# 总结

## Java基础

### 创建对象四种方式

1. New
2. 反射：

使用Class类的newInstance方法

使用Constructor类的newInstance方法

1. 采用clone
2. 采用序列化机制

### ==与equals

==比较两个对象的地址,首先要是同类型 **（是同一个对象）**

对于基本数据类型,他们是作为常量在方法区中的常量池里面以HashSet策略存储起来的,基本数据的包装类型（Byte, Short, Character，Integer，Float, Double，Long,  Boolean）除了Float和Double之外，其他的六种都是实现了常量池, Integer 在常量池中的存储范围为[-128,127]

Equals

Object类中定义的equals方法是**比较地址值（是同一个对象）**;我们可以重写来根据我们定义的规则比较（即相同不一定是同一个对象）

Hashcode

Hashcode比equals快

1. equal()相等的两个对象他们的hashCode()肯定相等，也就是用equal()对比是绝对可靠的。

 2.hashCode()相等的两个对象他们的equal()不一定相等，也就是hashCode()不是绝对可靠的。

### 流

### 序列化:

其实[序列化](https://www.baidu.com/s?wd=%E5%BA%8F%E5%88%97%E5%8C%96&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)就是有格式的[二进制](https://www.baidu.com/s?wd=%E4%BA%8C%E8%BF%9B%E5%88%B6&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "https://zhidao.baidu.com/question/_blank)数据(可以写入读取的规则),序列化就是将内存中的类或者对象（你写的类都是存储在内存中的）变成可以存储到存储媒介中的流，你将类序列化成流之后可以通过互联网传输给别人

定义:将对象的状态信息转换为可以存储或传输的形式的过程。在序列化期间，对象将其当前状态写入到临时或持久性存储区。以后，可以通过从存储区中读取或反序列化对象的状态，重新创建该对象

序列化条件:

1. 必须实现java.io.Serializable 接口
2. 该类的所有属性必须是可序列化的。如果有一个属性不需要序列化的，则该属性必须注明是瞬态的，使用transient 关键字修饰

transient瞬态修饰成员,不会被序列化,不会把这个属性保存到文件中,持久化

Java对象的static，transient 修饰的属性不能被序列化

有些对象并不能被序列化

比如我们用一个对象来关联一个进程，实现对操作系统进程操作的包装。这个对象包含进程id这样的字段，以及诸如复制进程、停止进程这样的方法。那么这样的对象就不能序列化。  
因为序列化没有意义。你将这个对象的存储全部dump下来，重新开机，这个进程id还有意义么？  
同样的场景还发生在那些GUI类型上面、网络通讯的类型上面，它们并不是纯粹的Java对象。

当然，你可以强行去系列化这样的对象，反射遍历字段，然后保存。反之类似。当然我说了，如果一个类型不支持序列化，肯定有深层次的原因，简单的对属性的保存和恢复得到的未必是可用的

### Java的数据类型

#### String、StringBuffer、StringBuilder 的区别？

#### 计算hashcode:

hashCode() 方法用于返回字符串的哈希码。

字符串对象的哈希码根据以下公式计算：

s[0]\*31^(n-1) + s[1]\*31^(n-2) + ... + s[n-1]

使用 int 算法，这里 s[i] 是字符串的第 i 个字符，n 是字符串的长度，^ 表示求幂。空字符串的哈希值为 0。

Aa和BB 的hash值相同.

（1）、可变不可变

String：字符串常量，在修改时不会改变自身；若修改，等于重新生成新的字符串对象。

StringBuffer：在修改时会改变对象自身，每次操作都是对 StringBuffer 对象本身进行修改，不是生成新的对象；使用场景：对字符串经常改变情况下，主要方法：append（），insert（）等。

1. 线程是否安全

String：对象定义后不可变，线程安全。

StringBuffer：是线程安全的（对调用方法加入同步锁），执行效率较慢，适用于多线程下操作字符串缓冲区大量数据。

StringBuilder：是线程不安全的，适用于单线程下操作字符串缓冲区大量数据。

（3）、共同点

StringBuilder 与 StringBuffer 有公共父类 AbstractStringBuilder(抽象类)。StringBuilder、StringBuffer 的方法都会调用 AbstractStringBuilder 中的公共方法，如 super.append(...)。 只是 StringBuffer 会在方法上加 synchronized 关键字，进行同步。最后，如果程序不是多线程的，那么使用 StringBuilder 效率高于 StringBuffer。

### 集合

#### 1.ArrayList 内部用什么实现的

底层 : 用无参构造函数new一个集合时数组长度为0,第一次add数组长度变为默认长度10,新增和插入都是先判断该数组满没满,满了就用grow方法将当前数组的长度变为原来容量的1.5倍.

每次创建新数组,将旧元素复制进去

add(E e) 方法:

执行逻辑如下：

确保数组已使用长度（size）加1之后足够存下 下一个数据

修改次数modCount 标识自增1，如果当前数组已使用长度（size）加1后的大于当前的数组长度(即当前数组已满)，则调用grow方法，增长数组，grow方法会将当前数组的长度变为原来容量的1.5倍。

确保新增的数据有地方存储之后，则将新元素添加到位于size的位置上。

返回添加成功布尔值

add(int index, E element)方法:

指定位置插入元素，具体的执行逻辑如下：

确保数插入的位置小于等于当前数组长度，并且不小于0，否则抛出异常

确保数组已使用长度（size）加1之后足够存下 下一个数据

修改次数（modCount）标识自增1，如果当前数组已使用长度（size）加1后的大于当前的数组长度(数组已满)，则调用grow方法，增长数组grow方法会将当前数组的长度变为原来容量的1.5倍。

确保有足够的容量之后，使用System.arraycopy 将需要插入的位置（index）后面的元素统统往后移动一位。

将新的数据内容存放到数组的指定位置（index）上

注意：使用该方法的话将导致指定位置后面的数组元素全部重新移动，即往后移动一位。

#### 2.Hashmap底层

数组 + 链表 jdk1.8后引入二叉树

先用Hashcade计算哈希值决定在数组哪里,若同一位置再进行equals比较内容放入该位置链表,超过8个变成二叉树,少于6个变成链表.

第一次put会生成一个长度为16的数组,0.75时扩容

为了pun进去分布均匀可以取模但消耗大,java用的&2的N次方-1运算

哈希值对数组长度取模:即取hash值二进制的后四位 100(1100100)/16(2的四次方)=4=0100

100(1100100)/64(2的六次方)=36=100100(hash值后六位)

Java中是&(2的N次方-1:即所有位为1)(与运算:两位同时为“1”，结果才为“1”，否则为0),结果与取模相同,但当扩容时不需要重新计算其在新数组中的位置(原因:16(2的四次方)扩容变为32(2的五次方),100本来在第四个位置扩容后,其二进制第五位为0则仍为第四个位置,位置不变,若为1则翻倍,在第20,可以自己算算结果相同),因此只需要看hash值二进制是0还是1即可得出这个key需不需要变位置

注意:同一个桶的hash值可能不同,hash值同也可能不是同一个对象

#### 3..map是线程安全的么，concurrentmap(分段锁)线程安全的原理是啥?

Hashmap不安全: 键值可以为null

1. 当用在方法内的局部变量时，局部变量属于当前线程级别的变量，其他线程访问不了，所以这时也不存在线程安全不安全的问题了。

当用在单例对象成员变量的时候呢？这时候多个线程过来访问的就是同一个HashMap了，对同个HashMap操作这时候就存在线程安全的问题了。

为了避免出现场景2的线程安全的问题，不能使用HashMap作为成员变量

HashTable安全: 键值不可以为null

HashTable的get/put方法都被synchronized关键字修饰，说明它们是方法级别阻塞的，它们占用共享资源锁，所以导致同时只能一个线程操作get或者put，而且get/put操作不能同时执行，所以这种同步的集合效率非常低，一般不建议使用这个集合

ConcurrentHashMap安全:性能比其他线程安全的map好

线程安全的容器只能保证自身的数据不被破坏，和数据在多个线程间是可见的，但无法保证业务的行为是否正确.

ConcurrentHashMap只对put,remove操作使用了同步操作，get操作并不影响,可以遍历的同时删除及多线程删除但业务操作的线程安全不能保证

在jdk1.6中 数组＋单向链表 数据0.75因子分布

ConcurrentHashMap使用锁分段技术提高并发访问效率。首先将数据分成一段一段地存储，然后给每一段数据配一个锁，当一个线程占用锁访问其中一段数据时，其他段的数据也能被其他线程访问。

每个map分为16个segment(数组),每个segment下面是HashEntry(链表),每一个Segment 都想的于小的hash table并且都有自己锁

jdk1.8中 数组＋单向链表＋红黑树

的实现已经抛弃了Segment分段锁机制，利用CAS+Synchronized来保证并发更新的安全，底层依然采用数组+链表+红黑树的存储结构。

CAS是什么?比较的是什么?交换的是什么?

CAS是compare and swap的缩写，即我们所说的比较交换,cas是一种基于锁的操作，而且是乐观锁, 比较的是

CAS 操作包含三个操作数 —— 内存位置（V）、预期原值（A）和新值(B)。如果内存地址里面的值和A的值是一样的，那么就将内存里面的值更新成B。CAS是通过无限循环来获取数据的，若果在第一轮循环中，a线程获取地址里面的值被b线程修改了，那么a线程需要自旋，到下次循环才有可能机会执行

问题:

①.CAS容易造成ABA问题。一个线程a将数值改成了b，接着又改成了a，此时CAS认为是没有变化，其实是已经变化过了，而这个问题的解决方案可以使用版本号标识，每操作一次version加1。在java5中，已经提供了AtomicStampedReference来解决问题。

②.CAS造成CPU利用率增加。之前说过了CAS里面是一个循环判断的过程，如果线程一直没有获取到状态，cpu资源会一直被占用。

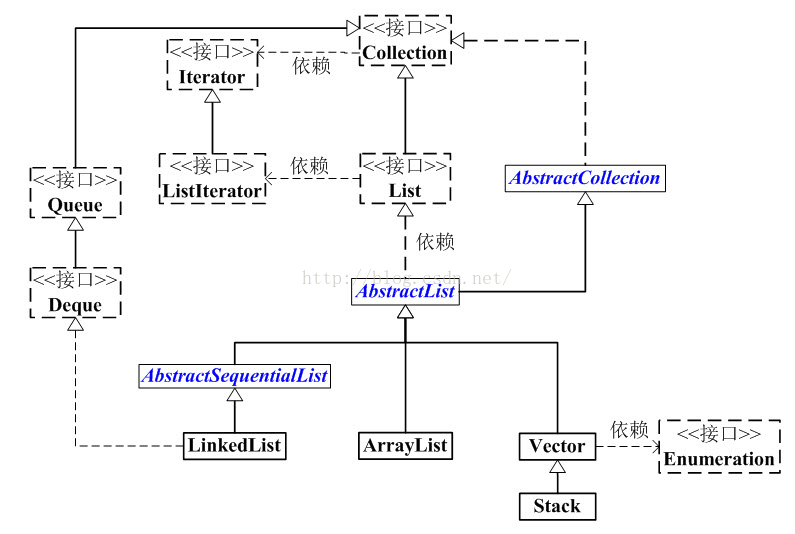
#### 使hashmap安全

1.两种方法可以实现HashMap线程安全

(1).通过collections.synchroizedMap()的接口，返回的是一个Map，这个map就是安全的，这个需要面向接口编程，返回的是一个Map

(2).重写HashMap，通过java.util.Concurrent.ConcurrentHashMap（）,把hashmap里面的方法进行拆分，​

#### 4.Java 中 ArrayList 和 Linkedlist 区别？

Arraylist底层数组查找快增删慢(索引查询)

ArrayList初始容量为0,第一次添加为10,超出则(int newCapacity = oldCapacity + (oldCapacity >> 1);)相当于扩容1.5倍但位运算快,将就数组元素copy到新数组

不是线程安全的，只能用在单线程环境下，多线程环境下可以考虑用Collections.synchronizedList(List l)函数返回一个线程安全的ArrayList类，也可以使用concurrent并发包下的CopyOnWriteArrayList类

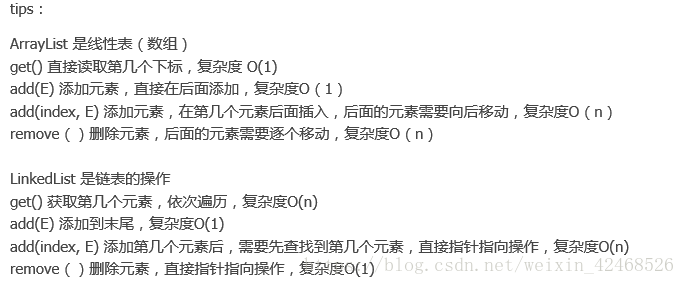
Linkedlist底层链表查找慢增删快(遍历查询) 双向链表

只有头尾插入的时候Linkedlist比Arraylist慢

总结：1、对于随机访问get和set，ArrayList优于LinkedList，因为LinkedList要移动指针。

                 对于新增和删除操作add和remove，LinedList比较占优势，因为ArrayList要移动数据。

           2、各自效率问题：



#### Arraylist和LinkedList都不安全

1.List<String>list=Collections.synchronizedList(new LinkedList<String>());

　　2、LinkedList换成ConcurrentLinkedQueue

　　3、Vector

## 数据库

### [SQL中IN和EXISTS用法的区别](https://www.cnblogs.com/clarke157/p/7912871.html)

1. in()适合B表比A表数据小的情况

2. exists()适合B表比A表数据大的情况

当A表数据与B表数据一样大时,in与exists效率差不多,可任选一个使用.

