算法，该收集器与前两个收集器有所不同，它主要是为了达到一个可控的吞吐量。

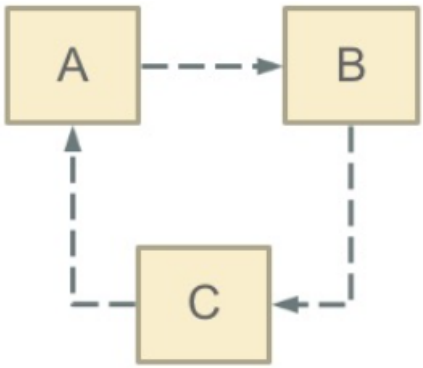
4> Parallel Old收集器，是Parallel Scavenge收集器的老年代版本，使用多线程和mark-Compact算法。

5> CMS（Concurrent Mark Sweep）收集器，是一种以获取最短回收停顿时间为目标的收集器，它是一种并发收集器，采用的是Mark-Sweep算法。

6> G1收集器，是当今收集器技术发展最前沿的成果，它是一款面向服务端引用的收集器，它能充分利用多CPU、多核环境，因此，它是一款并行与并发收集器，并且它能建立可预测的停顿时间模型。

3、Spring循环依赖的三种方式以及解决方法。

（1）什么是循环依赖：循环依赖其实就是循环引用，也就是两个或两个以上的bean互相持有对方，最终形成闭环。比如，A依赖于B，B依赖于C，C又依赖于A。



（2）Spring循环依赖的场景有：1> 构造器的循环依赖；2> field属性的循环依赖

其中，构造器的循环依赖问题无法解决，只能抛出BeanCurrentlyInCreationException异常，在解决属性循环依赖时，Spring采取的是提前暴露对象的方法。

（3）怎么检测是否存在循环依赖

Bean在创建的时候，可以给该Bean打标，如果递归调用回来发现正在创建的话，即说明了循环依赖。

（4）三种循环依赖

1> 构造器的循环依赖（这个Spring解决不了）

Spring容器会将每一个正在创建的Bean标识符放在一个“当前创建Bean池”中，Bean标识符在创建过程中将一直保持在这个池中，如果在创建过程中发现自己已经在“当前创建Bean池”中，将会抛出BeanCurrentlyInCreationException异常，表示循环依赖；而对于创建完毕的Bean，将从“当前创建Bean池”中清除掉。

2> Setter方式的循环依赖，单例

Spring先用无参构造实例化Bean对象，并放到一个map中，且Spring提供了获取这个未设置属性的实例化对象引用的方法。Spring实例化了各个对象后，紧接着会去设置对象的属性，此时A依赖B，就会去Map中取出存在里面的单例B对象，以此类推，不会出来循环问题。

3> Setter方式的循环依赖，多例

多例，即scope="prototype"时，每次请求都会创建一个实例对象，Spring容器不进行缓存，因此无法提前暴露一个创建中的Bean，Spring容器也无法完成依赖注入。

（5）Spring为了解决单例的循环依赖问题，使用了三级缓存。

singletonFactories： 单例对象工厂的cache

earlySingletonObjects：提前暴光的单例对象的Cache

singletonObjects：单例对象的cache

在创建bean的时候，首先想到的是从cache中获取这个单例的bean，这个缓存就是singletonObjects。如果获取不到，并且对象正在创建中，就再从二级缓存earlySingletonObjects中获取。如果还是获取不到且允许singletonFactories通过getObject()获取，就从三级缓存singletonFactory.getObject()(三级缓存)获取，如果获取到了则：从singletonFactories中移除，并放入earlySingletonObjects中。其实也就是从三级缓存移动到了二级缓存。

（6）总结

不要使用基于构造函数的依赖注入，可以通过以下方式解决：

1.在字段上使用@Autowired注解，让Spring决定在合适的时机注入

2.用基于setter方法的依赖注入。

4、数据库索引、为什么索引比较快、调优、存储引擎、Mysql索引查询失效的情况

（1）Mysql索引类型包括：1.普通索引；2.唯一索引（与普通索引类似，但索引列的值必须唯一，允许有空值）；3.主键索引（特殊的唯一索引，不允许有空值）；4.组合索引（列值的组合必须唯一）

（2）一个没有加索引的表，它的数据无序的放置在磁盘存储器上，一行一行的排列的很整齐；如果给表加上了索引，磁盘上的存储结构就由整齐排列的结构转换成了树状结构（平衡树结构），首先索引定位到叶节点，再通过叶节点取到对应的数据行。

（3）数据库优化4个维度：硬件、系统配置、数据库表结构、SQL及索引

（4）数据库SQL调优的方式：

1.创建索引：要尽量避免全表扫描，首先考虑再where及order by涉及的列上建索引；在经常需要进行检索的字段上创建索引；一个表的索引数最好不要超过6个，索引并不是越多越好，索引可以提高相应的select的效率，但同时也降低了insert和update的效率，因为insert和update有可能会重建索引。

2.避免在索引上使用计算（不会使用索引，会全表查询）。

3.使用预编译查询。

4.调整where字句中的连接顺序，where是自上而下被解析的，所以过滤掉最大数据记录的最好放在前面。

5.尽量将多条SQL压缩到一句SQL；

6.用where代替having，having只会在检索出所有记录之后才对结果集进行过滤，而where是在聚合前筛选记录。

7.使用表的别名，减少解析时间并减少列名歧义引起的语法错误。

8.用union all替换union，如果判断结果中不会有重复记录，应该用union all提高效率。

9.考虑使用“临时表”暂存中间结果，将临时结果暂存在临时表，后面的查询可以避免多次扫描主表，也减少了程序执行中“共享锁”阻塞“更新锁”，减少了阻塞，提高了并发性能。但是也得避免频繁创建和删除临时表，以减少系统表资源的消耗。

10.只在必要的情况下才使用事务。

11.尽量避免使用游标。

12.用varchar/nvarchar代替char/nchar，变长字段存储空间小，可以节省空间，对于查询来说，一个相对较小的字段内搜索效率更高。

13.select语句优化，不要使用select *；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；模糊查询不要前置%；对于连续的值能用between不要用in。

14.更新update语句优化：如果只更改一两个字段，不要update全部字段，否则频繁调用会引起明显的性能消耗，同时带来大量的日志。

（5）存储引擎：在Oracle 和SQL Server等数据库中只有一种存储引擎，所有数据存储管理机制都是一样的。而MySql数据库提供了多种存储引擎。

1.InnoDB 存储引擎：支持事务,其设计目标主要面向联机事务处理(OLTP)的应用，其特点是行锁设计、支持外键,并支持类似 Oracle 的非锁定读,即默认读取操作不会产生锁。 从 MySQL 5.5.8 版本开始是默认的存储引擎。