因为in不走索引,exists走索引,所以B表数据小的话走索引更慢,B表数据大的话existe可以走索引,所以A大用in,B大用exists

<https://blog.csdn.net/chadcao/article/details/6587887>

### Mysql

#### 并发数

Java应用系统部署的时候常选用的机器配置大致是2核4G和4核8G的较多一些，数据库部署的时候常选用的机器配置最低在8核16G以上，正常在16核32G，一般Java应用系统部署在**4核8G**的机器上，每秒钟抗下**500左右的并发**访问量，差不多是比较合适的，当然这个也不一定。一般**8核16G**的机器部署的MySQL数据库，每秒抗个**一两千并发**请求是没问题的，**16核32G**的机器部署的MySQL数据库而言，每秒抗个**两三千**，甚至三四千的并发请求也都是可以的，对于数据库而言，如果可以的话，最好是采用SSD固态硬盘而不是普通的机械硬盘

#### 引擎：

##### MyISAM

1. 不支持事务，但是整个操作是原子性的(事务具备四种特性：原子性、一致性、隔离性、持久性)
2. 不支持外键，支持表锁，每次锁住的是整张表，MyISAM的表锁有读锁和写锁(两个锁都是表级别)

读写锁互斥，总是写先获取锁，因此不太适合于有大量更新操作和查询操作，因为，大量的更新操作会造成查询操作很难获得读锁，从而可能永远阻塞

1. 一个MyISAM表有三个文件：索引文件，表结构文件，数据文件
2. 存储表的总行数，执行select count(\*) from table时只要简单的读出保存好的行数即可，count(\*)速度快的也仅仅是不带where条件的count
3. 采用**非聚集索引**，索引文件的**数据域存储指向数据文件的指针（不存储数据）**。辅索引与主索引基本一致，但是辅索引不用保证唯一性。
4. 支持全文索引和空间索引
5. 对于AUTO\_INCREMENT类型的字段，在MyISAM表中，可以和其他字段一起建立联合索引。

##### Innodb

1. 支持事务，支持事务的四种隔离级别;是一种具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表
2. 支持行锁和外键约束，因此可以支持写并发
3. **不存储总行数**；也就是说，执行select count(\*) from table时，InnoDB要扫描一遍整个表来计算有多少行。注意的是，当count(\*)语句包含 **where条件时，两种表的操作是一样的**。
4. 对于AUTO\_INCREMENT类型的字段，InnoDB中必须包含只有该字段的索引
5. DELETE FROM table时，InnoDB**不会重新建立表**，而是一行一行的删除
6. 一个Innodb表存储在一个文件内(共享表空间，表大小不受操作系统的限制)，也可能为多个(设置为独立表空间，表大小受操作系统限制，大小为2G)，受操作系统文件大小的限制
7. **主键索引**采用聚集索引（索引的**数据域存储数据文件本身**），**辅索引的数据域存储主键的值**；因此从**辅索引查找数据，需要先通过辅索引找到主键值，再访问主键索引**；**最好使用自增主键，防止插入数据时，为维持B+树结构，文件的大调整**。

#### 服务器优化（mysql优化配置文件）

https://www.cnblogs.com/mungerz/p/10416742.html

InnoDB：

##### InnoDB缓冲池（Buffer Pool）

如果大部分都是InnoDB表，InnoDB缓冲池或许比其他任何东西更需要内存。InnoDB缓冲池并不仅仅缓冲索引：它**还会缓存行数据、自适应hash、插入缓存、锁，以及其他内部数据结构**。InnoDB还使用缓冲池来帮助**延迟写入，这样就能合并多个写入操作**，然后一起顺序地写回。总之，InnoDB严重依赖缓冲区，你必须确认它分配了足够的内存，竟可能的大。

##### 优化MySQL的I/O行为

InnoDB**使用日志来减少提交事务时的开销**，因为日志中记录了事务，就无须在每个事务提交时把缓冲池的脏块刷新到磁盘中，事务修改的数据和索引通常会映射到表空间的随机位置，所以刷新这些变更到磁盘需要很多**随机I/O**。InnoDB假设使用的是常规磁盘（机械磁盘），随机I/O比顺序I/O要昂贵得多，因为一个I/O请求需要时间把磁头移动正确的位置，然后等待磁盘上读出需要的部分，再转到开始位置。

InnoDB使用一个后台线程智能地刷新这些变更到数据文件。这个线程可以**批量组合写入**，使得数据写入更顺序，以提高效率。实际上，**事务日志把数据文件的随机I/O转换为几乎顺序的日志文件和数据文件I/O**。把刷新操作转移到后台使查询可以更快完成，并且缓和查询高峰时I/O系统的压力。

整体的日志文件大小受控于innodb\_log\_file\_size和innodb\_log\_files\_in\_group两个参数，这对写性能非常重要。日志文件的总大小是每个文件的大小之和。默认情况下，只有两个5MB的文件，总共10MB。对高性能工作来说这太小了。至少需要几百MB或者上GB的日志文件。

了解清楚“把日志缓冲写到日志文件”和“把日志刷新到持久化存储”之间的不同是很重要的。在大部分操作系统中，**把缓冲写到日志只是简单地把数据从InnoDB的内存缓冲转移到操作系统的内存，并没有真的把数据写到持久化存储**。

日志缓冲必须被刷新到持久化存储，以确保提交的事务完全被持久化了。如果和持久化相比更在乎性能，可以修改innodb\_flush\_log\_at\_trx\_commit变量来**控制日志缓冲区刷新的频繁程度**。可能的设置如下：

0 ：把日志缓冲写到日志文件，并且每秒钟刷新一次，但是事务提交时不做任何事。

1 ：将日志缓冲写到日志文件，并且每次事务提交都刷新到持久化存储。这是默认的（并且是最安全的）设置，改设置能保证不会丢失任何已经提交的事务，除非磁盘或者操作系统是“伪”刷新的。

2 ：每次提交时把日志缓冲写到日志文件，但是并不刷新。InnoDB每秒做一次刷新。0与2最要的不同是（也是为什么2更合适），如果MySQL挂了，2不会丢失任何事务。如果整个服务器“挂了”或者断电了，则还是可能会丢失一些事务。

#### 参数配置：

<https://blog.csdn.net/miyatang/article/details/54881547?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-2.control&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-2.control>

##### 线程独享内存

###### thread\_stack线程栈信息使用内存

主要用来存放每一个线程自身的标识信息，如线程id，线程运行时基本信息等等，我们可以通过 thread\_stack 参数来设置为每一个线程栈分配多大的内存。  
Global,No Dynamic,Default 192K(32bit), 256K(32bit),  
**推荐配置：默认**

###### sort\_buffer\_size排序使用内存

MySQL 用此内存区域进行排序操作(filesort)，完成客户端的排序请求。当我们设置的排序区缓存大小无法满足排序实际所需内存的时候，MySQL会将数据写入磁盘文件来完成排序。由于磁盘和内存的读写性能完全不在一个数量级，所以sort\_buffer\_size参数对排序操作的性能影响绝对不可小视。  
使用确认：  
可以通过查询计划中的Extra列的值为Using file-sort来证实使用了和这个缓冲区。  
>explain select \* from user1;  
Global Session,Dynamic,Default 2M(32bit), 2M(32bit),  
**推荐配置：8M（内存足够的情况下），默认（内存紧张的情况）**优化建议：一种说法是增大可以提高order by,group by性能，防止数据写入磁盘占用IO资源，还有一种说法是不推荐增加这个缓冲区的大小，理由是当值太大时可能会降低查询的执行速度。目前我没有实验证实。

###### join\_buffer\_size Join操作使用内存

当Join Buffer太小，MySQL 不会将该Buffer存入磁盘文件，而是先将Join Buffer中的结果集与需要Join的表进行Join操作，然后清空Join Buffer中的数据，继续将剩余的结果集写入此Buffer中，如此往复。这势必会造成被驱动表需要被多次读取，成倍增加IO访问，降低效率。  
什么时候会用到？  
当**查询必须连接两个表（或多个）的数据集并且不能使用索引**时，这个缓冲区会被用到。这个缓冲区专门为每个线程的无索引链接操作准备的。  
使用确认：  
可以通过查询计划中的Extra列的值为Using join bufer来证实使用了和这个缓冲区。  
>explain select \* from user1;  
+------+-------------+-------+-------+---------------+------+---------+------+------+-------------+  
| id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |  
+------+-------------+-------+-------+---------------+------+---------+------+------+-------------+  
| 1 | SIMPLE | user1 | index | NULL | name | 78 | NULL | 3 | Using index |  
+------+-------------+-------+-------+---------------+------+---------+------+------+-------------+  
Global Session,Dynamic,Default 128K 各版本平台最大值不一样  
**推荐配置：8M（内存足够的情况下），默认（内存紧张的情况）**优化建议：有一种说法是增加这个缓冲区的大小不会加快全连接操作的速度。目前我没有实验证实。

###### read\_buffer\_size 顺序读取数据缓冲区使用内存

这部分内存主要用于当需要顺序读取数据的时候，如无法使用索引的情况下的全表扫描，全索引扫描等。在这种时候，MySQL按照数据的存储顺序依次读取数据块，每次读取的数据快首先会暂存在read\_buffer\_size中，当buffer空间被写满或者全部数据读取结束后，再将buffer中的数据返回给上层调用者，以提高效率。  
Global Session,Dynamic,Default 128K  
推荐配置：4M/8M

###### read\_rnd\_buffer\_size 随机读取数据缓冲区使用内存

和顺序读取相反，当MySQL进行非顺序读取（随机读取）数据块的时候，会利用这个缓冲区暂存读取的数据。如根据索引信息读取表数据，根据排序后的结果集与表进行Join等等。总的来说，就是当数据块的读取需要满足一定的顺序的情况下，MySQL就需要产生随机读取，进而使用到read\_rnd\_buffer\_size 参数所设置的内存缓冲区。  
Global Session,Dynamic,Default 256K  
推荐配置：8M

###### bulk\_insert\_buffer\_size批量插入暂存使用内存

当我们使用如 insert … values(…),(…),(…)… 的方式进行批量插入的时候，MySQL会先将提交的数据放如一个缓存空间中，当该缓存空间被写满或者提交完所有数据之后，MySQL才会一次性将该缓存空间中的数据写入数据库并清空缓存。此外，当我们进行 LOAD DATA INFILE操作来将文本文件中的数据Load进数据库的时候，同样会使用到此缓冲区。  
Global Session,Dynamic,Default 8M  
推荐配置：默认 8M

###### tmp\_table\_size临时表使用内存

当我们进行一些特殊操作如需要使用临时表才能完成的Order By,Group By 等等。当我们的临时表较小（小于tmp\_table\_size 参数所设置的大小）的时候，MySQL会将临时表创建成内存临时表，  
只有当tmp\_table\_size所设置的大小无法装下整个临时表的时候，MySQL才会将该表创建成MyISAM存储引擎的表存放在磁盘上。不过，当另一个系统参数 max\_heap\_table\_size 的大小还小于 tmp\_table\_size 的时候，MySQL将使用 max\_heap\_table\_size 参数所设置大小作为最大的内存临时表大小，而忽略tmp\_table\_size 所设置的值。而且 **tmp\_table\_size** 参数从 MySQL 5.1.2 才开始有，之前一直使用 **max\_heap\_table\_size。谁小谁生效**.另外还有一个参数max\_tmp\_tables,没有使用  
tmp\_table\_size  
Global Session,Dynamic,Default 16M  
推荐配置：64M（show global status like 'create%tables';调整该值直到created\_tmp\_disk\_tables / created\_tmp\_tables \* 100% <= 25%）

##### 全局共享内存

###### key\_buffer\_size MyISAM索引缓存 Key Buffer

MyISAM 索引缓存将MyISAM表的索引信息（.MYI文件）缓存在内存中，以提高其访问性能。这个缓存可以说是影响MyISAM存储引擎性能的最重要因素之一了，通过 key\_buffere\_size 设置可以使用的最大内存空间。  
注意：即使运行一个全部采用innodb的模式，仍需要定义一个索引码缓冲区，因为MYSQL元信息与MyISAM定义相同。  
Global ,Dynamic,Default 8M  
推荐配置：默认 8M  
如何确认key\_buffer\_size不够用？  
使用show full proceslist的State列中，值Repairing the keycache是一个明显的指标，它指出当前索引码缓冲区大小不足以执行当前运行的SQL语句。这将导致额外的磁盘I/O开销。

比例key\_reads / key\_read\_requests应该尽可能的低，至少是1:100，1:1000更好(上述状态值可以使用SHOW STATUS LIKE ‘key\_read%’获得)

###### query\_cache\_size查询缓存 Query Cache

MySQL针对Query Cache设计了多个query\_cache\_type（0,1,2）值  
和两个Query Hint：SQL\_CACHE和SQL\_NO\_CACHE。

1. 当query\_cache\_type设置为0（或者 OFF）的时候不使用Query Cache，
2. 当设置为1（或者 ON）的时候，当且仅当Query中使用了SQL\_NO\_CACHE 的时候MySQL会忽略Query Cache，  
   3. 当query\_cache\_type设置为2（或者DEMAND）的时候，当且仅当Query中使用了SQL\_CACHE提示之后，MySQL才会针对该Query使用Query Cache。

推荐配置：**16M**  
如何确定系统query cache的情况？  
show global status like 'Qcache%';或者  
select \* from information\_schema.GLOBAL\_STATUS where VARIABLE\_NAME like 'Qcache%';  
公式：  
（Qcache\_hits/Qcache\_hits+Com\_select+1)\*100来确定查询缓存的有效性

###### table\_open\_cache 表缓存 Table Cache

表缓存区主要用来缓存表文件的文件句柄信息,设置的是可以缓存的表文件句柄信息的数目，而不是内存空间的大小。

推荐配置：根据内存配置4G 2048 大于最大Opened\_tables

调优：

show variables like 'table\_open\_cache';

show global status like 'open%\_tables';

其中Open\_tables是当前正在打开表的数量，Opened\_tables是所有已经打开表的数量。  
如果Open\_tables的值已经接近table\_cache的值，且Opened\_tables还在不断变大，则说明mysql正在将缓存的表释放以容纳新的表，此时可能需要加大table\_cache的值。对于大多数情况，比较适合的值：  
Open\_tables / Opened\_tables >= 0.85  
Open\_tables / table\_cache <= 0.95  
如果对此参数的把握不是很准，VPS管理百科给出一个很保守的设置建议：把MySQL数据库放在生产环境中试运行一段时间，然后把参数的值调整得比Opened\_tables的数值大一些，并且保证在比较高负载的极端条件下依然比Opened\_tables略大。  
在mysql默认安装情况下，table\_cache的值在2G内存以下的机器中的值默认时256到 512，如果机器有4G内存,则默认这个值是2048，

###### table\_definition\_cache 表定义信息缓存

存放表定义信息，设置可以缓存的表的数量

推荐配置：根据内存配置4G 2048 和Table Cache一样即可

###### binlog\_cache\_size 二进制日志缓冲区

二进制日志缓冲区主要用来缓存由于各种数据变更操做所产生的 Binary Log 信息。为了提高系统的性能，MySQL 并不是每次都是将二进制日志直接写入 Log File，而是先将信息写入 Binlog Buffer 中，  
当满足某些特定的条件（如 sync\_binlog参数设置）之后再一次写入 Log File 中。我们可以通过 binlog\_cache\_size 来设置其可以使用的内存大小，同时通过 max\_binlog\_cache\_size 限制其最大大小  
（当单个事务过大的时候 MySQL 会申请更多的内存）。当所需内存大于 max\_binlog\_cache\_size 参数设置的时候，MySQL 会报错：“Multi-statement transaction required more than ‘max\_binlog\_cache\_size’ bytes of storage”。  
Global,Dynamic,Default 32K  
推荐配置：2M

###### innodb\_log\_buffer\_size InnoDB 日志缓冲区

这是 InnoDB 存储引擎的事务日志所使用的缓冲区。类似于 Binlog Buffer，InnoDB 在写事务日志的时候，为了提高性能，也是先将信息写入 Innofb Log Buffer 中，当满足 innodb\_flush\_log\_trx\_commit 参数所设置的相应条件（或者日志缓冲区写满）之后，才会将日志写到文件（或者同步到磁盘）中。可以通过 innodb\_log\_buffer\_size 参数设置其可以使用的最大内存空间。  
注：innodb\_flush\_log\_trx\_commit 参数对 InnoDB Log 的写入性能有非常关键的影响。该参数可以设置为0，1，2，解释如下：  
  
0：log buffer中的数据将以**每秒一次**的频率写入到log file中，且同时会**进行文件系统到磁盘的同步操作**，但是每个事务的commit并**不会触发**任何log buffer 到log file的刷新或者文件系统到磁盘的**刷新操作**；  
1：在**每次事务提交**的时候将log buffer 中的数据**都会写入到log file**，同时也会**触发**文件系统到磁盘的**同步**；（**默认，最安全**）  
2：事务提交**会触发**log buffer 到log file的**刷新**，但并**不会触发磁盘文件系统到磁盘的同步**。此外，**每秒会有一次文件系统到磁盘同步操作**。  
此外，MySQL文档中还提到，这几种设置中的每秒同步一次的机制，可能并不会完全确保非常准确的每秒就一定会发生同步，还取决于进程调度的问题。实际上，InnoDB 能否真正满足此参数所设置值代表的意义正常 Recovery 还是受到了不同 OS 下文件系统以及磁盘本身的限制，可能有些时候在并没有真正完成磁盘同步的情况下也会告诉 mysqld 已经完成了磁盘同步。  
  
Global,Dynamic,Default 8M  
推荐配置：8M 默认

#### 数据库索引:

建立索引的原理过程，数据结构，B+tree能画么

#### 性能优化:

1、当只要一行数据时使用 limit 1

查询时如果已知会得到一条数据，这种情况下加上 limit 1 会增加性能。因为 mysql 数据库引

擎会在找到一条结果停止搜索，而不是继续查询下一条是否符合标准直到所有记录查询完毕。

2、选择正确的数据库引擎

Mysql 中有两个引擎 MyISAM 和 InnoDB，每个引擎有利有弊。

MyISAM 适用于一些大量查询的应用，但对于有大量写功能的应用不是很好。甚至你只需要 update 一个字段整个表都会被锁起来。而别的进程就算是读操作也不行要等到当前 update 操作完成之后才能继续进行。另外，MyISAM 对于 select count(\*)这类操作是超级快的。