2.MyISAM 存储引擎：不支持事务、表锁设计、支持全文索引,主要面向一些 OLAP 数 据库应用,在 MySQL 5.5.8 版本之前是默认的存储引擎。

（6）Mysql索引失效的情况

1. like 以%开头，索引无效；当like前缀没有%，后缀有%时，索引有效；

2. or语句前后两个数据表字段没有同时使用索引；

3. 组合索引，不是使用第一列索引，索引失效；

4. 数据类型出现隐式转换，如varchar不加单引号的话可能会自动转换为int，使所有无效，产生全表扫描；

5. 在索引列上使用IS NULL或IS NOT NULL操作，索引可能失效；

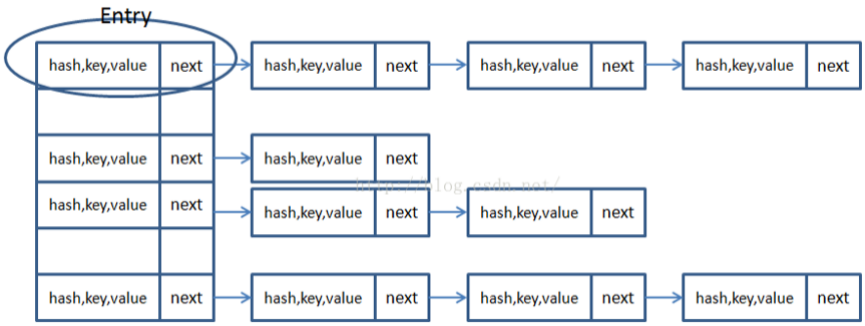
6. 在索引字段上使用not、<>、!=。不等于操作符是永远不会用到索引的，对它的处理只会产生全表扫描。优化方法是：将key<>0改为key>0 or key<0；

7. 对索引字段进行计算操作、字段上使用函数；

8. 当全表扫描速度比索引速度快时，mysql会使用全表扫描，此时索引失效。

5、HashMap、ConcurrentHashMap、HashTable、HashSet区别

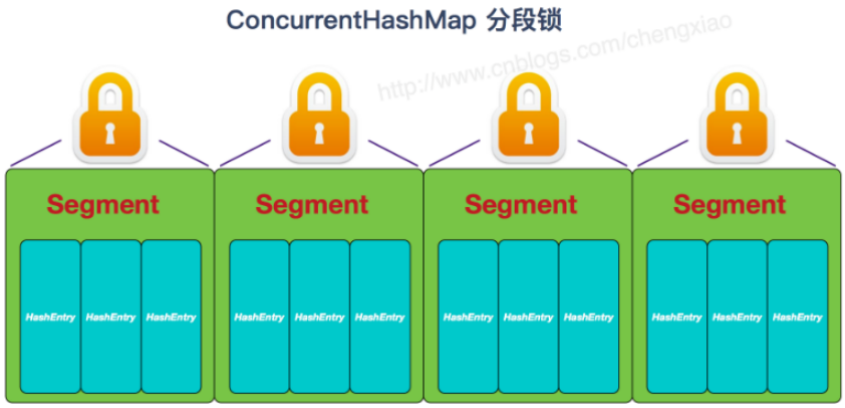
（1）HashMap：Map接口的实现类，无序集合，元素存储和取出的顺序可能不一致，允许使用null值，null键；底层数据结构是哈希表（链表、数组结合体），特点是，存储、取出快，线程不安全，运行快、无序、没有索引，不存储重复元素；（竖向代表数组，java中数组的初始长度默认为16，加载因子默认为0.75，即存储数据时，数组的长度达到16*0.75=12时，数组扩大为原来的两倍（数组在哈希）；横向代表链表）；存储一个键值对时，首先计算key的哈希值，相同hashi值的元素放在一个数组元素中，形成链表，即一个数组元素中存放的是具有相同hash值的链表。 (jdk1.8后链表长度超过8时，链表就转为红黑树，这样大大提高了查找的效率。)



（2）LinkedHashMap：有序集合，存储元素和取出元素的顺序一致；底层数据结构是 哈希表+链表；

（3）HashTable：HashMap和Hashtable都是Map的接口实现类，方法都一样，不同点是，HashMap：线程不安全，运行速度快；允许存储null值，null键；Hashtable：线程安全，运行速度慢；键和值都不允许存储null。HashTable线程安全的策略实现代价却比较大，get/put所有相关操作都是synchronized的，这相当于给整个哈希表加了一把大锁，多线程访问时候，只要有一个线程访问或操作该对象，那其他线程只能阻塞。

（4）ConcurrentHashMap：线程安全，JDK1.7版本： 容器中有多把锁，每一把锁锁一段数据，这样在多线程访问时不同段的数据时，就不会存在锁竞争了，这样便可以有效地提高并发效率。这就是ConcurrentHashMap所采用的"分段锁"思想。ConcurrentHashMap的get操作上面并没有加锁。所以在多线程操作的过程中，并不能完全的保证一致性，是**弱一致性**，关于弱一致性的原因，主要是因为Java的内存模型，写操作所做的修改并没有即时的刷到内存，导致读操作对刚写的内容不可见。在JDK1.8中的实现抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发安全，底层依然采用数组+链表+红黑树的存储结构。取消segments字段，直接采用`transient volatile HashEntry<K,V> table`保存数据，采用table数组元素作为锁，从而实现了对每一行数据进行加锁，进一步减少并发冲突的概率。将原先table数组＋单向链表的数据结构，变更为table数组＋单向链表＋红黑树的结构，查询的复杂度降低，改进了性能。

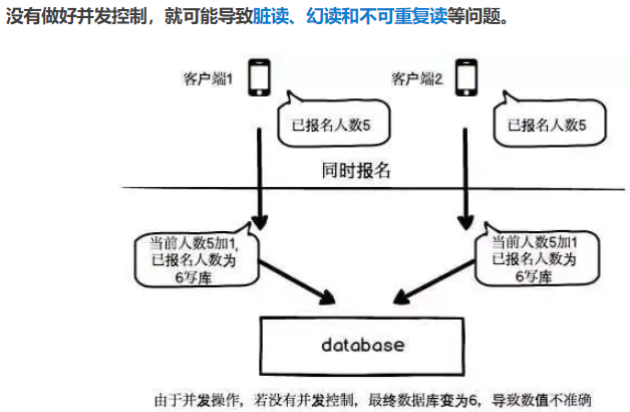


（5）HashMap：实现了Map接口；存储键值对；使用put方法将元素放到map中；使用键计算hashcode；比HashSet运行速度快（因为它使用唯一的键获取对象）；