InnoDB 的趋势会是一个非常复杂的存储引擎，对于一些小的应用会比 MyISAM 还慢，但是支

持“行锁”，所以在写操作比较多的时候会比较优秀。并且，它支持很多的高级应用，例如：事物。

3. 用 not exists 代替 not in

Not exists 用到了连接能够发挥已经建立好的索引的作用，not in 不能使用索引。Not in 是最慢的方式要同每条记录比较，在数据量比较大的操作中不建议使用这种方式。

4. 对操作符的优化，尽量不采用不利于索引的操作符

如：in not in is null is not null <> 等

某个字段总要拿来搜索，为其建立索引：

Mysql 中可以利用 alter table 语句来为表中的字段添加索引，语法为：alter table 表明 add index (字段名)；

MySQL 单表数据量大于 2000 万行，性能会明显下降，单表行数超过 500 万行或者单表容量超过 2GB，才推荐进行分库分表

### 你所了解的数据库优化都有哪些？

问题分析  
考官主要是对数据库优化方面的考核，一般数据库优化分为性能和应用方面的，如你了解sql优化吗；百万数据怎么优化等

核心答案讲解

、根据服务层面：配置mysql性能优化参数；

、从系统层面增强mysql的性能：优化数据表结构、字段类型、字段索引、分表，分库、读写分离等等。

、从数据库层面增强性能：优化SQL语句，合理使用字段索引。

、从代码层面增强性能：使用缓存和NoSQL数据库方式存储，如MongoDB/Memcached/Redis来缓解高并发下数据库查询的压力。  
（5）、减少数据库操作次数，尽量使用数据库访问驱动的批处理方法。  
（6）、不常使用的数据迁移备份，避免每次都在海量数据中去检索。  
（7）、提升数据库服务器硬件配置，或者搭建数据库集群。  
（8）、编程手段防止SQL注入：使用JDBC PreparedStatement按位插入或查询；正则表达式过滤（非法字符串过滤）；

问题扩展  
Sql优化（4-5条即可）：  
1）应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。  
2）应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：  
select id from t where num is null  
可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：  
select id from t where num=0  
3）很多时候用 exists 代替 in 是一个好的选择  
4）用Where子句替换HAVING 子句 因为HAVING 只会在检索出所有记录之后才对结果集进行过滤  
5）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  
索引  
索引概念：对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息。如果我们把一个表的内容认为是一本字典，那索引就相当于字典的目录  
索引类型：  
Oracle:  
逻辑上：Single column 单行索引  
Concatenated 多行索引  
Unique 唯一索引  
NonUnique 非唯一索引  
Function-based函数索引  
Domain 域索引  
物理上：  
Partitioned 分区索引  
NonPartitioned 非分区索引  
B-tree：  
Normal 正常型B树  
Rever Key 反转型B树  
Bitmap 位图索引

MySQL索引分为普通索引、唯一索引、主键索引、组合索引、全文索引  
何时使用索引

主键，unique字段；

和其他表做连接的字段需要加索引；

在where里使用＞，≥，＝，＜，≤，is null和between等字段；

使用不以通配符开始的like，where A like ‘China%’；  
（5）聚集函数MIN()，MAX()中的字段；  
（6）order by和group by字段；  
索引何时失效

组合索引未使用最左前缀，例如组合索引（A，B），where B=b不会使用索引；

like未使用最左前缀，where A like ‘%China’；

搜索一个索引而在另一个索引上做order by，where A=a order by B，只使用A上的索引，因为查询只使用一个索引 ；

or会使索引失效。如果查询字段相同，也可以使用索引。例如where A=a1 or A=a2（生效），where A=a or B=b（失效）  
（5）如果列类型是字符串，要使用引号。例如where A=’China’，否则索引失效（会进行类型转换）；  
（6）在索引列上的操作，函数（upper()等）、or、！=(<>)、not in等；

结合项目中使用  
1.常用但不经常修改的字段建索引(譬如商品表的商品名称等字段)，达到检索速度增快，用户体验度增高的目的  
2.用mycat进行分库分表  
垂直拆分是基于数据库中的”列”进行，某个表字段较多，可以新建一张扩展表，将不经常用或字段长度较大的字段拆分出去到扩展表中。例如用户表，在字段很多的情况下（例如一个大表有100多个字段），通过”大表拆小表”，更便于开发与维护，也能避免跨页问题  
水平分表  
水平切分分为库内分表和分库分表，是根据表内数据内在的逻辑关系，将同一个表按不同的条件分散到多个数据库或多个表中，每个表中只包含一部分数据，从而使得单个表的数据量变小，达到分布式的效果（如订单表）

参考文档：

[https://blog.csdn.net/yzllz001/article/details/54848513/](https://blog.csdn.net/yzllz001/article/details/54848513/" \t "https://www.showdoc.cc/_blank)  
[https://blog.csdn.net/weixin\_39420024/article/details/80040549](https://blog.csdn.net/weixin_39420024/article/details/80040549" \t "https://www.showdoc.cc/_blank)  
[https://blog.csdn.net/boonya/article/details/60962774](https://blog.csdn.net/boonya/article/details/60962774" \t "https://www.showdoc.cc/_blank)  
[https://www.cnblogs.com/sunny3096/p/8595058.html](https://www.cnblogs.com/sunny3096/p/8595058.html" \t "https://www.showdoc.cc/_blank)  
[https://www.cnblogs.com/butterfly100/p/9034281.html](https://www.cnblogs.com/butterfly100/p/9034281.html" \t "https://www.showdoc.cc/_blank)

### Innodb和MyIASM

首先：

1.简单介绍这两种引擎，以及该如何去选择。  
2.这两种引擎所使用的数据结构是什么。

1.

a.Innodb引擎，Innodb引擎提供了对数据库ACID事务的支持。并且还提供了行级锁和外键的约束。它的设计的目标就是处理大数据容量的数据库系统。它本身实际上是基于Mysql后台的完整的系统。Mysql运行的时候，Innodb会在内存中建立缓冲池，用于缓冲数据和索引。但是，该引擎是不支持全文搜索的。同时，启动也比较的慢，它是不会保存表的行数的。当进行Select count(\*) from table指令的时候，需要进行扫描全表。所以当需要使用数据库的事务时，该引擎就是首选。由于锁的粒度小，写操作是不会锁定全表的。所以在并发度较高的场景下使用会提升效率的。

b.MyIASM引擎，它是MySql的默认引擎(5.5以前,之后默认引擎是innodb)，但不提供事务的支持，也不支持行级锁和外键。因此当执行Insert插入和Update更新语句时，即执行写操作的时候需要锁定这个表。所以会导致效率会降低。不过和Innodb不同的是，MyIASM引擎是保存了表的行数，于是当进行Select count(\*) from table语句时，可以直接的读取已经保存的值而不需要进行扫描全表。所以，如果表的读操作远远多于写操作时，并且不需要事务的支持的。可以将MyIASM作为数据库引擎的首先。

补充2点：

c.大容量的数据集时趋向于选择Innodb。因为它支持事务处理和故障的恢复。Innodb可以利用数据日志来进行数据的恢复。主键的查询在Innodb也是比较快的。

d.大批量的插入语句时（这里是INSERT语句）在MyIASM引擎中执行的比较的快，但是UPDATE语句在Innodb下执行的会比较的快，尤其是在并发量大的时候。

2.两种引擎所使用的索引的数据结构是什么？

答案:都是B+树!

MyIASM引擎，B+树的数据结构中存储的内容实际上是实际数据的地址值。也就是说它的索引和实际数据是分开的，只不过使用索引指向了实际数据。这种索引的模式被称为非聚集索引。

Innodb引擎的索引的数据结构也是B+树，只不过数据结构中存储的都是实际的数据，这种索引有被称为聚集索引。

参考：[http://blog.csdn.net/lulei1217/article/details/50954232](http://blog.csdn.net/lulei1217/article/details/50954232" \t "https://www.cnblogs.com/xiaohaillong/p/_blank)

### 聚集索引(相当于拼音目录对应的A-Z的字顺序)

只有叶子节点有数据---->整行数据

数据行的物理顺序与列值（一般是主键的那一列）的逻辑顺序相同，一个表中只能拥有一个聚集索引,索引的叶子节点就是对应的数据节点,可以直接获得对应的所有列的数据(比如id=1的整行数据),最好还是在创建表的时候添加聚集索引，由于聚集索引的物理顺序上的特殊性，因此如果再在上面创建索引的时候会根据索引列的排序移动全部数据行上面的顺序，会非常地耗费时间以及性能

### 非聚集索引:(像新华字典的偏旁字典)

叶子节点上是索引节点--->指针指向数据块--->当前列数据与主键--->二次查询

该索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同，除了聚集索引以外的索引都是非聚集索引,又分成普通索引，唯一索引，全文索引,**非聚集索引叶节点仍然是索引节点**，只是有一个指针指向对应的数据块，此如果使用非聚集索引查询，而查询列中包含了其他该索引没有覆盖的列，那么他还要进行**第二次的查询**，查询节点上对应的数据行的数据(**即只有该索引列和聚集索引列数据**),因为Innodb二级索引(即聚集索引)存储的是主键，所以通过索引查找时，**第一次查询是通过二级索引找到主键值，第二次查询是通过主键在聚集索引找到对应的行位置**

## 设计模式

### 单例模式

饿汉式 对象加载时创建对象 静态成员变量处创建对象

懒汉式 调用方法时创建对象 双重锁 特点延迟加载 应用场景配置文件

双重锁：

两个线程，第一次判空 ，都进来，第一个线程获取锁执行完，第二个线程获取锁再判空，对象存在，不用再创建对象一次。

### 工厂模式

抽象工厂-->实体工厂--->抽象产品---->实体产品

### 模板方法

异步导入用到，抽象类解析excel，具体实现类只需继承抽象类完成逻辑部分

### 代理模式

导入导出记录日志采用aop

### 原型模式

对于经常需要new的对象，利用已有的一个原型对象，快速地生成和原型对象一样的实例（利用克隆），轻量级的对象可以使用new，其他对象可以使用clone（如构造函数中截取字符串）

### 创建型模式

### 结构型模式

### 行为型模式

## 负载均衡算法

### 静态均衡算法：

#### 1、轮询法

将请求按顺序轮流地分配到每个节点上，不关心每个节点实际的连接数和当前的系统负载。

优点：简单高效，易于水平扩展，每个节点满足字面意义上的均衡；

缺点：没有考虑机器的性能问题，根据木桶最短木板理论，集群性能瓶颈更多的会受性能差的服务器影响

#### 2、随机法

将请求随机分配到各个节点。由概率统计理论得知，随着客户端调用服务端的次数增多，其实际效果越来越接近于平均分配，也就是轮询的结果。

#### 3、源地址哈希法

    源地址哈希的思想是根据客户端的IP地址，通过哈希函数计算得到一个数值，用该数值对服务器节点数进行取模，得到的结果便是要访问节点序号。采用源地址哈希法进行负载均衡，同一IP地址的客户端，当后端服务器列表不变时，它每次都会落到到同一台服务器进行访问。

优点：相同的IP每次落在同一个节点，可以人为干预客户端请求方向，例如灰度发布；

缺点：如果某个节点出现故障，会**导致这个节点上的客户端无法使用，无法保证高可用**。当某一用户成为热点用户，那么会有巨大的流量涌向这个节点，导致冷热分布不均衡，无法有效利用起集群的性能。所以当热点事件出现时，一般会将源地址哈希法切换成轮询法。

#### 4、加权轮询法

    给配置高、负载低的机器配置更高的权重，让其处理更多的请，加权轮询能很好地按照请求顺序且按照权重分配到后端。

加权轮询算法要生成一个服务器序列，该序列中包含n个服务器。n是所有服务器的权重之和。在该序列中，每个服务器的出现的次数，等于其权重值。并且，生成的序列中，服务器的分布应该尽可能的均匀。比如序列{a, a, a, a, a, b, c}中，前五个请求都会分配给服务器a，这就是一种不均匀的分配方法，更好的序列应该是：{a, a, b, a, c, a, a}。

优点：可以将不同机器的性能问题纳入到考量范围，集群性能最优最大化；

缺点：生产环境复杂多变，服务器抗压能力也无法精确估算，静态算法导致无法实时动态调整节点权重，只能粗糙优化。

#### 5、加权随机法

与加权轮询法一样，加权随机法也根据后端机器的配置，系统的负载分配不同的权重。不同的是，它是按照权重随机请求后端服务器，而非顺序。

#### 6、键值范围法

根据键的范围进行负债，比如0到10万的用户请求走第一个节点服务器，10万到20万的用户请求走第二个节点服务器……以此类推。

优点：容易水平扩展，随着用户量增加，可以增加节点而不影响旧数据；

缺点：容易负债不均衡，比如新注册的用户活跃度高，旧用户活跃度低，那么压力就全在新增的服务节点上，旧服务节点性能浪费。而且也容易单点故障，无法满足高可用。

### 动态均衡算法：

#### 1、最小连接数法

根据每个节点当前的连接情况，动态地选取其中**当前积压连接数最少的一个节点处理当前请求**，尽可能地提高后端服务的利用效率，将请求合理地分流到每一台服务器。俗称闲的人不能闲着，大家一起动起来。

优点：动态，根据节点状况实时变化；

缺点：提高了复杂度，每次连接断开需要进行计数；

实现：将连接数的倒数当权重值。

#### 2、最快响应速度法

根据请求的响应时间，来动态调整每个节点的权重，将响应速度快的服务节点分配更多的请求。

优点：动态，实时变化，控制的粒度更细，跟灵敏；

缺点：复杂度更高，每次需要计算请求的响应速度；

实现：可以根据响应时间进行打分，计算权重。

#### 3、观察模式法

观察者模式是综合了最小连接数和最快响应度，同时考量这两个指标数，进行一个权重的分配

## 多线程

多核cpu可以同时执行多个线程,单核cpu同一时间只会执行一个线程,cpu会根据时间片切换线程,并发由此产生,例如io操作比较耗时间这时cpu可以切换到别的线程执行,可以理解为一个进程便相当于一个单 CPU 操作系统，而线程便是这个系统中运行的多个任务。

CPU切换线程并不会管你线程是否将代码执行完，而是和分给线程的时间片是否到期有关，时间片到期了就会切换线程，并发也就由此产生了

### 六种状态：

**新建状态**（New）：当线程对象对创建后，即进入了新建状态，如：Thread t = new MyThread();但是还没有调用start方法

**就绪状态**（Runnable）：当调用线程对象的start()方法（t.start();），线程即进入就绪状态。处于就绪状态的线程，只是说明此线程已经做好了准备，随时等待CPU调度执行，并不是说执行了t.start()此线程立即就会执行；

**阻塞状态**（Blocked）：

Lock 或者synchronize 关键字产生的状态，**还未获取锁**

处于运行状态中的线程由于某种原因，暂时放弃对CPU的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才有机会再次被CPU调用以进入到运行状态。根据阻塞产生的原因不同，阻塞状态又可以分为三种：

1.等待阻塞：运行状态中的线程执行wait()（释放锁）方法，使本线程进入到等待阻塞状态；

2.同步阻塞 -- 线程在获取synchronized同步锁失败(因为锁被其它线程所占用)，它会进入同步阻塞状态；

3.其他阻塞 -- 通过调用线程的sleep()（不释放锁）或join()或发出了I/O请求时，线程会进入到阻塞状态。当sleep()状态超时、join()等待线程终止或者超时、或者I/O处理完毕时，线程重新转入就绪状态。

**等待（WAITING）**

**获取到锁了，释放锁，等待**

一个线程进入了锁，但是需要等待其他线程执行某些操作。时间不确定  
当wait，join，park方法调用时，进入waiting状态。**前提是这个线程已经拥有锁了**。

**TIMED\_WAITING**

一个线程进入了锁，但是需要等待其他线程执行某些操作。时间确定  
通过sleep或wait timeout方法进入的**限期等待的状态**

**终止（terminated）**：线程执行完了或者因异常退出了run()方法，该线程结束生命周期。

### 进程与线程

1.进程

​ 是指一个内存中运行的应用程序，每个进程都有一个独立的内存空间，一个应用程序可以同时运行多个进程；进程也是程序的一次执行过程，是系统运行程序的基本单位；系统运行一个程序即是一个进程从创建、运行到消亡的过程。

2.线程

​ 进程内部的一个独立执行单元；一个进程可以同时并发的运行多个线程，可以理解为一个进程便相当于一个单 CPU 操作系统，而线程便是这个系统中运行的多个任务。

进程与线程的区别

​ 进程：有独立的内存空间，进程中的数据存放空间（堆空间和栈空间）是独立的，至少有一个线程。

​ 线程：堆空间是共享的，栈空间是独立的，线程消耗的资源比进程小的多。

### 主线程和自定义线程执行顺序:

谁抢到谁执行,可以调整优先级,子线程初始优先级与父线程相同。不过主线程先启动占用了cpu资源，因此主线程总是优于子线程。然而，其实设置了优先级，也无法保障线程的执行次序。只不过，优先级高的线程获取CPU资源的概率较大，优先级低的并非没机会执行。 线程的优先级用1-10之间的整数表示，数值越大优先级越高，默认的优先级为5,可以用join()方法.

### Start和run方法有什么区别

Start是新开启一个线程,run只是调用了run方法在原线程执行.

1.start（）方法来启动线程，真正实现了多线程运行。这时无需等待run方法体代码执行完毕，可以直接继续执行下面的代码；通过调用Thread类的start()方法来启动一个线程， 这时此线程是处于就绪状态， 并没有运行。 然后通过此Thread类调用方法run()来完成其运行操作的， 这里方法run()称为线程体，它包含了要执行的这个线程的内容， Run方法运行结束， 此线程终止。然后CPU再调度其它线程。

2.run（）方法当作普通方法的方式调用。程序还是要顺序执行，要等待run方法体执行完毕后，才可继续执行下面的代码； 程序中只有主线程——这一个线程， 其程序执行路径还是只有一条， 这样就没有达到写线程的目的。

记住：多线程就是分时利用CPU，宏观上让所有线程一起执行 ，也叫并发

### [java中创建线程的四种方法以及区别](http://www.cnblogs.com/3s540/p/7172146.html)

1）继承Thread类创建线程

2）实现Runnable接口创建线程

3）使用Callable和Future创建线程

4）使用线程池例如用Executor框架

### [Java线程池的四种创建方式](https://www.cnblogs.com/lanseyitai1224/p/7895652.html)

Java通过Executors提供四种线程池，分别为：  
newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。

newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

<https://blog.csdn.net/qq_33453910/article/details/81413285>

### **[线程池的工作队列及新线程的流程顺序](https://www.cnblogs.com/liumz0323/p/11287037.html)**

New ThreadPoolExecutor(corePoolSize, maxPoolSize , keepAliveTime ,timeUnit, workQueue,threadFactory,rejectMethod )

新线程加入:

1. Running 的线程 小于 corePoolSize ，直接创建新的线程在Pool执行

2. Running 的线程  等于corePoolSize ，则任务加入工作队列

3.Running 的线程  等于corePoolSize，工作队列已满，则加入 大于corePoolSize 小于 maxPoolSize 线程

4. 全部满，执行拒绝策略

#### 三种类型:

https://www.cnblogs.com/liumz0323/p/11287037.html

1. 直接提交

例如：Executors.*newCachedThreadPool*();

new ThreadPoolExecutor(0, Integer.*MAX\_VALUE*,  
 60L, TimeUnit.*SECONDS*,  
 new SynchronousQueue<Runnable>());

当有新任务到来，则插入到SynchronousQueue中，在池中寻找可用线程来执行，若有可以线程则执行，若没有可用线程则创建一个线程来执行该任务；超过存活时间销毁。

SynchronousQueue作用：此策略可以避免在处理可能具有内部依赖性的请求集时出现锁

如果你的任务**A1，A2有内部关联**，A1需要先运行，那么先提交A1，再提交A2，当使用SynchronousQueue我们可以保证，A1必定先被执行，在A1没有被执行前，A2不可能添加入queue中，实现生产者，消费者模式。

1. 无界队列：

例如：Executors.*newFixedThreadPool*(3);

new ThreadPoolExecutor(nThreads, nThreads,  
 0L, TimeUnit.*MILLISECONDS*,  
 new LinkedBlockingQueue<Runnable>());

使用无界队列（例如，不具有预定义容量的 LinkedBlockingQueue）将导致在所有corePoolSize 线程都忙时，新任务在队列中等待。（因此，maximumPoolSize的值也就无效了。）当每个任务完全独立于其他任务，即**任务执行互不影响**时，适合于使用无界队列；例如，在 Web页服务器中。这种排队可用于处理瞬态突发请求，当命令以超过队列所能处理的平均数连续到达时，此策略允许无界线程具有增长的可能性。

1. 有界队列（复杂不推荐）：

当使用有限的 maximumPoolSizes时，有界队列（如 ArrayBlockingQueue）有助于防止资源耗尽，但是可能较难调整和控制。队列大小和最大池大小可能需要相互折衷：使用大型队列和小型池可以最大限度地降低 CPU 使用率、操作系统资源和上下文切换开销，但是可能导致人工降低吞吐量。如果任务频繁阻塞（例如，如果它们是 I/O边界），则系统可能为超过您许可的更多线程安排时间。使用小型队列通常要求较大的池大小，CPU使用率较高，但是可能遇到不可接受的调度开销，这样也会降低吞吐量。

### 使用线程池的优点

1.重用线程池的线程，避免因为线程的创建和销毁锁带来的性能开销

2.有效控制线程池的最大并发数，避免大量的线程之间因抢占系统资源而阻塞

3.能够对线程进行简单的管理，并提供一下特定的操作如：可以提供定时、定期、单线程、并发数控制等功能

### 拒绝策略：

#### AbortPolicy

丢弃任务并抛出RejectedExecutionException异常

#### DiscardPolicy

丢弃任务，但是不抛出异常

#### DiscardOldestPolicy

丢弃队列最前面的任务，然后重新提交被拒绝的任务

#### CallerRunsPolicy

如果任务被拒绝了，则由调用线程（提交任务的线程）直接执行此任务

### 实现Runnable接口比继承Thread类所具有的优势：

1.适合多个相同的程序代码的线程去共享同一个资源。

2.可以避免java中的单继承的局限性。

3.增加程序的健壮性，实现解耦操作，代码可以被多个线程共享，代码和数据独立。

4.线程池只能放入实现Runable或callable类线程，不能直接放入继承Thread的类

### 什么是线程安全:

如果有多个线程在同时运行，而这些线程可能会同时运行这段代码。程序每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的，反之则是线程不安全的。

### 死锁

例子中: 比如线程一拿到lock锁而线程二拿到this锁,线程一需要拿到this锁才能执行方法,线程二需要拿到lock锁才能执行方法内同步代码块因此形成死锁

多线程死锁：同步中嵌套同步,导致锁无法释放。

死锁解决办法：不要在同步中嵌套同步

### 守护线程

setDeamon(true)

当只有守护线程时 虚拟机就会自动退出

### Join

等待当前线程结束

### Yield

释放锁重新竞正锁

### wait()、notify()

不要在Thread实例上使用wait( ) notify() 方法 因为线程结束会自动唤醒等待当前线程的所有线程(即调用notifyAll ())

wait()、notify()、notifyAll()是三个定义在Object类里的方法，可以用来控制线程的状态。

1.wait 方法会使持有该对象的线程把该对象的控制权交出去，然后处于等待状态。

2.notify 方法会通知某个正在等待这个对象的控制权的线程继续运行。

3.notifyAll 方法会通知所有正在等待这个对象的控制权的线程继续运行。

### wait与sleep区别

1.对于sleep()方法，首先要知道该方法是属于Thread类中的。而wait()方法，则是属于Object类中的。

2.sleep()方法导致了程序暂停执行指定的时间，让出cpu给其他线程，但是他的监控状态依然保持者，当指定的时间到了又会自动恢复运行状态。

wait()是把控制权交出去，然后进入等待此对象的等待锁定池处于等待状态，只有针对此对象调用notify()方法后本线程才进入对象锁定池准备获取对象锁进入运行状态。

3.在调用sleep()方法的过程中，线程不会释放对象锁。而当调用wait()方法的时候，线程会放弃对象锁。

### 线程停止方式

结束线程有以下三种方法：

1. 设置退出标志，使线程正常退出。
2. 使用interrupt()方法中断线程。
3. 使用stop方法强行终止线程（不推荐使用Thread.stop, 这种终止线程运行的方法已经被废弃，使用它们是极端不安全的！可能操作一半被终止）
4. interrupt()是设置一个中断标志，线程仍会继续运行 代表走完就结束 但是这仅仅是会通知到被终止的线程“你该停止运行了”，被终止的线程自身拥有决定权（决定是否、以及何时停止） Thread.sleep()阻塞过程中能够检测到中断标记，而抛出中断异常。 如果线程是个死循环会走完逻辑就跳出
5. 如果resumed(继续执行)在suspend(挂起)之前可能发生死锁

### 线程并发的3个特性

原子性：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行

可见性：当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值

有序性：程序执行的顺序按照代码的先后顺序执行

小结:要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能会导致程序运行不正确。

### Java内存模型(JMM)

<https://www.hollischuang.com/archives/2550>

### Synchronized

synchronized可以保证并发的原子性，可见性，有序性

JMM关于synchronized的两条规定：

线程解锁前（退出同步代码块时）：必须把自己工作内存中共享变量的最新值刷新到主内存中

线程加锁时（进入同步代码块时）：将清空本地内存中共享变量的值，从而使用共享变量时需要从主内存中重新读取最新的值（加锁与解锁是同一把锁）

#### synchronized实现可见性的过程:

1.获得互斥锁（同步获取锁）

2.清空本地内存

3.从主内存拷贝变量的最新副本到本地内存

4.执行代码

5.将更改后的共享变量的值刷新到主内存

6.释放互斥锁

#### synchronized同步原理:

synchronized的同步可以解决原子性、可见性和有序性的问题

Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：

1.普通同步方法，锁是当前实例对象this

2.静态同步方法，锁是当前类的class对象

3.同步方法块，锁是括号里面的对象

### synchronized与lock区别



### Volatile(变量真正独立于其他变量和自己以前的值)

保证可见性

#### 适合使用场景

a）对变量的写入操作不依赖其当前值

​ 不满足：number++、count=count\*5等

​ 满足：boolean变量、直接赋值的变量等

b）该变量没有包含在具有其他变量的不变式中

​ 不满足：不变式 low<up

总结：变量真正独立于其他变量和自己以前的值，在单独使用的时候，适合用volatile

新值能立即同步到主内存，以及每次使用前立即从主内存刷新。

1.volatile 修饰的变量，是直接拿的主内存的值，就是说这个值永远是最新的，对其他线程是可见的。

2.而访问非 volatile 变量时，每个线程都会从系统内存（主内存）拷贝变量到工作内存中，然后修改工作内存中的变量值，操控的变量可能不同。

在一个变量被 volatile 修饰后，JVM 会为我们做两件事：

在每个 volatile 写操作前插入 StoreStore 屏障，在写操作后插入 StoreLoad 屏障。（StoreStore-写-StoreLoad）

在每个 volatile 读操作前插入 LoadLoad 屏障，在读操作后插入LoadStore屏障。（LoadLoad-读-LoadStore）

虽然 volatile 可见性保证了对 volatile 变量所有的写操作都能立刻反应到其他线程之中（即 volatile 变量在各个线程中都是一致的），但是 Java 里面的运算并非原子操作。只有是原子操作的 volatile 变量才是线程安全的，比如我们很常见的 变量++ 自增操作，在这个过程中，自增包括取数，加一，保存三个过程的操作，所以自增并不是原子性操作，使用 volatile 修饰的变量自增操作仍然是不安全的。

### synchronized和volatile比较

1.volatile不需要加锁，比synchronized更轻便，不会阻塞线程

2.synchronized既能保证可见性，又能保证原子性，而volatile只能保证可见性，无法保证原子性

与锁相比，Volatile 变量是一种非常简单但同时又非常脆弱的同步机制，它在某些情况下将提供优于锁的性能和伸缩性。如果严格遵循 volatile 的使用条件（变量真正独立于其他变量和自己以前的值 ） 在某些情况下可以使用 volatile 代替 synchronized 来优化代码提升效率。

### Volatile和锁的区别

　　Volatile:

　　当把变量声明为volatile类型后，编译器和运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其它内存操作一起重排序。volatile变量不会被缓存在寄存器或者对其他处理器不可见的地方，跳过 CPU cache ，因此在读取volatile类型变量时总会返回最新的值。

 　　也就是说volatile类型的变量保证了可见性 但是不能保证原子性 在进行自增等非原子性操作的时候依然会出现并发问题。

　　Volatile和锁:

　　当多个线程同时请求锁的时候，一些线程将被挂起并且等待其他线程执行完它们的时间片后才能被调度执行。频繁的线程间上下文切换及线程调度是十分耗资源的。另外锁还存在着死锁的风险。

　　与锁相比，volatile是一种更加轻量级的同步机制，因为在使用这些变量的时候不会发生上下文切换和线程调度等操作。但是volatile同样也存在局限性:当变量依赖于其他变量或旧值时（自增）就不能使用volatile变量，

　　因为他们不是原子操作。

## Redis

### 数据类型

#### 字符串（string）set key value

数据结构：简单动态字符串（SDS）

定义SDS对象，此对象中包含三个属性：

len buf中已经占有的长度(表示此字符串的实际长度)

free buf中未使用的缓冲区长度

buf[] 实际保存字符串数据的地方

空间分配原则：当len小于IMB（1024\*1024）时增加字符串分配空间大小为原来的2倍，当len大于等于1M时每次分配 额外多分配1M的空间。

#### 列表(List) lpush key value

数据结构：

在3.2版本之前，列表是使用ziplist和linkedlist实现的，之后，重新引入了一个quicklist的数据结构，列表的底层都是由quicklist实现的，quickList是一个ziplist组成的双向链表。每个节点使用ziplist来保存数据。。

##### Ziplist（压缩列表为 Redis 节约内存而开发）

当列表对象同时满足以下两个条件（可在redis.conf配置）时，列表对象使用ziplist编码：

1.列表对象保存的所有字符串元素的长度都小于64字节

2.列表对象保存的元素数量小于512个

ziplist 是由一系列特殊编码的内存块构成的列表(**像内存连续的数组，但每个元素长度不同**)， 一个 ziplist 可以包含多个节点（zlentry）。  
ziplist 将表中每一项存放在前后连续的地址空间内，每一项因占用的空间不同，而采用**变长编码**。  
由于内存是连续分配的，所以遍历速度很快。

ziplist 是一个特殊的双向链表  
特殊之处在于：没有维护双向指针:prev next；而是**存储上一个** zlentry**的长度和 当前**zlentry**的长度，通过长度推算下一个元素在什么地**方。  
**牺牲读取的性能，获得高效的存储空间，因为(简短字符串的情况)存储指针比存储**zlentry**长度 更费内存。这是典型的“时间换空间”**。

zlentry结构：

prevrawlen：记录前一个节点所占有的内存字节数，通过该值可以用来实现表尾向表头节点遍历；

len/encoding：记录了当前节点content占有的内存字节数及其存储类型，用来解析content用；

content：保存了当前节点的值。  
最关键的是prevrawlen和len/encoding，content只是实际存储数值的比特位。

###### ziplist连锁更新问题

但引起的性能问题的概率是极低的：需要列表中存在大量的节点长度接近254的zlentry。

##### Linkedlist

标准的双向链表

##### linkedlist和zipList优缺点：

1.双向链表linkedlist便于在表的两端进行push和pop操作，在**插入节点上复杂度很低，但是它的内存开销比较大**。首先，它在每个节点上除了要保存数据之外，还要**额外保存两个指针**；其次，双向链表的各个节点是单独的内存块，地址不连续，节点多了**容易产生内存碎片**。