HashSet：实现Set接口；仅存储对象；调用add方法向Set中添加元素；使用成员对象计算hashcode值，若两个对象的hashcode值相同，使用equals方法判断两个对象是否相等；较HashMap运行慢

6、乐观锁和悲观锁

（1）并发控制：当程序中可能出现并发的情况时，就需要保证在并发情况下数据的准确性，以此确保当前用户和其他用户一起操作时，所得到的结果和他单独操作时的结果时一样的，这种手段就叫做并发控制。并发控制的目的是保证一个用户的工作不会对另一个用户的工作产生不合理的影响。

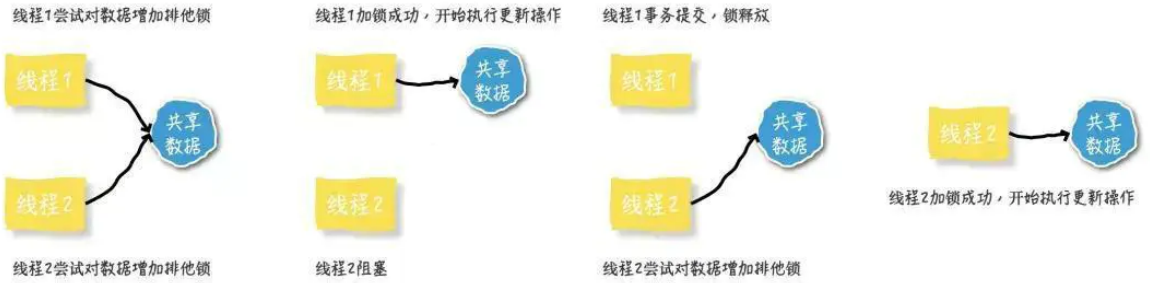


常说的并发控制，一般都和数据库管理系统（DNMS）有关，在DBMS中的并发控制的任务，是确保在多个事务同时存取数据库中的同意数据时，不破坏事务的隔离性、一致性和数据库的统一性。

实现并发控制的主要手段大致可以分为**乐观并发控制**和**悲观并发控制**两种，不仅仅是关系型数据库系统中有乐观锁和悲观锁的概念，像Hibernate、memcache等都有类似的概念，所以，不应该拿乐观锁、悲观锁和其他数据库锁等进行对比。**乐观锁比较适用于读多写少的情况（多读场景），悲观锁比较适用于写多读少的情况（多写场景）**

（2）悲观锁：当要对数据库中的一条数据进行修改时，为了避免同时被其他人修改，最好的办法就是直接对该数据加锁以防止并发。这种借助数据库锁机制，在修改数据之前先锁定，再修改的方式被称之为悲观并发控制，又名悲观锁。

悲观锁具有强烈的独占和排他性，它指的是对数据被外界修改持保守态度，再整个数据处理过程中，将数据处于锁定状态，悲观锁的实现，往往依靠数据库提供的锁机制。



之所以叫做悲观锁，是因为这是一种对数据的修改持有悲观态度的并发控制方式，总是假设最坏的情况，每次读取数据的时候都默认其他线程会更改数据，因此需要进行加锁操作，当其他线程想要访问数据时，都需要阻塞挂起。悲观锁的实现：传统的关系型数据库使用行锁、表锁、读锁、写锁等，都是在操作之前先上锁；Java里面的同步使用synchronized关键字实现。

悲观锁主要分为**共享锁**和**排他锁**，**共享锁**又称读锁，简称S锁，多个事务对于同一数据可以共享一把锁，都能访问到数据，但是只能读不能修改；**排他锁**又称写锁，简称X锁，不能与其他锁并存，如果一个事务获取了一个数据行的排他锁，其他事务就不能再获取该行的其他锁，包括共享锁和排他锁，但是排他锁的事务是可以对数据行读取和修改的。

悲观并发控制实际上是“先取锁再访问”的保守策略，为数据处理的安全提供了保证。但是，在效率方面，处理加锁的机制会让数据库产生额外的开销，还有增加产生死锁的机会。另外，还会降低并行性，一个事务如果锁定了某行数据，其他事务就必须等待该事务处理完才可以处理那行数据。

（3）乐观锁

乐观锁假设数据一般情况下不会造成冲突，所以，在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则返回给用户错误的信息，让用户决定如何去做，乐观锁适合读操作多的场景，这样可以提高程序的吞吐量。

乐观锁是相对悲观锁而言，也是为了避免数据库幻读、业务处理时间过长引起数据处理错误的一种机制，但乐观锁不会刻意使用数据库本身的锁机制，而是依据数据本身来保证数据的正确性。乐观锁的实现：CAS实现；版本号控制，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+1，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值与当前数据库中的version值相等才能更新，否则重试更新操作，直到更新成功。

乐观并发控制相信事务之间的数据竞争的概率是比较小的，因此尽可能直接做下去，直到提交的时候才去锁定，所以不会产生任何锁和死锁。

（4）悲观锁的实现方式：

1> 在对记录进行修改前，先尝试为该记录加上排他锁；

2> 如果加锁失败，说明该记录正在被修改，那么当前查询可能要等待或者抛出异常，具体响应方式由开发者根据实际需要决定；

3> 如果成功加锁，那么就可以对记录做修改，事务完成后就会解锁了；

4> 期间如果有其他对该记录做修改或加排他锁的操作，都会等待解锁或直接抛出异常。

Mysql InnoDB默认行级锁。行级锁都是基于索引的，如果一条sql语句用不到索引，是不会使用行级锁的，会使用表级锁，把整张表锁住。

（5）乐观锁的实现方式

乐观锁不需要借助数据库的锁机制，主要就是两个步骤：冲突检测和数据更新，比较典型的就是CAS（Compare and Swap比较并交换）。

CAS是解决多线程并行情况下使用锁造成性能损耗的一种机制，CAS操作包含三个操作数-内存位置(V)、预期原值(A)和新值(B)，如果内存位置的值与预期原值相匹配，那么处理器会自动将该位置的值更新为新值，否则，处理器不做任何操作。无论哪种情况，它都会在CAS指令之前返回该位置的值。

在高并发环境下，要尽量减少乐观锁的力度，否则只有一个线程可以修改成功，会存在大量的失败，所以，要控制好锁的粒度，在保证数据安全的情况下，可以大大提升吞吐率，进而提升性能。