2.ziplist存储在一段连续的内存上，所以**存储效率很高**。但是，它**不利于修改操作**，插入和删除操作需要频繁的申请和释放内存。特别是当ziplist长度很长的时候，一次realloc可能会导致大批量的数据拷贝。

##### quickList（3.2之后）

quickList就是一个标准的**双向链表**的配置，有head 有tail;  
每一个节点是一个quicklistNode，包含prev和next指针。  
每一个quicklistNode 包含 一个ziplist，\*zp 压缩链表里存储键值。  
所以quicklist是对ziplist进行一次封装，使用小块的ziplist来既保证了少使用内存，也保证了性能。

内部默认**单个 ziplist 长度为 8k 字节**

#### 哈希（hash）sadd key filed value

redis的hash实现分为ziplist(压缩列表)和dict(字典)。

##### Ziplist：

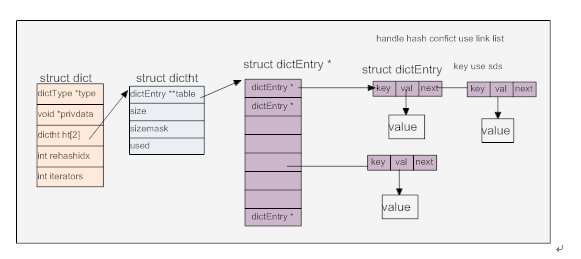
当hash对象可以同时满足一下两个条件时，哈希对象使用ziplist编码。

哈希对象保存的所有键值对的键和值的字符串长度都小于64字节  
哈希对象保存的键值对数量小于512个

##### dict(字典)

数据结构：hashtable哈希表的二维结构

数组+单链表



###### Rehash

**双哈希表**(ht[2])结构

1. 为ht[1]开辟空间，rehashidx赋值0，开始从\*table[0]开始将ht[0]的数据rehash至ht[1]
2. 随着rehashidx自增，对table[rehashidx]进行rehash。在rehash期间，对disc的读和修改操作会同时作用于ht[0]和ht[1]，插入作用于ht[1]
3. ht[0]的数据全部rehash至ht[1]时，rehash完成。rehashidx赋值-1
4. 将ht[1]赋值给ht[0]，清空ht[1]

#### 有序集合（sorted set）sadd key 1 value

#### 集合（set）sadd key value

### 分布式锁

加锁：Set key value nx px

value设置为唯一值用于解锁判断，nx不存在设置已存在不做设置，px过期时间毫秒

EX 过期时间秒

解锁："if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end"，

首先获取锁对应的value值，检查是否与传入值相等，如果相等则删除锁（解锁）

锁延时：

另外起一个job到一定的时间查看key是否存在，存在则延长时间，

不存在说明已经释放

### 命令：

<http://doc.redisfans.com/>

https://www.runoob.com/redis/redis-sets.html

只是在处理我们的网络请求的时候只有一个线程来处理

（1）多路 I/O 复用模型

多路I/O复用模型是利用 select、poll、epoll 可以同时监察多个流的 I/O 事件的能力，在空闲的时候，会把当前线程阻塞掉，当有一个或多个流有 I/O 事件时，就从阻塞态中唤醒，于是程序就会轮询一遍所有的流（epoll 是只轮询那些真正发出了事件的流），并且只依次顺序的处理就绪的流，这种做法就避免了大量的无用操作。

这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络 IO 的时间消耗），且 Redis 在内存中操作数据的速度非常快，也就是说内存内的操作不会成为影响Redis性能的瓶颈，主要由以上几点造就了 Redis 具有很高的吞吐量。

<https://blog.csdn.net/bntX2jSQfEHy7/article/details/79479152>

### 删除策略

定时删除:到了过期时间即清除

惰性删除:访问key时到期即清除

定期删除:每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key

Redis中同时使用了惰性过期和定期过期两种过期策略

<https://www.jianshu.com/p/8aa619933ebb>

内存满了:

1. 增加内存
2. 使用内存淘汰策略
3. Redis集群

### 内存淘汰策略:

redis设置配置文件的maxmemory参数，可以控制其最大可用内存大小（字节）

配置文件中的maxmemory-policy,其默认值是noeviction



#### LRU算法：

least RecentlyUsed，最近最少使用算法。也就是说默认删除最近最少使用的键

#### TTL算法(volatile):

过期时间表中随机挑选几个键值对

但是一定要注意一点！redis中并不会准确的删除所有键中最近最少使用的键，而是随机抽取3个键，删除这三个键中最近最少使用的键。

那么3这个数字也是可以设置的，对应位置是配置文件中的maxmeory-samples

<http://stor.51cto.com/art/201904/594773.htm>

Redis集群有三种方式：客户端分片、代理分片、RedisCluster

### 集群

#### 主从复制:

从服务器默认只读可以配置文件修改

master同步数据时是非阻塞式的，可以接收用户的读写请求。然而在slave端是阻塞模式的，slave在同步master数据时，并不能够响应客户端的查询

1. 从服务器向主服务器发送sync命令
2. 服务器接收到SYNC命名后，开始执行BGSAVE命令生成RDB文件并使用缓冲区记录此后执行的所有写命令
3. 主服务器BGSAVE执行完后，向所有从服务器发送快照文件，并在发送期间继续记录被执行的写命令；
4. 从服务器收到快照文件后丢弃所有旧数据，载入收到的快照；
5. 主服务器快照发送完毕后开始向从服务器发送缓冲区中的写命令；
6. 从服务器完成对快照的载入，开始接收命令请求，并执行来自主服务器缓冲区的写命令；（从服务器初始化完成）
7. 主服务器每执行一个写命令就会向从服务器发送相同的写命令，从服务器接收并执行收到的写命令（从服务器初始化完成后的操作）

缺点:

1.Redis不具备自动容错和恢复功能，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者手动切换前端的IP才能恢复。

2.主机宕机，宕机前有部分数据未能及时同步到从机，切换IP后还会引入数据不一致的问题，降低了系统的可用性。

3.Redis较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂

#### 哨兵模式:

总结:哨兵每秒一次向集群所有主.从.哨兵发送ping info命令-->实例回复时间超过down-after-milliseconds则主观下线(SDOWN)---->所有哨兵每秒一次确认主观下线----->大于等于配置文件指定值数量的哨兵在一定时间确认主观下线则被标记客观下线---->下线master的所有salve发送info命令改为每秒一次-->没有足够哨兵同意下线客观下线状态移除---->master返回有效回复则移除主观下线

1.每个Sentinel（哨兵）进程以每秒钟一次的频率向整个集群中的Master主服务器，Slave从服务器以及其他Sentinel（哨兵）进程发送一个 PING 命令。

2.如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 则这个实例会被 Sentinel（哨兵）进程标记为主观下线（SDOWN）

3.如果一个Master主服务器被标记为主观下线（SDOWN），则正在监视这个Master主服务器的所有 Sentinel（哨兵）进程要以每秒一次的频率确认Master主服务器的确进入了主观下线状态

4.当有足够数量的 Sentinel（哨兵）进程（大于等于配置文件指定的值）在指定的时间范围内确认Master主服务器进入了主观下线状态（SDOWN）， 则Master主服务器会被标记为客观下线（ODOWN）

5.在一般情况下， 每个 Sentinel（哨兵）进程会以每 10 秒一次的频率向集群中的所有Master主服务器、Slave从服务器发送 INFO 命令。

6.当Master主服务器被 Sentinel（哨兵）进程标记为客观下线（ODOWN）时，Sentinel（哨兵）进程向下线的 Master主服务器的所有 Slave从服务器发送 INFO 命令的频率会从 10 秒一次改为每秒一次。

7.若没有足够数量的 Sentinel（哨兵）进程同意 Master主服务器下线， Master主服务器的客观下线状态就会被移除。若 Master主服务器重新向 Sentinel（哨兵）进程发送 PING 命令返回有效回复，Master主服务器的主观下线状态就会被移除。

#### Redis-Cluster集群

增删节点迁移数据

<https://blog.csdn.net/wuxu_nanjing/article/details/77330568>

每个节点有两个东西一个插槽（slot），它的的取值范围是：0-16383,一个cluster(集群管理的插件),key到达时根据crc16算法得出一个结果对16384求余数,这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，通过这个值，去找到对应的插槽所对应的节点，然后直接自动跳转到这个对应的节点上进行存取操作

为了保证高可用，redis-cluster集群引入了主从模式，一个主节点对应一个或者多个从节点，当主节点宕机的时候，就会启用从节点。当其它主节点ping一个主节点A时，如果半数以上的主节点与A通信超时，那么认为主节点A宕机了。如果主节点A和它的从节点A1都宕机了，那么该集群就无法再提供服务了。

<https://www.cnblogs.com/runnerjack/p/10269277.html>

### 分布式事物，redis分布式锁的机制

分布式事务:

1. Redis的事务没有关系数据库事务提供的回滚（rollback），所以开发者必须在事务执行失败后进行后续的处理；  
   2.如果在一个事务中的命令出现错误，那么所有的命令都不会执行；  
   3.如果在一个事务中出现运行错误，那么正确的命令会被执行。

实现分布式锁有很多实现方式和工具，如Zookeeper、Redis等。

使用Redis实现分布式锁原理： setnx key value

Redis为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对Redis的连接并不存在竞争关系，基于此，Redis中可以使用SETNX命令实现分布式锁。(见谷歌收藏)

### 解决死锁

如果一个持有锁的客户端失败或崩溃了不能释放锁，该怎么解决？

答：给锁设置一个过期时间，可以通过两种方法实现：通过命令 “setnx 键名 过期时间 “；或者通过设置锁的expire时间，让Redis去删除锁。

1.使用 setnx key “当前系统时间+锁持有的时间”和getset key “当前系统时间+锁持有的时间”组合的命令就可以实现。

2.通过Redis中expire()给锁设定最大持有时间，如果超过，则Redis来帮我们释放锁

### 当redis百万key失效(expire 过期时间失效)，redis是怎么做的，会全部删除么 影响redis性能么

#### 会造成缓存雪崩问题

小编在做电商项目的时候，一般是采取不同分类商品，缓存不同周期。在同一分类中的商品，加上一个随机因子。这样能尽可能分散缓存过期时间，而且，热门类目的商品缓存时间长一些，冷门类目的商品缓存时间短一些，也能节省缓存服务的资源。

Redis采用的过期策略:惰性删除+定期删除

过期key对RDB没有任何影响

master删除过期key会写del命令到aof文件,slave的key没有过期时间而是直接执行aof的del命令删除键.

Redis 的主键失效机制对系统性能的影响

Redis 会定期地检查设置了失效时间的主键并删除已经失效的主键，但是通过对每次处理数据库个数的限制、**activeExpireCycle** 函数在一秒钟内执行次数的限制、分配给 activeExpireCycle 函数CPU时间的限制、继续删除主键的失效主键数百分比的限制，Redis 已经大大降低了主键失效机制对系统整体性能的影响，但是如果在实际应用中出现大量主键在短时间内同时失效的情况还是会产生很多问题，

也就是缓存穿透的情况。

#### 如何避免大量主键在同一时间同时失效造成数据库压力过大(即缓存雪崩)

合理的配置缓存可以增加系统的健壮性，避免缓存失效造成的事故。

1.在缓存失效后，通过加互斥锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。根据 key 去缓存层查询数据，当缓存层为命中时，对 key 加锁，然后从存储层查询数据，将数据写入缓存层，最后释放锁。若其他线程发现获取锁失败，则让线程休眠一段时间后重试。对于锁的类型，如果是在单机环境下可以使用 [Java](http://www.elecfans.com/tags/java/" \t "http://www.elecfans.com/consume/_blank) 并发包下的 Lock，如果是在分布式环境下，可以使用分布式锁（Redis 中的 SETNX 方法）。

2.可以通过缓存reload机制，预先去更新缓存.

3.不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

做二级缓存，或者双缓存策略。A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期。

1. 异步重建缓存：在这种方案下构建缓存采取异步策略，每个Redis key 维护逻辑超时时间，当逻辑超时时间小于当前时间时，则说明当前缓存已经失效，应当进行缓存更新，否则说明当前缓存未失效，直接返回缓存中的 value 值。如在Redis 中将 key 的过期时间设置为 60 [mi](http://www.hqpcb.com/" \t "http://www.elecfans.com/consume/_blank)n，在对应的 value 中设置逻辑过期时间为 30 min。这样当 key 到了 30 min 的逻辑过期时间，就可以异步更新这个 key 的缓存，但是在更新缓存的这段时间内，旧的缓存依然可用。这种异步重建缓存的方式可以有效避免大量的 key 同时失效。

5.缓存不过期

#### 缓存穿透

不存在的对象每次都会透过redis直接访问数据库

采用缓存空值的方式,查询为null也设置到redis设置过期时间

代码流程

1.参数传入对象主键ID

2.根据key从缓存中获取对象

3.如果对象不为空，直接返回

4.如果对象为空，进行数据库查询

5.如果从数据库查询出的对象不为空，则放入缓存（设定过期时间）

想象一下这个情况，如果传入的参数为-1，会是怎么样？这个-1，就是一定不存在的对象。就会每次都去查询数据库，而每次查询都是空，每次又都不会进行缓存。假如有恶意攻击，就可以利用这个漏洞，对数据库造成压力，甚至压垮数据库。即便是采用UUID，也是很容易找到一个不存在的KEY，进行攻击。

小编在工作中，会采用缓存空值的方式，也就是【代码流程】中第5步，如果从数据库查询的对象为空，也放入缓存，只是设定的缓存过期时间较短，比如设置为60秒。

1. 布隆过滤器拦截

#### 无底洞问题

加机器反而性能下降

#### 热点key的重建优化

解决:

1. 利用二级缓存：

比如利用ehcache，或者一个HashMap都可以。在你发现热key以后，把热key加载到系统的JVM中。  
针对这种热key请求，会直接从jvm中取，而不会走到redis层。假设此时有十万个针对同一个key的请求过来,如果没有本地缓存，这十万个请求就直接怼到同一台redis上了。  
现在假设，你的应用层有50台机器，OK，你也有jvm缓存了。这十万个请求平均分散开来，每个机器有2000个请求，会从JVM中取到value值，然后返回数据。避免了十万个请求怼到同一台redis上的情形。

1. 备份热key  
   这个方案也很简单。不要让key走到同一台redis上不就行了。我们把这个key，在多个redis上都存一份不就好了。接下来，有热key请求进来的时候，我们就在有备份的redis上随机选取一台，进行访问取值，返回数据。

https://blog.csdn.net/zhangdx001/article/details/106889182

1.互斥锁 思路简单保证一致性 缺点:代码复杂度增加 存在死锁风险

2.永不过期 给value设置逻辑过期时间 超过时间 单独一个线程刷新缓存 优点基本杜绝热点key问题 缺点不保证一致性,增加维护成本和内存成本

### Redis实战

#### Redis 有序集合(sorted set)

：ZADD runoobkey 1 redis

redis 正是通过分数来为集合中的成员进行从小到大的排序。

有序集合的成员是唯一的,但分数(score)却可以重复。

Sadd = tagging(可应用于标签)

Spop/srandmember = randow item(随机数)

Sadd + sinter = social graph(社交相关)

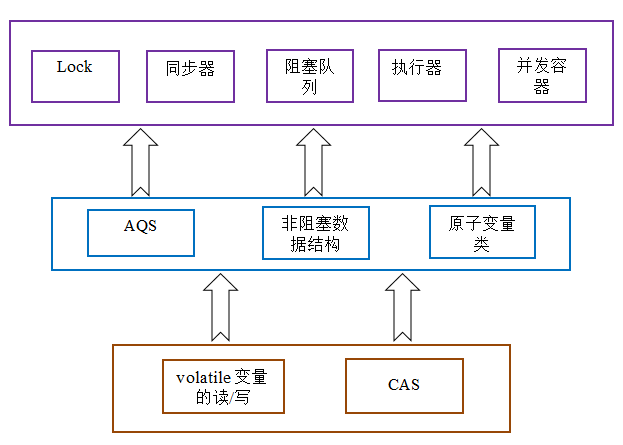
Lrush + lpop =stack(实现栈的功能)