（6）CAS性能优化

大量的线程同时并发修改一个数据，可能有很多线程会不停的自旋，进入到一个无限重复的循环中。这些线程不停地获取值，然后发起CAS操作，但发现这个值被别人修改过了，于是再次进入下一个循环，获取值，发起CAS操作又失败，再次进入下一个循环，导致大量的线程空循环，自旋转，性能和效率都不是特别好。

Java8有一个新的类，LongAdder，它尝试使用分段CAS以及自动分段迁移的方式来大幅度提升多线程高并发执行CAS操作的性能。LongAdder的核心思想就是热点分离，将value值分离成一个数组，当多线程访问时，通过hash算法映射到其中的一个数字进行计数，而最终的结果，就是这些数组的求和累加，这样一来，就减小了锁的粒度。

（7）如何选择

1> 响应效率：如果需要非常高的响应速度，建议采用乐观锁方案，成功就执行，不成功就失败，不需要等待其他并发去释放锁。乐观锁并未真正加锁，效率高，一旦锁的粒度掌握不好，更新失败的概率就会比较高，容易发生业务失败。

2> 冲突频率：如果冲突频率非常高，建议采用悲观锁，保证成功率，冲突频率大，若选用乐观锁，需要多次重试才能成功，代价比较大；

3> 重试代价：如果重试代价大，建议采用悲观锁，悲观锁依赖数据库锁，效率低，更新失败的概率比较低；

4> 乐观锁如果有人在你之前更新了，你的更新应当是被拒绝的，可以让用户重新操作，悲观锁则会等待前一个更新完成。

随着互联网高并发、高性能、高可用的提出，悲观锁已经越来越少的被应用到生产环境中，尤其是并发量比较大的业务场景。

7、锁用过哪些，Synchronized与ReentrantLock的共同点与区别。

（1）乐观锁、悲观锁（共享锁、排他锁Synchronized，ReentrantLock），

可重入锁（Synchronized，ReentrantLock都是可重入锁，某个线程已经获得某个锁，可以再次获取锁而不会出现死锁），

分段锁（ConcurrentHashMap用到了）

（2）Synchronized与ReentrantLock的共同点与区别

相同点：都是加锁方式同步，都是阻塞式的同步，也就是说如果一个线程获得了对象锁，进入了同步块，其他访问该同步块的线程都必须阻塞在同步块外面等待。

区别：1> Synchronized是java语言的关键字，是原生语法层面的互斥，需要JVM实现；

2> ReentrantLock是JDK1.5之后提供的API层面的互斥锁，需要lock()和unlock()方法配合try/finally语法块来完成。

3> ReentrantLock类中提供了一些高级功能：1.等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相对于Synchronized来说可以避免死锁的情况；2.ReentrantLock默认的构造函数是创建非公平锁，但是可以通过参数true设为公平锁，Synchronized是非公平锁。（公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，但公平锁表现的性能不是很好）；3.一个ReentrantLock对象可以绑定多个对象。

8、多线程

（1）多线程的几种实现方法：

1.继承Thread类，重写run方法；

2.实现Runnable接口，重写run方法；

3.实现Callable接口，重写call方法（有返回值）；

（2）线程执行run方法和start方法的区别：

调用run方法，是在当前线程中执行run方法，没有开启新线程；

调用start方法，是开启了一个线程，并自动调用run方法在新线程中执行；

（3）实现Runnable接口实现多线程的好处：

1.避免了单继承的局限性；

2.增强了代码的扩展性，降低了程序的耦合；实现Runnable接口的方式，把设置线程任务和开启线程进行了分离。

（4）如何保证线程安全?

多线程访问共享数据时，会出现线程安全问题；解决方法：同步代码块，同步方法，Lock锁。

（5）同步方法和静态同步方法的区别：

同步方法：同步方法的锁对象是this；

静态同步方法：静态同步方法的锁对象是本类的class属性（不能是this，this是创建对象之后产生的，静态优先于对象存在）。

9、Java8新特性

（1）**Lambda 表达式** – Lambda 允许把函数作为一个方法的参数（函数作为参数传递到方法中）。

（2）**方法引用** – 可以直接引用已有Java类或对象（实例）的方法或构造器。

（3）**默认方法** –一个在接口里面有了一个实现的方法。

（4）**新工具** – 新的编译工具，如：Nashorn引擎 jjs、 类依赖分析器jdeps。

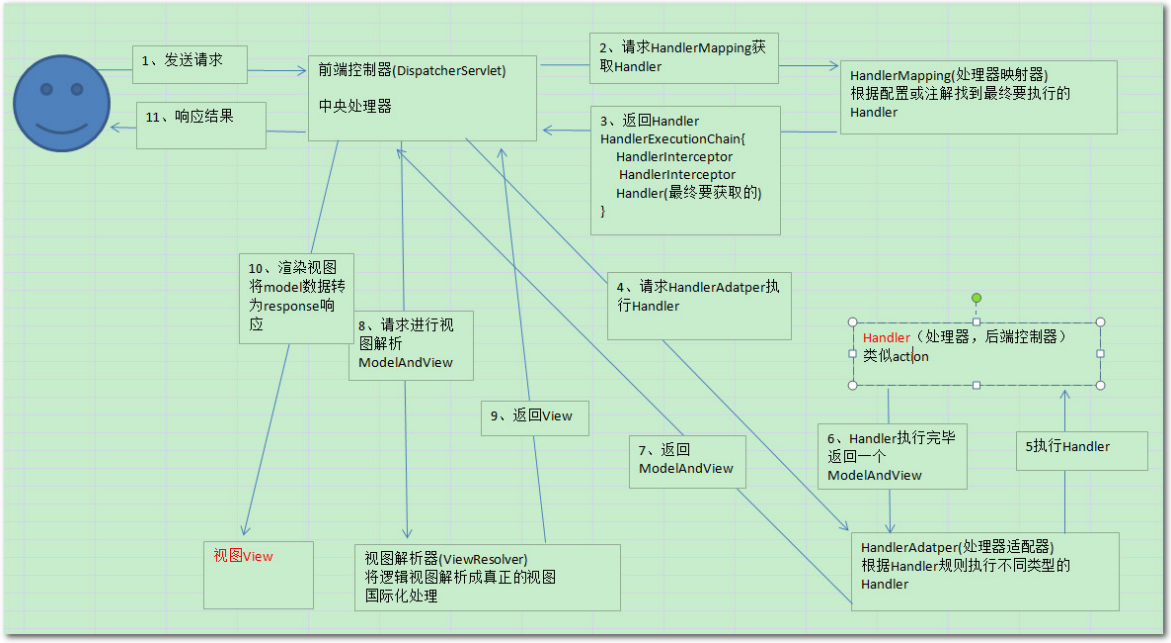
（5）**Stream API** –新添加的Stream API（java.util.stream） 把真正的函数式编程风格引入到Java中。