Lpush + rpop = Queue(实现队列功能)

Lpush + trim = capped collection(控制列表大小)

Lpush + brpop(阻塞队列) = message queue(消息队列)

## JUC



### CAS

#### 原理:

CAS的思想很简单：三个参数，一个当前内存值V、旧的预期值A、即将更新的值B，当且仅当旧的预期值A和内存值V相同时，将内存值修改为B并返回true，否则什么都不做，并返回false。如果CAS操作失败，通过自旋的方式等待并再次尝试，直到成功。

#### 多CPU的CAS处理

CAS可以保证一次的读-改-写操作是原子操作，在单处理器上该操作容易实现，但是在多处理器上实现就有点儿复杂了。CPU提供了两种方法来实现多处理器的原子操作：总线加锁或者缓存加锁。

总线加锁：总线加锁就是就是使用处理器提供的一个LOCK#信号，当一个处理器在总线上输出此信号时，其他处理器的请求将被阻塞住,那么该处理器可以独占使用共享内存。但是这种处理方式显得有点儿霸道，不厚道，他把CPU和内存之间的通信锁住了，在锁定期间，其他处理器都不能其他内存地址的数据，其开销有点儿大。

缓存加锁：其实针对于上面那种情况我们只需要保证在同一时刻对某个内存地址的操作是原子性的即可。缓存加锁就是缓存在内存区域的数据如果在加锁期间，当它执行锁操作写回内存时，处理器不在输出LOCK#信号，而是修改内部的内存地址，利用缓存一致性协议来保证原子性。缓存一致性机制可以保证同一个内存区域的数据仅能被一个处理器修改，也就是说当CPU1修改缓存行中的i时使用缓存锁定，那么CPU2就不能同时缓存了i的缓存行。

#### CAS缺陷

CAS虽然高效地解决了原子操作，但是还是存在一些缺陷的，主要表现在三个方法：循环时间太长、只能保证一个共享变量原子操作、ABA问题。

1.循环时间太长

如果CAS一直不成功呢？这种情况绝对有可能发生，如果自旋CAS长时间地不成功，则会给CPU带来非常大的开销。在JUC中有些地方就限制了CAS自旋的次数，例如BlockingQueue的SynchronousQueue。

2.只能保证一个共享变量原子操作

看了CAS的实现就知道这只能针对一个共享变量，如果是多个共享变量就只能使用锁了。

3.ABA问题

CAS需要检查操作值有没有发生改变，如果没有发生改变则更新。但是存在这样一种情况：如果一个值原来是A，变成了B，然后又变成了A，那么在CAS检查的时候会发现没有改变，但是实质上它已经发生了改变，这就是所谓的ABA问题。对于ABA问题其解决方案是加上版本号，即在每个变量都加上一个版本号，每次改变时加1，即A —> B —> A，变成1A —> 2B —> 3A(AtomicStampedReference)。

## 并发包

### AQS(AbstractQueuedSynchronizer），即队列同步器

Java 并发世界的核心基石：双向列表结构，加锁不成功时，当前的线程就会把自己纳入到等待链表的尾部，然后调用 LockSupport.park 将自己休眠。其它线程解锁时，会从链表的表头取一个节点，调用 LockSupport.unpark 唤醒它。底层是CAS

AQS提供了一种实现阻塞锁和一系列依赖FIFO（先进先出）等待队列的同步器的框架

底层三个内容：  
                1.state（用于计数器，原子变量，锁状态）  
                2.线程标记（哪一个线程加的锁）  
                3.阻塞队列（用于存放阻塞线程）

http://blog.itpub.net/69902581/viewspace-2666640/

### synchronized 和Lock区别

<https://blog.csdn.net/hefenglian/article/details/82383569>

### Lock的condition实现原理

<https://blog.csdn.net/javazejian/article/details/75043422>

### ReentrantLock(重入锁)

可重入

可中断

可限时

公平锁

### Condition

singal()相当于notify(),singalAll()等方法

### ReadWriteLock

### CountDownLatch（同步器）

* 运行多个或一个线程，等待其他多个或一个线程执行完毕，在执行当前线程,

CountDownLatch使用AQS实现的，使用AQS的state变量来存放计数器的值。在调用CountDownLatch的构造函数时，会调用内部类Sync的构造函数将值赋给state变量，当多个线程调用countdown方法时实际是使用CAS递减state变量的值；当线程调用await方法后当前线程会被放入AQS阻塞队列，等待计数器为0时返回，即所有线程都调用了countdown方法时。最后，当计数器的值变为0时，当前线程还会调用AQS的doReleasedShared()方法激活调用await()方法而被阻塞的线程。

(生成excel多个sheet时可以使用)

### CyclicBarrier（循环屏障）

* 运行多个线程之间相互等待，全部执行到await（）时再全部执行await（）后面的代码。

### Semaphore(信号量)

原理：初始化时设置令牌数量（信号量），线程来时，semaphore.acquire();

获取令牌，令牌数减1，执行完时，调用semaphore.release();令牌数加1，令牌用完时，线程加入阻塞队列并挂起。

常用于限流：如连接池，同时进行连接的线程有数量限制，连接不能超过一定的数量，当连接达到了限制数量后，后面的线程只能排队等前面的线程释放了数据库连接才能获得数据库连接。

### LockSupport(挂起)

Unpark发生在park之前并不会被阻塞

Park可以被中断 不会抛出异常会直接返回

### ConcurrentHashMap

不加锁,会一直尝试获取锁,自旋,性能高

见集合3

### BlockingQueue(阻塞（加锁）队列)

https://blog.csdn.net/testcs\_dn/article/details/78083966

在多线程领域：所谓阻塞，在某些情况下会挂起线程（即阻塞），一旦条件满足，被挂起的线程又会自动被唤醒

先进先出（FIFO）：先插入的队列的元素也最先出队列，类似于排队的功能。从某种程度上来说这种队列也体现了一种公平性。

后进先出（LIFO）：后插入队列的元素最先出队列，这种队列优先处理最近发生的事件。

#### 核心方法：

##### 1.放入数据

（1）offer(anObject):表示如果可能的话,将anObject加到BlockingQueue里,即如果BlockingQueue可以容纳,则返回true,否则返回false.（本方法不阻塞当前执行方法 的线程）；　　　　　　   
（2）offer(E o, long timeout, TimeUnit unit)：可以设定等待的时间，如果在指定的时间内，还不能往队列中加入BlockingQueue，则返回失败。

（3）put(anObject):把anObject加到BlockingQueue里,如果BlockQueue没有空间,则调用此方法的线程**被阻断**直到BlockingQueue里面有空间再继续.

##### 2. 获取数据

（1）poll(time):取走BlockingQueue里排在首位的对象,若不能立即取出,则可以等time参数规定的时间,取不到时返回null;

（2）poll(long timeout, TimeUnit unit)：从BlockingQueue取出一个队首的对象，如果在指定时间内，队列一旦有数据可取，则立即返回队列中的数据。否则知道时间

超时还没有数据可取，返回失败。

（3）take():取走BlockingQueue里排在首位的对象,若BlockingQueue为空,**阻断进入等待状态**直到BlockingQueue有新的数据被加入;

（4）drainTo():一次性从BlockingQueue获取所有可用的数据对象（还可以指定获取数据的个数），通过该方法，可以提升获取数据效率；**不需要多次分批加锁或释放锁**。

#### ArrayBlockingQueue（基于数组）

ArrayBlockingQueue在生产者放入数据和消费者获取数据，都是共用**同一个锁对象**，也许是因为ArrayBlockingQueue的数据写入和获取操作已经足够轻巧，以至于引入独立的锁机制，除了给代码带来额外的复杂性外，其在性能上完全占不到任何便宜，和LinkedBlockingQueue区别，前者在插入或删除元素时不会产生或销毁任何额外的对象实例，而后者则会生成一个额外的Node对象，对GC有影响

#### LinkedBlockingQueue（基于链表）

只有当队列缓冲区**达到最大值缓存容量**时（LinkedBlockingQueue可以通过构造函数指定该值），才会**阻塞**生产者队列，直到消费者从队列中消费掉一份数据，生产者线程会被唤醒，反之对于消费者这端的处理也基于同样的原理。而LinkedBlockingQueue之所以能够高效的处理并发数据，还因为其对于生产者端和消费者端分别采用了**独立的锁**来控制数据同步，这也意味着在高并发的情况下生产者和消费者可以并行地操作队列中的数据，以此来提高整个队列的并发性能。如果构造一个LinkedBlockingQueue对象，而**没有指定其容量大小**，LinkedBlockingQueue会默认一个类似**无限大小**的容量（Integer.MAX\_VALUE），这样的话，如果生产者的速度一旦大于消费者的速度，也许还没有等到队列满阻塞产生，**系统内**存就有可能已被**消耗殆尽**了。

#### DelayQueue

DelayQueue中的元素只有当其指定的**延迟时间到了**，才能够从队列中获取到该元素。DelayQueue是一个没有大小限制的队列，因此往队列中插入数据的操作（**生产者）永远不会被阻塞，而只有获取数据的操作（消费者）才会被阻塞**。

使用场景：

DelayQueue使用场景较少，但都相当巧妙，常见的例子比如使用一个DelayQueue来管理一个**超时未响应的连接队列**。

#### PriorityBlockingQueue

基于优先级的阻塞队列，不会阻塞数据生产者，而只会在没有可消费的数据时，阻塞数据的消费者，因此使用的时候要特别注意，生产者生产数据的速度绝对不能快于消费者消费数据的速度，否则时间一长，会最终耗尽所有的可用堆内存空间

#### SynchronousQueue

无缓冲的等待队列，如果一方没有找到合适的目标，那么对不起，大家都会等待。

声明一个SynchronousQueue有两种不同的方式，它们之间有着不太一样的行为。公平模式和非公平模式的区别:

如果采用公平模式：SynchronousQueue会采用公平锁，并配合一个FIFO队列来阻塞多余的生产者和消费者，从而体系整体的公平策略；

但如果是非公平模式（SynchronousQueue默认）：SynchronousQueue采用非公平锁，同时配合一个LIFO队列来管理多余的生产者和消费者，而后一种模式，如果生产者和消费者的处理速度有差距，则很容易出现**饥渴**的情况，即可能有某些生产者或者是消费者的数据永远都**得不到处理**。

### ConcurrentLinkedQueue （非阻塞（自旋）队列）

不加锁,会一直尝试获取锁,自旋,性能高

## 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

1. cas实现
2. 版本号控制

### 悲观锁：

先取锁再访问策略分为共享锁，排他锁

共享锁：读锁

排他锁：写锁

### 锁优化

#### 减少锁持有的时间

#### 减小锁粒度

将大对象拆成小对象如ConcurrentHashMap

#### 锁分离

根据功能进行锁分离：读写锁ReadWriteLock

LinkedBrockingQueue头部拿 尾部插

#### 锁粗化

对同一个锁不停地请求同步释放本身也会消耗系统宝贵的资源反而不利于性能的优化如：循环中加锁

#### 锁消除

内置于JVM中的获取锁的优化方法和获取锁的步骤

– 偏向锁可用会先尝试偏向锁

– 轻量级锁可用会先尝试轻量级锁

– 以上都失败，尝试自旋锁

– 再失败，尝试普通锁，使用OS互斥量在操作系统层挂起

### 轻量级锁

### 自旋锁

TryLock

## Java8

### LongAdder

– 和AtomicInteger类似的使用方式

– 在AtomicInteger上进行了热点分离

– public void add(long x)

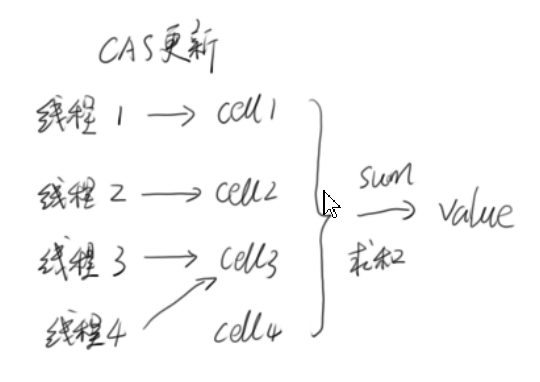
– public void increment()

– public void decrement()

– public long sum()

– public long longValue()

– public int intValue()



### StampedLock

– 读写锁的改进

– 读不阻塞写

：tryOptimisticRead乐观读

：readLock读写锁

## 高并发

高并发就是系统能同时处理很多请求，它的指标有响应时间、吞吐量、QPS

提高并发能力有两种方法：

垂直扩展：只要是提升单机处理能力。它也分为两种：第一增强单机硬件性能，第二提升单机架构性能，例如使用cache来减少IO次数、使用异步来增加单服务吞吐量、使用无锁数据结构来减少响应时间。

水平扩展：只要增加服务器数量，就能线性扩充系统性能

反向代理层可以通过“DNS轮询”的方式进行水平扩展

应用层通过Nginx来进行水平扩展

服务层通过服务连接池来进行水平扩展

数据库层可以按照数据范围、或者数据哈希的方式来进行水平扩展

## 框架

### Mybatis

流程:

1.创建SqlSessionFactoryBuilder对象 构建者模式

2.将主配置文件转换成流

3.创建sessionFactory工厂 工厂模式

4.创建一个sqlSession,自动提交

5.通过动态代理获得Dao接口的代理对象 代理模式

#### 预编译

<https://www.bbsmax.com/A/MAzALYGnd9/>

#### 缓存

##### 一级缓存:

一级缓存是SqlSession级别的缓存。在操作数据库时需要构造sqlSession对象，在对象中有一个数据结构用于存储缓存数据。不同的sqlSession之间的缓存数据区域是互相不影响的。也就是他只能作用在同一个sqlSession中，不同的sqlSession中的缓存是互相不能读取的

还有一个原因，实际开发中，MyBatis通常和Spring进行整合开发。Spring将事务放到Service中管理，对于每一个service中的sqlsession是不同的，这是通过mybatis-spring中的org.mybatis.spring.mapper.MapperScannerConfigurer创建sqlsession自动注入到service中的。 每次查询之后都要进行关闭sqlSession，关闭之后数据被清空。所以spring整合之后，如果没有事务，一级缓存是没有意义的

1、一级缓存的生命周期有多长？

　　a、MyBatis在开启一个数据库会话时，会 创建一个新的SqlSession对象，SqlSession对象中会有一个新的Executor对象。Executor对象中持有一个新的PerpetualCache对象；当会话结束时，SqlSession对象及其内部的Executor对象还有PerpetualCache对象也一并释放掉。

　　b、如果SqlSession调用了close()方法，会释放掉一级缓存PerpetualCache对象，一级缓存将不可用。

　　c、如果SqlSession调用了clearCache()，会清空PerpetualCache对象中的数据，但是该对象仍可使用。

　　d、SqlSession中执行了任何一个update操作(update()、delete()、insert()) ，都会清空PerpetualCache对象的数据，但是该对象可以继续使用

2、怎么判断某两次查询是完全相同的查询？

　　mybatis认为，对于两次查询，如果以下条件都完全一样，那么就认为它们是完全相同的两次查询。

　　2.1 传入的statementId

　　2.2 查询时要求的结果集中的结果范围

　　2.3. 这次查询所产生的最终要传递给JDBC java.sql.Preparedstatement的Sql语句字符串（boundSql.getSql() ）

　　2.4 传递给java.sql.Statement要设置的参数值

##### 二级缓存:

是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。

UserMapper有一个二级缓存区域（按namespace分），其它mapper也有自己的二级缓存区域（按namespace分）。每一个namespace的mapper都有一个二级缓存区域，两个mapper的namespace如果相同，这两个mapper执行sql查询到数据将存在相同的二级缓存区域中