（6）**Date Time API** – 加强对日期与时间的处理。

（7）**Optional 类** – Optional 类已经成为 Java 8 类库的一部分，用来解决空指针异常。

（8）**Nashorn, JavaScript 引擎** – Java 8提供了一个新的Nashorn javascript引擎，它允许我们在JVM上运行特定的javascript应用。

10、SpringMVC工作流程



（1）用户向服务端发送一次请求，这个请求会先到前端控制器DispatcherServlet（也叫中央处理器）；

（2）DispatcherServlet接收到请求后会调用HandlerMappering处理器映射器，由此得知，该请求由哪个Controller来处理（并未调用Controller，至少得知）；

（3）DispatcherServlet调用HandlerAdapter处理器适配器，告诉处理器适配器应该要去执行哪个Controller；

（4）HandlerAdapter处理器适配器去执行Controller并得到ModelAndView(数据和视图)，并层层返回给DispatcherServlet；

（5）DispatcherServlet将ModelAndView交给ViewReslover视图解析器解析，然后返回真正的视图。

（6）DispatcherServlet将模型数据填充到视图中；

（7）DispatcherServlet将结果响应给用户。

11、事务、Spring事务、事务失效怎么办

（1）事务：一系列的动作综合在一起的一个完整的工作单元，这些动作必须全部完成，如果有一个失败的话，事务就会回滚到最开始的状态。

（2）事务有4个特性ACID：

1> 原子性（Atomicity）：事务由一系列动作组成，事务的原子性确保动作要么全部完成，要么完全不起作用。

2> 一致性（Consistency）：一旦事务完成，事务就会被提交。

3> 隔离性（Isolation）：可能有许多事务会同时处理相同的数据，因此，每个事务都应该与其他事务隔离开来，防止数据损坏。

4>持久性（Durability）：一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响。通常情况下，事务的结果会被写到持久化容器中。

（3）Spring中的事务管理：Spring既支持编程式事务管理，也支持声明式事务管理

编程式事务管理：将事务管理代码嵌入到业务方法中，来控制事务的提交和回滚，在编程式事务中，必须在每个业务中包含额外的事务管理代码（需要在代码中显式调用beginransaction()、commit()、rollback()等事务管理方法）。

声明式事务管理：它将事务管理代码从业务方法中分离出来，以声明的方式来实现事务管理。声明式事务管理有两种方式，一种是在配置文件（XML）中做相关的事务规则说明，另一种是基于Transactional注解的方式（如果加了@Transactional注解，此时Spring会使用AOP思想，在这个方法执行之前开启一个事务，执行完毕之后，根据方法是否报错，决定回滚还是提交事务）。

（4）@Transactional失效的场景：

1> 底层数据库引擎不支持事务：如果数据库引擎不支持事务，Spring自然无法支持事务，以Mysql为例，MyISAM引擎不支持事务操作，InnoDB引擎支持事务操作；

2> 在非public修饰的方法使用：@Transactional注解使用的是AOP，在使用动态代理的时候只能针对public方法进行代理；

3> 在整个事务的方法中使用try...catch语句对异常进行了捕获，而catch语句块没有，导致异常无法抛出，自然会导致事务失效；

4> 默认在业务代码中抛出RuntimeException异常，事务回滚，但抛出Exception，事务不回滚；若想触发其他异常的回滚，需要配置@Transactional(rollbackFor = Exception.class)

5> 方法中调用同类的方法：一个类中的A方法（未标注声明式事务）在内部调用了B方法（标注了声明式事务），这样会导致B方法中的事务失效；解决方案，将B方法写在另一个类ServiceB中，在A方法所在的类ServiceA中注入ServiceB，并给A方法加上@Transaction注解，再调用B方法，B方法上的注解改为@Transaction(propagation=Propagation.REQUIRES_NEW)

6> 如果采用Spring+SpringMVC，则context:component-scan重复扫描问题可能会导致事务失败。（如果Spring和MVC的配置文件中都扫描了service层，那么事务就会失效，原因是：按照spring配置文件的加载顺序来讲，先加载SpringMVC的配置文件，再加载Spring的配置文件，事务一般都在Spring配置文件中进行配置，如果再加载SpringMVC配置文件的时候，把service也给注册了，但此时事务还没有加载，也就导致后面的事务无法成功注册到service中。所以要把对service的扫描放在Spring配置文件中或是其他配置文件中）

（5）数据库的4种隔离级别

数据库事务的隔离级别有4种，由低到高分别为Read uncommitted 、Read committed 、Repeatable read 、Serializable 。而且，在事务的并发操作中可能会出现脏读，不可重复读，幻读。

Read uncommitted：**读未提交**，顾名思义，就是一个事务可以读取另一个未提交事务的数据，会造成脏读。

Read committed：**读提交**，顾名思义，就是一个事务要等另一个事务提交后才能读取数据。若有事务对数据进行更新（UPDATE）操作时，读操作事务要等待这个更新操作事务提交后才能读取数据，可以解决脏读问题。但在一个事务范围内两个相同的查询可能会返回不同数据，这就是不可重复读。

Repeatable read：**重复读**，就是在开始读取数据（事务开启）时，不再允许修改操作。不可重复读**对应的是修改，即UPDATE操作。**但是可能还会有幻读问题。因为**幻读问题对应的是插入INSERT操作，而不是UPDATE操作。**

Serializable 序列化：Serializable 是最高的事务隔离级别，在该级别下，事务串行化顺序执行，可以避免脏读、不可重复读与幻读。但是这种事务隔离级别效率低下，比较耗数据库性能，一般不使用。

12、mongoDB和Redis

（1）mongoDB是以文档的形式存储数据的；

（2）Redis是以键值对的形式存储数据的；

（3）Redis持久化方式：

1. RDB（快照形式写入磁盘，注重结果，可能会丢数据，数据特别多时，比较适合，无需配置）、

2. AOF（日志形式，注重过程，存储及时，默认没开）。

（4）缓存雪崩：Redis缓存中key同一时间大量失效，导致大量请求发送到数据库，造成数据库挂掉。

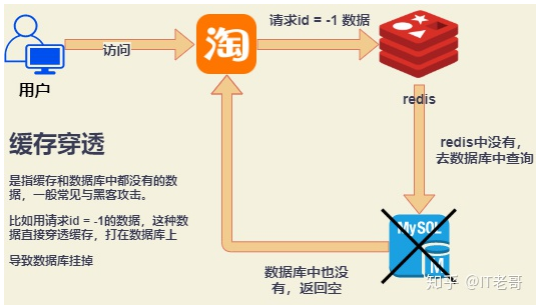
解决方案：1. 随机设置key失效时间，避免大量key集体失效；

2. 若是集群部署，可将热点数据均匀分布在不同的Redis库中，也能避免key全部失效的问题；

3. 不设置过期时间；

4. 跑定时任务，在缓存失效前刷进新的缓存。

（5）缓存穿透：Reids缓存和数据库中都没有相关数据，Redis中没有这样的数据，无法进行拦截，直接穿透到数据库，导致数据库压力过大宕机。



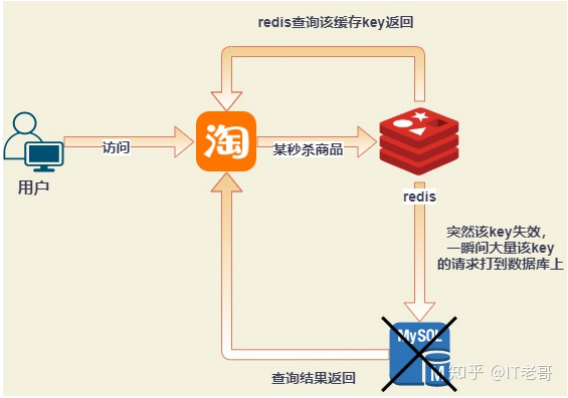
解决方案：1. 当查询返回一个空数据时，直接将这个空数据存到缓存中，过期时间不宜设置过长，建议不超过5分钟；

2. 拉黑该IP地址；

3. 对参数进行校验，不合法参数进行拦截；

4. 采用布隆过滤器，将所有可能存在的数据，分别通过多个哈希函数生成多个哈希值，然后将这些哈希值存到一个足够大的bitmap中，此时，一个一定不存在的数据就会被这个bitmap拦截，从而减少了数据库的查询压力。

（6）缓存击穿：某一个热点key，在不停地扛着高并发，当这个热点key在失效的一瞬间，持续的高并发访问就击破缓存直接访问数据库，导致数据库宕机。



解决方案：1. 设置热点数据“永不过期”；

2. 加上互斥锁：缓存击穿的现象是多个线程同时去查询数据库的这条数据，那么我们可以在第一个查询数据的请求上使用一个互斥锁来锁住它，其他的线程走到这一步拿不到锁就等着，等第一个线程查询到了数据，然后将数据放到Redis缓存起来，后面的线程发现已经有缓存了，就直接走缓存。

总结：

雪崩是大面积的key缓存失效；**穿透**是Redis里不存在这个缓存key；**击穿**是Redis中某一个热点key突然失效，最终的受害者都是数据库。

思考：

`未雨绸缪`：将Redis、Mysql等搭建成高可用的集群，防止单点；

`亡羊补牢`：服务中进行限流+降级，防止Mysql被打崩溃；

`重振旗鼓`：Redis持久化，宕机重启，自动从磁盘上加载数据，快速恢复缓存数据。

13、Spring Cloud Config：在分布式系统中，由于服务数量巨多，为了方便服务配置文件统一管理，实时更新，所以需要分布式配置中心组件。在Spring Cloud中，有分布式配置中心组件`Spring Cloud Config`，它支持配置服务放在配置服务的内存中（即本地），也支持放在远程Git仓库中。

14、AtomicInteger 是对int的封装，提供原子性的访问和更新操作，其原子性操作的实现是基于CAS。

15、Collection分类：List接口（有序，可重复，存在索引）、Set接口（无序、不可重复）。

16、如何遍历ArrayList集合，并安全删除其中的元素?

如果是遍历删除list集合中的某个特定的元素，使用for循环、增强for循环、迭代器iterator都可以；

如果要循环遍历删除多个元素，最好使用迭代器。其次使用普通for循环，反过来遍历list集合。

17、ThreadLocal的介绍

（1）什么是ThreadLocal：它是一个数据结构，有点儿像HashMap，可以保存key、value键值对，但是一个ThreadLocal只能保存一个，并且各个线程的数据互不干扰。

（2）ThreadLocal和Synchronized的区别：

1. Synchronized用于线程间的数据共享，而ThreadLocal则用于线程间的数据隔离。
2. Synchronized是利用锁的机制，使变量或代码块在某一时刻只能被一个线程访问；而ThreadLocal为每一个线程都提供了变量的副本，使得每一个线程在某一时间访问到的并不是同一个对象，这样

就隔离了多个线程对数据的共享。

（3）ThreadLocal中的key和value

ThreadLocal实例的弱引用对象会作为key存放在ThreadLocalMap中，然后set方法加入的值就作为ThreadLocalMap中的value。（在线程内value值是共享的，在线程外value是隔离的，两个线程的value值互不影响）。

（4）ThreadLocal内存泄漏

如果ThreadLocal没有外部强引用，在垃圾回收时，ThreadLocal必定会被回收，而ThreadLocal又作为了Map中的key，ThreadLocal被回收就会导致一个key为null的entry，外部就无法通过key来访问这个entry，垃圾回收也无法回收，这就造成了内存泄漏。

解决方法：1. 每次使用完ThreadLocal都调用它的remove()方法清除数据；

2. 将ThreadLocal变量定义成private static，这样就一直存在ThreadLocal的强引用，也就能保证任何时候都能通过ThreadLocal的弱引用访问到entry的value值，进而清除掉。

18、内存溢出和内存泄漏

内存溢出：指程序申请内存时，没有申请足够的内存供申请者使用，例，申请了int的内存，存了long型的数据；

内存泄漏：指程序在申请内存后，无法释放已申请的内存空间。

19、浮点数相减运算不精确的原因

浮点数由十进制转成二进制时，小数部分不能完全精确表示。

解决办法：使用BigDecimal类，可以支持任意精度的浮点数运算，它的原理是将十进制小数扩大N倍，变成整数，再将十进制数转成二进制，使数据在整数维度上进行计算。

20、封装、继承、多态的特点，并用生活中的场景描述

封装：将一些细节信息隐藏起来，对于外界不可见，生活中的场景：冰箱、电视等

继承：在一组相同或类似的对象中，抽取出共性的特征（属性）和行为（方法），实现重用性；继承的特点：1. 子类可以拥有父类的内容；2. 子类还可以拥有自己的专有内容； 生活中的场景：动物都要吃饭，但具体吃的东西可能不同；

多态：父类引用指向子类对象，继承或实现是多态性的前提；生活中的场景：可以将人这个引用指向程序员对象。

21、抽象类和接口的区别：

抽象类：用abstract修饰；不能实例化；可以有构造器，抽象方法的修饰符可以是public、protected、default，可以含有抽象方法，也可以有具体方法；子类若没有重写父类的所有抽象方法，子类也是抽象类；抽象类只能被单继承；