前面我们说到，Spring和MyBatis整合时， 每次查询之后都要进行关闭sqlSession，关闭之后数据被清空。所以spring整合之后，如果没有事务，一级缓存是没有意义的。那么如果开启二级缓存，关闭sqlsession后，会把该sqlsession一级缓存中的数据添加到namespace的二级缓存中。这样，缓存在sqlsession关闭之后依然存在。

如果我们配置了二级缓存就意味着：

 映射语句文件中的所有select语句将会被缓存。

映射语句文件中的所欲insert、update和delete语句会刷新缓存。

缓存会使用默认的Least Recently Used（LRU，最近最少使用的）算法来收回。

根据时间表，比如No Flush Interval,（CNFI没有刷新间隔），缓存不会以任何时间顺序来刷新。

缓存会存储列表集合或对象(无论查询方法返回什么)的1024个引用

缓存会被视为是read/write(可读/可写)的缓存，意味着对象检索不是共享的，而且可以安全的被调用者修改，不干扰其他调用者或线程所做的潜在修改。

总结：

对于查询多commit少且用户对查询结果实时性要求不高，此时采用mybatis二级缓存技术降低数据库访问量，提高访问速度。

但不能滥用二级缓存，二级缓存也有很多弊端，从MyBatis默认二级缓存是关闭的就可以看出来。

二级缓存是建立在同一个namespace下的，如果对表的操作查询可能有多个namespace，那么得到的数据就是错误的。

举个简单的例子:

订单和订单详情，orderMapper、orderDetailMapper。在查询订单详情时我们需要把订单信息也查询出来，那么这个订单详情的信息被二级缓存在orderDetailMapper的namespace中，这个时候有人要修改订单的基本信息，那就是在orderMapper的namespace下修改，他是不会影响到orderDetailMapper的缓存的，那么你再次查找订单详情时，拿到的是缓存的数据，这个数据其实已经是过时的。

根据以上，想要使用二级缓存时需要想好两个问题：

1）对该表的操作与查询都在同一个namespace下，其他的namespace如果有操作，就会发生数据的脏读。

2）对关联表的查询，关联的所有表的操作都必须在同一个namespace。

<https://blog.csdn.net/weixin_39312465/article/details/86624486>

#### [$与#的区别](https://www.cnblogs.com/wang-yaz/p/10044683.html)

1.#{} 传入值时，sql解析时，参数是带引号的，而${}传入值，sql解析时，参数是不带引号的

2.使用#可以很大程度上防止sql注入。(语句的拼接  #{xxx},使用的是PreparedStatement,会有类型转换，比较安全 简单的说就是#{}是经过预编译的，是安全的，${}是未经过预编译的，仅仅是取变量的值，是非安全的，存在SQL注入。）

3.order by 中就需要使用 $ : ORDER BY integral ASC 与 ORDER BY 'integral' 'asc' 的区别

<https://www.jb51.net/article/101090.htm>

<https://blog.csdn.net/u014476019/article/details/45878771>

### Spring

#### Bean的生命周期

1. 实例化bean对象(通过**构造方法或者工厂方法**)利用SimpleInstantiationStragety进行**初始化**（可以是工厂方式的，也可以是单参数形式的，也可以是无参数的简单初始化），结果是我们得到了一个实例，但是属性都是空的
2. 进行**属性的注入**，这里分别通过byName或者byType获取属性信息，并递归调用getBean方法获取属性bean
3. 根据bean实现的aware接口调用相应的aware方法**设置资源**（beanNameAware（**设置bean的名称**，可以直接使用），BeanClassLoaderAware，BeanFactoryAware（可以根据之前的**bean name获取这个对象来使用**））
4. 将Bean实例传递给Bean的**前置处理器**的postProcessBeforeInitialization(Object bean, String beanname)方法
5. 调用用户自定义的**init-method**方法
6. 将Bean实例传递给Bean的**后置处理器**的postProcessAfterInitialization(Object bean, String beanname)方法
7. bean可以使用了
8. 容器关闭之前，调用Bean的**销毁方法**

#### Ioc

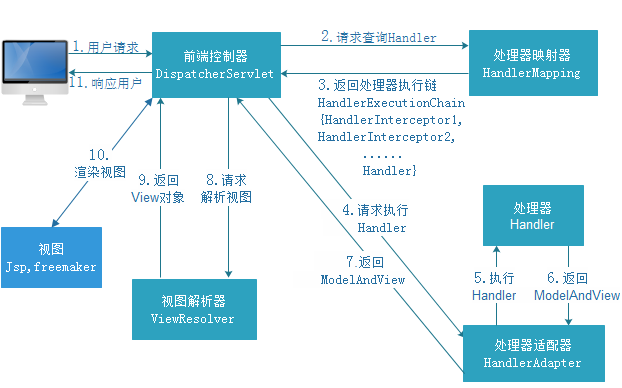
控制反转,

#### 解决循环依赖

https://blog.csdn.net/lkforce/article/details/97183065?utm\_medium=distribute.pc\_relevant.none-task-blog-BlogCommendFromBaidu-1.control&depth\_1-utm\_source=distribute.pc\_relevant.none-task-blog-BlogCommendFromBaidu-1.control

| **步骤** | **操作** | **三层列表中的内容** |
| --- | --- | --- |
| 1 | 开始初始化对象A | singletonFactories：  earlySingletonObjects：  singletonObjects： |
| 2 | 调用A的构造，把A放入singletonFactories | singletonFactories：A  earlySingletonObjects：  singletonObjects： |
| 3 | 开始注入A的依赖，发现A依赖对象B | singletonFactories：A  earlySingletonObjects：  singletonObjects： |
| 4 | 开始初始化对象B | singletonFactories：A,B  earlySingletonObjects：  singletonObjects： |
| 5 | 调用B的构造，把B放入singletonFactories | singletonFactories：A,B  earlySingletonObjects：  singletonObjects： |
| 6 | 开始注入B的依赖，发现B依赖对象A | singletonFactories：A,B  earlySingletonObjects：  singletonObjects： |
| 7 | 开始初始化对象A，发现A在singletonFactories里有，则直接获取A，  把A放入earlySingletonObjects，把A从singletonFactories删除 | singletonFactories：B  earlySingletonObjects：A  singletonObjects： |
| 8 | 对象B的依赖注入完成 | singletonFactories：B  earlySingletonObjects：A  singletonObjects： |
| 9 | 对象B创建完成，把B放入singletonObjects，  把B从earlySingletonObjects和singletonFactories中删除 | singletonFactories：  earlySingletonObjects：A  singletonObjects：B |
| 10 | 对象B注入给A，继续注入A的其他依赖，直到A注入完成 | singletonFactories：  earlySingletonObjects：A  singletonObjects：B |
| 11 | 对象A创建完成，把A放入singletonObjects，  把A从earlySingletonObjects和singletonFactories中删除 | singletonFactories：  earlySingletonObjects：  singletonObjects：A,B |
| 12 | 循环依赖处理结束，A和B都初始化和注入完成 | singletonFactories：  earlySingletonObjects：  singletonObjects：A,B |

### Springmvc

springmvc的执行流程:

处理器映射器返回处理器执行链(即找到合适的处理器适配器)--->处理器适配器根据url匹配---->处理器---->controller返回modelandview--->dispatcherServlet---->视图解析器返回view对象---->视图------>响应客户

### Springboot

起步依赖,自动装配

#### Springboot的核心配置是什么？自动装配？

Spring Boot 的核心配置

: 启动类:SpringApplication.run(SampleController.class, args);

: 核心注解@SpringBootApplication

: 配置文件:

application配置文件:application配置文件是应用级别的，是当前应用的配置文件。

bootstrap配置文件:bootstrap配置文件是**系统级别**的，用来加载外部配置，如**配置中心的配置信息**，也可以用来定义系统不会变化的属性。**bootstatp文件的加载先于application**文件。

自动装配:@SpringBootConfiguration、@EnableAutoConfiguration和@ComponentScan

当我们使用@EnableAutoConfiguration注解激活自动装配时，实质对应着很多XXXAutoConfiguration类在执行装配工作，这些XXXAutoConfiguration类是在spring-boot-autoconfigure jar中的META-INF/spring.factories文件中配置好的，@EnableAutoConfiguration通过SpringFactoriesLoader机制创建XXXAutoConfiguration这些bean。XXXAutoConfiguration的bean会依次执行并判断是否需要创建对应的bean注入到Spring容器中。

在每个XXXAutoConfiguration类中，都会利用多种类型的条件注解@ConditionOnXXX对当前的应用环境做判断，如应用程序是否为Web应用、classpath路径上是否包含对应的类、Spring容器中是否已经包含了对应类型的bean。如果判断条件都成立，XXXAutoConfiguration就会认为需要向Spring容器中注入这个bean，否则就忽略。

### Springcloud

#### Eureka：

较详细：<https://segmentfault.com/a/1190000020889150>

Service之间通过定时任务同步信息（PeerEurekaNodes与PeerEurekaNode类实现）

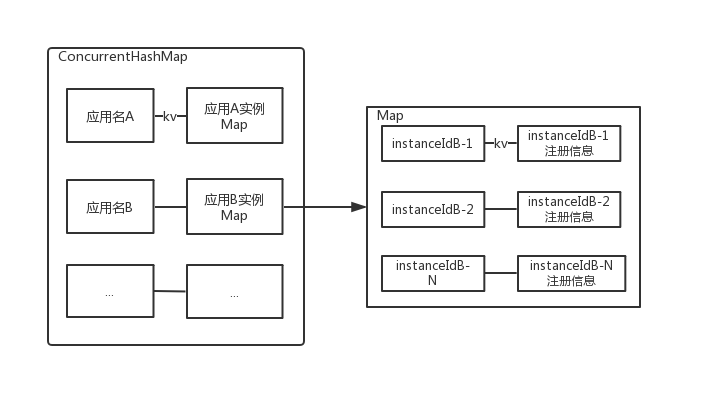
##### 注册表：

直接基于纯内存

Key1：服务名称

Key2:服务实例的id

 Lease:InstanceInfo就代表了服务实例的具体信息，比如机器的ip地址、hostname以及端口号。每个服务最近一次发送心跳的时间



Eureka 的数据存储分了两层：数据存储层和缓存层。  
Eureka Client 在拉取服务信息时，先从缓存层获取（相当于 Redis），如果获取不到，先把数据存储层的数据加载到缓存中（相当于 Mysql），再从缓存中获取。值得注意的是，数据存储层的数据结构是服务信息，而缓存中保存的是经过处理加工过的、可以直接传输到 Eureka Client 的数据结构。  
Eureka 这样的数据结构设计是把内部的数据存储结构与对外的数据结构隔离开了，就像是我们平时在进行接口设计一样，对外输出的数据结构和数据库中的数据结构往往都是不一样的。

##### Eureka Server：注册中心服务端

注册中心服务端主要对外提供了三个功能：

1、服务注册  
服务提供者启动时，会通过 Eureka Client 向 Eureka Server 注册信息，Eureka Server 会存储该服务的信息，Eureka Server 内部有二层缓存机制来维护整个注册表

2、提供注册表  
服务消费者在调用服务时，如果 Eureka Client 没有缓存注册表的话，会从 Eureka Server 获取最新的注册表

3、同步状态  
Eureka Client 通过注册、心跳机制和 Eureka Server 同步当前客户端的状态。

##### Eureka Client：注册中心客户端 Eureka Client 是一个 Java 客户端，用于简化与 Eureka Server 的交互。Eureka Client 会拉取、更新和缓存 Eureka Server 中的信息。因此当所有的 Eureka Server 节点都宕掉，服务消费者依然可以使用缓存中的信息找到服务提供者，但是当服务有更改的时候会出现信息不一致。

1、Register: 服务注册  
服务的提供者，将自身注册到注册中心，服务提供者也是一个 Eureka Client。当 Eureka Client 向 Eureka Server 注册时，它提供自身的元数据，比如 IP 地址、端口，运行状况指示符 URL，主页等。

2、Renew: 服务续约  
Eureka Client 会每隔 30 秒发送一次心跳来续约。 通过续约来告知 Eureka Server 该 Eureka Client 运行正常，没有出现问题。 默认情况下，如果 Eureka Server 在 90 秒内没有收到 Eureka Client 的续约，Server 端会将实例从其注册表中删除，此时间可配置，一般情况不建议更改。

Eviction 服务剔除  
当 Eureka Client 和 Eureka Server 不再有心跳时，Eureka Server 会将该服务实例从服务注册列表中删除，即服务剔除。

Cancel: 服务下线  
Eureka Client 在程序关闭时向 Eureka Server 发送取消请求。 发送请求后，该客户端实例信息将从 Eureka Server 的实例注册表中删除。该下线请求不会自动完成，它需要调用以下内容：

##### GetRegisty: 获取注册列表信息

该注册列表信息定期（每30秒钟）更新一次，如果由于某种原因导致注册列表信息不能及时匹配，Eureka Client 则会重新获取整个注册表信息

##### Remote Call: 远程调用 当 Eureka Client 从注册中心获取到服务提供者信息后，就可以通过 Http 请求调用对应的服务；服务提供者有多个时，Eureka Client 客户端会通过 Ribbon 自动进行负载均衡。

##### 自我保护机制

如果 Eureka Server 在一定的 90s 内没有接收到某个微服务实例的心跳，会注销该实例。Eureka Server 在运行期间会去统计心跳失败比例在 15 分钟之内是否低于 85%，如果低于 85%，Eureka Server 即会进入自我保护机制。

Eureka Server 进入自我保护机制，会出现以下几种情况：  
(1 Eureka 不再从注册列表中移除因为长时间没收到心跳而应该过期的服务  
(2 Eureka 仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上(即保证当前节点依然可用)  
(3 当网络稳定时，当前实例新的注册信息会被同步到其它节点中

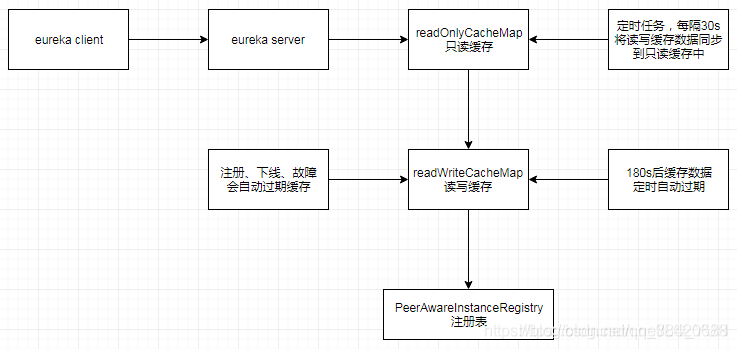
##### 注册

https://www.cnblogs.com/joimages/p/13258939.html

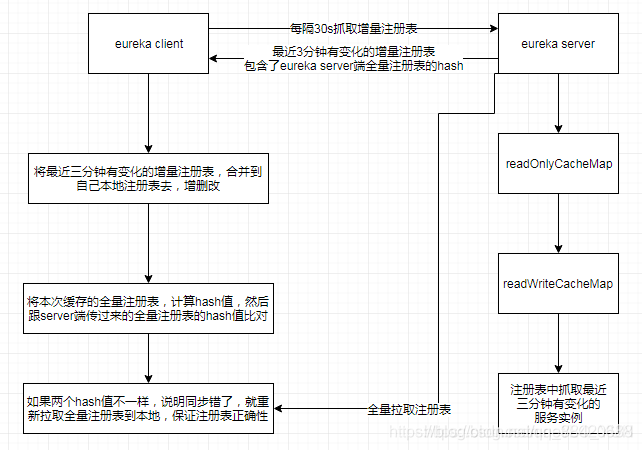
##### 同步注册表

https://blog.csdn.net/qq\_38420688/article/details/108684492

第一次从注册表**全量拉取**注册表到eureka client，后面**每隔30秒增量**拉取注册表，先从一级缓存ReadOnlyCacheMap（ConcurrentHashMap）中去获取注册表数据，如果获取不到的话再去耳二级缓存ReadWriteCacheMap（guava cache）中找，如果还是找不到的话，那就只能重新从注册表（ClassLoader的load方法）中拉取了