接口：用interface修饰；不能被实例化；不能有构造器；方法的默认修饰符是public，不能有其他修饰符；接口中的方法全部是抽象的；可以实现多个接口。

22、使用notify时是先释放锁还是先唤醒线程

使用notify不释放锁，只有等到同步方法执行结束才会释放锁，它只是告诉调用过wait方法的线程可以去参与竞争锁了，即唤醒线程；使用wait方法，会释放锁。

23、死锁

（1）什么是死锁：两个或两个以上的线程在执行过程中，由于竞争资源或由于彼此通信而造成的一种阻塞现象，若无外力作用，它们都将无法推进下去，此时成系统处于死锁状态或系统产生了死锁。

（2）死锁的必要条件：

1. 互斥条件：在一段事件内某资源仅为一个进程占有，此时若有其他进程请求该资源，则请求进程只能等待；
2. 不可剥夺条件（非抢占）：进程所获得的资源在使用完毕之前，不能被其他进程强行剥夺，即只能由获得该资源的进程自己来释放；
3. 请求与保持（占有并等待）：进程已经保持了至少一个资源，但又提出了新的资源要求，而该资源已经被其他进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放；
4. 循环等待：存在一种进程资源的循环等待链，链中每个进程已获得的资源同时被链中下一个进程所请求。

（3）如何避免死锁

1. 加锁顺序：当多个线程需要相同的锁时，如果能确保所有线程按照相同的顺序获得锁，那么死锁就不会发生；
2. 加锁时限：若一个线程没有在给定的时限内成功获得所需要的锁，则会进行回退并释放所有已经获得的锁，然后等待一段随机的时间再重试；
3. 死锁检测：每当一个线程获得了锁，会在线程和锁相关的数据结构中将其记下，除此之外，每当有线程请求锁，也需要记录在这个数据结构中，当一个线程请求锁失败时，这个线程可以遍历锁的关系图，看看是否有死锁发生。

24、equals和==的区别

（1）了解JVM中内存分配的知识

在JVM中，内存分为堆内存和栈内存，二者的区别是，当我们创建一个对象（new Object）时，就会调用对象的构造函数来开辟空间，将对象数据存储到堆内存中，与此同时，在栈内存中生成对应的引用，当我们在后续代码中调用的时候用的都是栈内存的引用，还需注意的一点，基本数据类型是存储在栈内存中的。

（2）equals和==的区别

1. ==是判断两个变量或实例是不是指向同一个内存空间，是对内存地址进行比较，判断引用是否相同；
2. equals是判断两个变量或实例所指向的同一个内存空间的值是不是相同，比较的包括引用和引用指向的内存中的值（String类重写了Object类的equals方法，比较字符串的内容是否相同）。

25、二分法查找（折半查找）

（1）前提：被查找的数组中的元素，必须有序排列；

（2）折半公式：（最大索引max+最小索引min）/2；

（3）指针思想：折半后的指针索引上的元素 和 被查找的元素比较

若 元素 > 中间索引上的元素，小指针 = 中间指针 + 1；

若 元素 < 中间索引上的元素，大指针 = 中间索引 - 1；

元素 = 中间索引上的元素时，返回该索引，否则 小指针索引 > 大指针索引时 结束，没有找到要查找元素的索引。

26、冒泡排序

（1）冒泡排序是 数组中相邻元素进行比较，将较大的值放在右面，会首先确定最大值。

```
for(int i=0;i<arr.length-1;i++){
    for(int j=0;j<arr.length-i-1;j++){
        if(arr[j]>arr[j+1]){
            交换位置;
        }
    }
}
```

（2）选择排序是 数组中的每个元素都进行比较，将较小值放在前面，会先确定最小值。

```
for(int i=0;i<arr.length-1;i++){
    for(int j=i+1;j<arr.length;j++){
        if(arr[i]>arr[j]){
            交换位置;
        }
    }
}
```

27、ArrayList的扩容机制

ArrayList实现了List接口，底层实现基于数据，因此可以认为是一个可变长数组。以无参构造方式创建时，初始化赋值的是一个空数组，当真正向数组进行添加元素操作时，才真正分配容量，即向数组中添加第一个元素时，数组容量扩为10，当长度超过10时，扩容为原来的1.5倍（int newCapacity = oldCapacity + (oldCapacity >> 1)）。

28、SpringBoot

（1）SpringBoot的核心功能：1. 起步依赖，将具备某种功能的坐标打包在一起；2.自动配置，应用程序启动时，考虑众多因素，决定Spring配置应该用哪个，不该用哪个。

（2）springboot是通过注解`@EnableAutoConfiguration`的方式，去查找，过滤，加载所需的`configuration`，

`@ComponentScan`扫描我们自定义的bean，

`@SpringBootConfiguration`使得被`@SpringBootApplication`注解的类声明为注解类．因此`@SpringBootApplication`的作用等价于同时组合使用`@EnableAutoConfiguration`，`@ComponentScan`，`@SpringBootConfiguration`

29、java常用包名及功能

（1）java.io　提供了通过数据流、对象序列以及文件系统实现的系统输入、输出

（2）java.lang.* java编程语言的基本类库

（3）java.math.* 提供了简明的整数算术以及十进制算术的基本函数

（4）java.net 提供了用于实现网络通讯应用的所有类

（5）java.security.* 提供了设计网络安全方案需要的一些类

（6）java.sql 提供了访问和处理来自Java标准数据源数据的类

（7）java.test 包括以一种独立于自然语言的方式处理文本、日期、数字和消息的类和接口

（8）java.util.* 包括集合类、时间处理模式、日期时间工具等各类常用工具包

（9）....　等

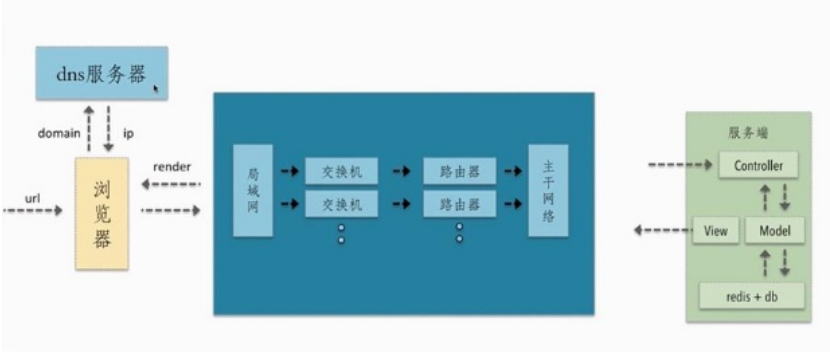
30、java中的Object九大方法

（1）protected Object clone() 创建并返回此对象的一个副本。
（2）boolean equals(Object obj) 指示某个其他对象是否与此对象“相等”。
（3）protected void finalize() 当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法。
（4）Class<? extendsObject> getClass() 返回一个对象的运行时类。
（5）int hashCode() 返回该对象的哈希码值。
（6）void notify() 唤醒在此对象监视器上等待的单个线程。
（7）void notifyAll() 唤醒在此对象监视器上等待的所有线程。
（8）String toString() 返回该对象的字符串表示。
（9）void wait() 导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法。
void wait(long timeout) 导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 方法，或者超过指定的时间量。
void wait(long timeout, int nanos) 导致当前的线程等待，直到其他线程调用此对象的 notify()

31、客户端发起一个网络请求都经历了什么?

（1）简单认识： 客户端获取URL---> DNS解析---> TCP连接---> 发送HTTP请求---> 服务器处理请求---> 返回报文---> 浏览器解析渲染页面---> TCP断开连接

（2）详细经过：



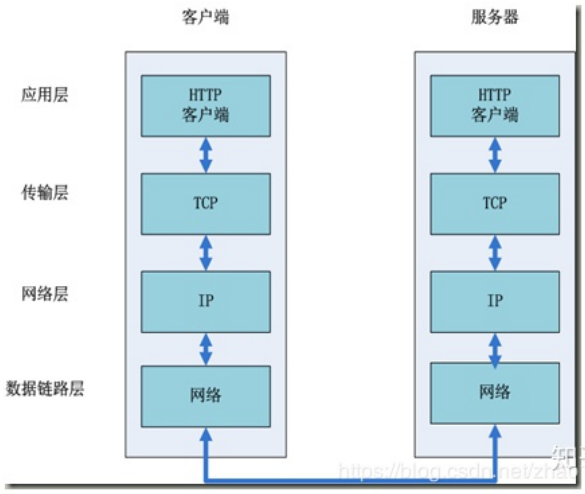
浏览器输入url经历图

网络通信：

1、用户输入`url`，浏览器内部代码将`url`进行拆分解析

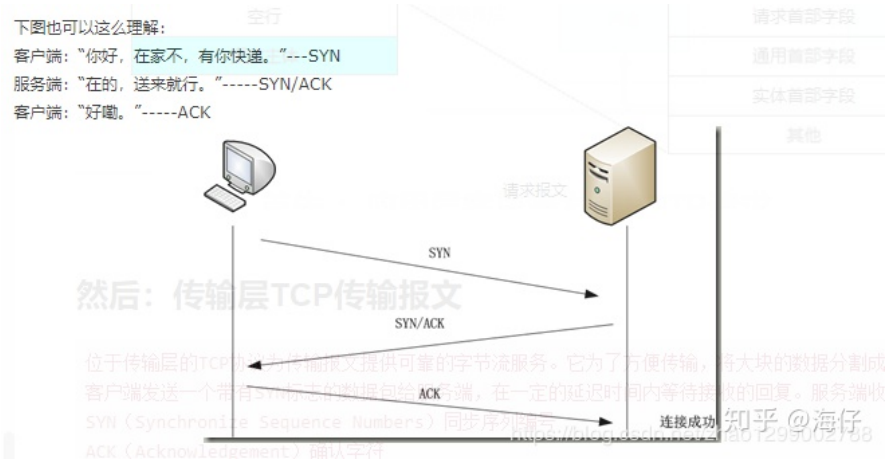
2、浏览器首先去本地的`hosts`文件，检查在该文件中是否有相应的域名，如果有，则向其对应的`IP`地址发起请求，如果没有，就将`domain`（域）发送给`dns`（域名）服务器进行解析，将域名解析成对应的服务器`IP`地址，发回给浏览器

拿到服务器IP后，接下来就是网络通信，分层由高到低分别为：应用层、传输层、网络层、数据链路层。发送端从应用层往下走，接收端从数据链路层往上走。



3、应用层客户端生成HTTP报文并发送HTTP请求：HTTP请求包含请求报头和请求主体两个部分，其中请求报头包含了至关重要的信息，包括请求的方法（GET/POST）、目标url、遵循的协议（http/https/ftp等），返回的信息是否需要缓存，以及客户端是否发送cookie等。

4、传输层TCP传输报文：位于传输层的TCP协议为传输报文提供可靠的字节流服务（通过”三次握手“等方式）。为了方便传输，它将大块的数据分割成以报文段为单位的数据包进行管理，并为它们编号，方便服务器接收时能准确地还原报文信息。



客户端发送一个带有SYN标志的数据包给服务端，在一定的延迟时间内等待接收的回复；服务端收到后，回传一个带有SYN/ACK标志的数据包传达确认信息；最后客户端再回传一个带有ACK标志的数据包，代表握手结束，连接成功。SYN（Aynchronize Sequence Number同步序列编号），ACK（Acknowledgement）确认字符。

5、网络层IP协议查询MAC地址：IP协议的作用是把TCP分割好的各种数据包传送给接收方，而要保证确实能传到接收方还需要接收方的MAC地址，也就是物理地址。IP地址和MAC地址是一一对应的关系，一个网络设备的IP地址可以更换，单MAC地址一般是固定不变的。ARP协议可以将IP地址解析成对应的MAC地址。当通信的双方不在同一个局域网时，需要多次中转才能到达最终的目标，在中转的过程中需要通过下一个中转站的MAC地址来搜索下一个中转目标。

6、数据到达数据链路层：在找到对方的MAC地址后，就将数据发送到数据链路层传输，这时，客户端发送请求的阶段结束。

服务器接收数据：

1、接收端的服务器在链路层接收到数据包，再层层向上，直到应用层。这过程中包含在运输层通过TCP协议将分段的数据包重新组成原来的HTTP请求报文。

2、服务器响应请求：服务器接收到客户端发送的HTTP请求后，查找客户端请求的资源，并返回响应报文，响应报文中包括一个重要的信息--状态码。
服务器端收到请求后，由web服务器（准确说应该是http服务器）处理请求，web服务器解析用户请求，知道了需要调度哪些资源文件，再通过相应的资源文件处理用户请求和参数，并调用数据库信息，最后将结果通过web服务器返回给浏览器客户端。

3、关闭TCP连接（通过四次挥手的方法）：为了避免服务器与客户端双方的资源占用和损耗，当双方没有请求或响应传递时，任意一方都可以发起关闭请求。