增量拉取注册表的实现借助了一个ConcurrentLinkedQueue类型的变量recentlyChangedQueue，默认情况下recentlyChangedQueue里面存放的是180秒内修改的服务实例信息，后台会有一个定时任务来维护recentlyChangedQueue，增量拉取注册表时，会将本地的注册表和recentlyChangedQueue中的服务实例进行一个合并，同时会获取eureka server全量注册表的hash值，在eureka client增量同步注册表完成之后，也会计算一个hash值，然后将自己计算出来的这个hash值和eureka server全量注册表的hash值进行比对，如果是一致的，说明增量数据同步没问题，反之则说了增量数据同步出现了不一致，那么就会重新从eureka server（存储层）全量拉取一份最新的服务注册表



###### 多级缓存机制有多种过期策略

* 主动过期：当服务实例发生注册、下线、故障的时候，ReadWriteCacheMap中所有的缓存过期掉
* 定时过期：readWriteCacheMap在构建的时候，指定了一个自动过期的时间，默认值就是180秒，所以你往readWriteCacheMap中放入一个数据，180秒过后，就将这个数据给他过期了
* 被动过期：默认是每隔30秒，执行一个定时调度的线程任务，对readOnlyCacheMap和readWriteCacheMap中的数据进行一个比对，如果两块数据是不一致的，那么就将readWriteCacheMap中的数据放到readOnlyCacheMap中来
* 删除二级缓存：

Eureka Client 发送 register、renew 和 cancel 请求并更新 registry 注册表之后，删除二级缓存；  
Eureka Server 自身的 Evict Task 剔除服务后，删除二级缓存；  
二级缓存本身设置了 guava 的失效机制，隔一段时间后自己自动失效；

* 加载二级缓存：

Eureka Client 发送 getRegistry 请求后，如果二级缓存中没有，就触发 guava 的 load，即从 registry 中获取原始服务信息后进行处理加工，再加载到二级缓存中。  
Eureka Server 更新一级缓存的时候，如果二级缓存没有数据，也会触发 guava 的 load。

* 更新一级缓存：

Eureka Server 内置了一个 TimerTask，定时将二级缓存中的数据同步到一级缓存（这个动作包括了删除和加载）。

#### Ribbon：

定时从Eureka Client拉取注册信息（默认30秒）

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,随机负载规则很简单，随机整数选择服务，最终达到随机负载均衡。我们可以配置不同的Rule来实现不同的负载方式。

#### Feign（HTTP）：

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

基于面向接口的动态代理方式生成实现类。

#### Hystrix：

<https://www.cnblogs.com/jing99/p/11625306.html>

打开走降级逻辑，关闭走正常逻辑

红线：初始时断路器处于closed状态，链路处于健康状态。当满足如下条件时，断路器从closed变成oped状态：

1. 周期内，总请求数超过一定量
2. 错误请求数占总请求数超过一定比例

绿线：断路器处于open状态，命令执行时，若当前时间超过断路器开启时间一定时间，断路器变成half\_open状态，尝试调用正常逻辑，只让一个请求通过根据执行是否成功，打开或者关闭熔断器（蓝线）

##### 解决灾难性雪崩效应：

降级、**隔离**、熔断、请求缓存、**请求合并**。

1）Hystrix使用命令模式HystrixCommand(Command)包装依赖调用逻辑，每个命令在单独线程中/信号 [授权](http://www.cxyclub.cn/Tag/shouquan.html" \t "https://blog.csdn.net/fight4gold/article/details/_blank) 下执行，每个command只能执行一次。

2）提供熔断器组件,可以自动运行或手动调用,停止当前请求一段时间(10秒)，熔断器默认 [错误](http://www.cxyclub.cn/Tag/cuowu.html" \t "https://blog.csdn.net/fight4gold/article/details/_blank) 率阈值为50%,超过将自动运行。

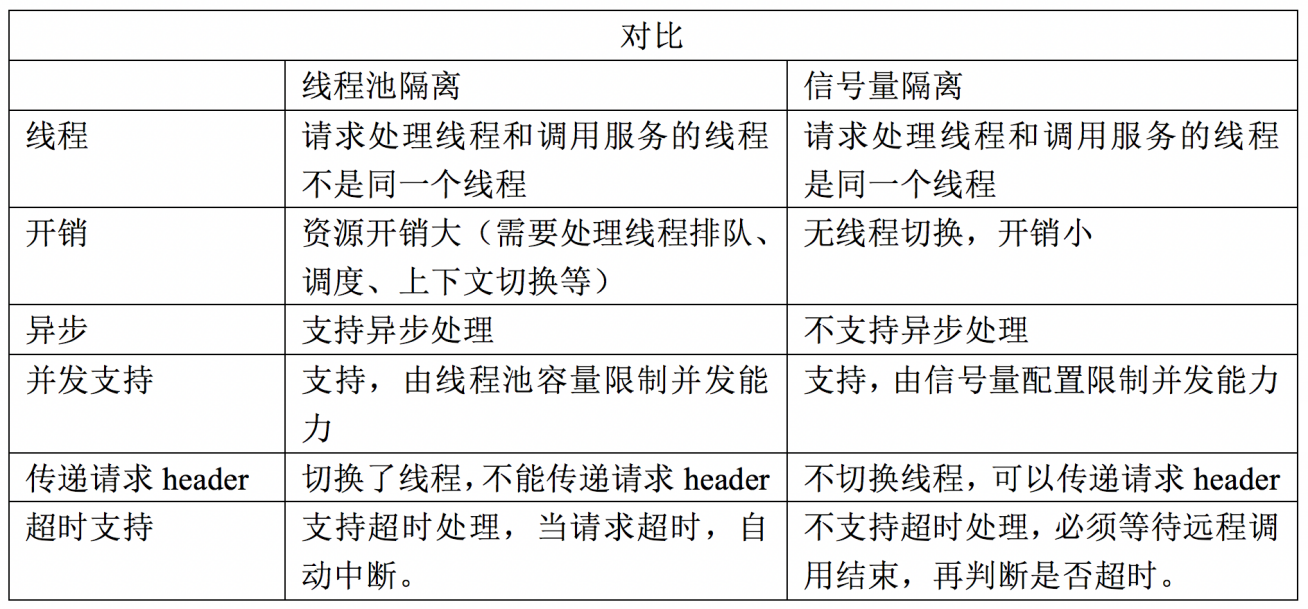
3）可配置依赖调用 [超时](http://www.cxyclub.cn/Tag/chaoshi.html" \t "https://blog.csdn.net/fight4gold/article/details/_blank) 时间,超时时间一般设为比99.5%平均时间略高即可.当调用超时时，直接返回或执行fallback逻辑。

4）为每个依赖提供一个小的线程池（或信号），如果线程池已满调用将被立即拒绝，默认不采用排队，从而加速失败判定时间。

5）依赖调用结果分:成功，失败（抛出 [异常](http://www.cxyclub.cn/Tag/yichang.html" \t "https://blog.csdn.net/fight4gold/article/details/_blank) ），超时，线程拒绝，短路。 请求失败(异常，拒绝，超时，短路)时执行fallback(降级)逻辑。

6）自我修复：断路器打开之后会自动进入**半开**状态。

##### 线程池隔离和信号量隔离的对比



* 线程池隔离：请求并发大，耗时较长（一般都是计算大，服务链长或访问数据库）时使用线程池隔离。可以尽可能保证外部容器（如Tomcat）线程池可用，不会因为服务调用的原因导致请求阻塞等待。
* 信号量隔离：请求并发大，耗时短（计算小，服务链段或访问缓存）时使用信号量隔离。因为这类服务的响应快，不会占用外部容器（如Tomcat）线程池太长时间，减少线程的切换，可以避免不必要的开销，提高服务处理效率。

#### Zuul：

1. 路由映射：
2. 鉴权：我们在用户登录成功之后会在返回头中添加cookie的值为openId=random(随机数)，并且将其保存在redis中，定义一个鉴权的过滤器，如果请求过来了，判断这个uri是否需要某种权限才能调用，如果不需要直接路由即可，如果需要那么判断cookie中是否有openId，如果没有表示没有登录，权限不够，如果有，需要判断和redis中的值是否相同，如果相同，表示有权限，直接路由到服务即可。
3. 限流：

https://blog.csdn.net/lizhengyu891231/article/details/104021737

1. 计数器法：维护一个单位时间内的计数器 Counter ，单位时间已经过去，则将计数器归零。 判断count 的计数值是否超过设定的频次，以及当前请求的时间和第一次请求时间是否在 1 分钟内。
2. 滑动窗口：假设每个时间窗口有 6 个格子，每个格子是 10 秒钟。每过 10 秒钟时间窗口向右移动一格，。我们为每个格子都设置一个独立的计数器 Counter，假如一个请求在 0:45 访问了那么我们将第五个格子的计数器 +1（也是就是 0:40~0:50 ），在判断限流的时候需要把所有格子的计数加起来和设定的频次进行比较即可。

1.漏桶算法：请求维护在队列中往桶中流，满了则拒绝，可以控制桶的出水速率，漏桶限制的是常量流出速率（一个固定常量值），所以最大的速率就是出水的速率，不能出现突发流量。

* 2.令牌桶算法：桶里存放令牌，固定速率往里面添加令牌，桶满了令牌被丢弃，请求来移除一个令牌，桶空了请求被丢弃，桶中令牌不足 N 个，则不会删除令牌，且请求将被限流（丢弃或阻塞等待），令牌桶限制的是平均流入速率（允许突发请求，只要有令牌就可以处理，支持一次拿3个令牌，4个令牌），并允许一定程度突发流量。

## Mq

### 消息确认

如果消息没有到exchange,则confirm回调,ack=false

如果消息到达exchange,则confirm回调,ack=true

exchange到queue成功,则不回调return

exchange到queue失败,则回调return(需设置mandatory=true,否则不回回调,消息就丢了)

<https://blog.csdn.net/qq315737546/article/details/54176560>

mq队列这一块，消费者重复接受消息怎么办，生产者发不过去怎么办

在redis存入一个消费状态自增键,每次消费前判断

### Kafka:

#### 特性：

https://book.51cto.com/art/201801/565243.htm

1. 快速持久化，可以在O(1)的系统开销下进行消息持久化；

线性写的速度约是随机写的6000多倍，现代的操作系统提供了预读（read-ahead）和延迟写（write-behind）技术，由于Kafka是基于JVM（Java Virtual Machine）的，而Java对象内存消耗非常高，且随着Java对象的增加JVM的垃圾回收也越来越频繁和繁琐，这些都加大了内存的消耗，鉴于以上因素，使用文件系统和依赖于页缓存（page cache）的存储比维护一个内存的存储或是应用其他结构来存储消息更有优势，因此Kafka选择以文件系统来存储数据。

1. 高吞吐，在一台普通的服务器上既可以达到10W/s的吞吐速率；

高吞吐量是Kafka设计的主要目标，Kafka将数据写到磁盘，充分利用磁盘的顺序读写。同时，Kafka在数据写入及数据同步采用了零拷贝（zero-copy）技术，采用sendFile()函数调用，sendFile()函数是在两个文件描述符之间直接传递数据，完全在内核中操作，从而避免了内核缓冲区与用户缓冲区之间数据的拷贝，操作效率极高。Kafka还支持数据压缩及批量发送，同时Kafka将每个主题划分为多个分区，这一系列的优化及实现方法使得Kafka具有很高的吞吐量。经大多数公司对Kafka应用的验证，Kafka支持每秒数\*\*\*别的消息。

3.完全的分布式系统，Broker、Producer、Consumer都原生自动支持分布式，自动实现负载均衡；Kafka依赖ZooKeeper来对集群进行协调管理，这样使得Kafka更加容易进行水平扩展，生产者、消费者和代理都为分布式，可配置多个。同时在机器扩展时无需将整个集群停机，集群能够自动感知，重新进行负责均衡及数据复制。

4.轻量级的消息系统，除了性能非常好之外，还是一个工作良好的分布式系统，Kafka的代理是无状态的，即代理不记录消息是否被消费，消费偏移量的管理交由消费者自己或组协调器来维护。同时集群本身几乎不需要生产者和消费者的状态信息，这就使得Kafka非常轻量级，同时生产者和消费者客户端实现也非常轻量级。

5.消息压缩

Kafka支持Gzip、Snappy、LZ4这3种压缩方式，通常把多条消息放在一起组成MessageSet，然后再把MessageSet放到一条消息里面去，从而提高压缩比率进而提高吞吐量。

6.数据备份

Kafka可以为每个主题指定副本数，对数据进行持久化备份，这可以一定程度上防止数据丢失，提高可用性。

7.多客户端支持。

Kafka核心模块用Scala语言开发，但Kafka支持不同语言开发生产者和消费者客户端应用程序，0.10之后的版本已废弃Scala语言实现的Producer及Consumer，默认使用Java版本的客户端，Kafka与当前主流的大数据框架都能很好地集成，如Flume、Hadoop、HBase、Hive、Spark、Storm等。

#### 优点

##### 解耦：

在项目启动之初来预测将来项目会碰到什么需求，是极其困难的。消息系统在处理过程中间插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口。这允许你独立的扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束。

##### 冗余：

有些情况下，处理数据的过程会失败。除非数据被持久化，否则将造成丢失。消息队列把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。许多消息队列所采用的"插入-获取-删除"范式中，在把一个消息从队列中删除之前，需要你的处理系统明确的指出该消息已经被处理完毕，从而确保你的数据被安全的保存直到你使用完毕。

##### 扩展性：

因为消息队列解耦了你的处理过程，所以增大消息入队和处理的频率是很容易的，只要另外增加处理过程即可。不需要改变代码、不需要调节参数。扩展就像调大电力按钮一样简单。

##### 灵活性&峰值处理能力：

在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见；如果为以能处理这类峰值访问为标准来投入资源随时待命无疑是巨大的浪费。使用消息队列能够使关键组件顶住突发的访问压力，而不会因为突发的超负荷的请求而完全崩溃。

##### 可恢复性：

系统的一部分组件失效时，不会影响到整个系统。消息队列降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入队列中的消息仍然可以在系统恢复后被处理。

##### 顺序保证：

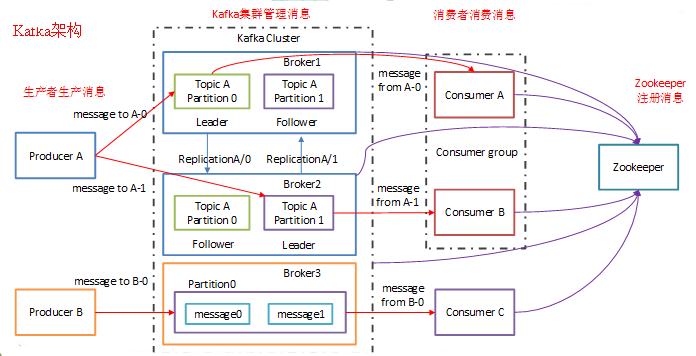
在大多使用场景下，数据处理的顺序都很重要。大部分消息队列本来就是排序的，并且能保证数据会按照特定的顺序来处理。Kafka保证一个Partition内的消息的有序性。

##### 缓冲:

在任何重要的系统中，都会有需要不同的处理时间的元素。例如，加载一张图片比应用过滤器花费更少的时间。消息队列通过一个缓冲层来帮助任务最高效率的执行———写入队列的处理会尽可能的快速。该缓冲有助于控制和优化数据流经过系统的速度。

##### 异步通信:

很多时候，用户不想也不需要立即处理消息。消息队列提供了异步处理机制，允许用户把一个消息放入队列，但并不立即处理它。想向队列中放入多少消息就放多少，然后在需要的时候再去处理它们。



### Broker

Kafka 集群包含一个或多个服务器，**服务器节点称为broker**

broker存储topic的数据。如果某topic有N个partition，集群有N个broker，那么每个broker存储该topic的一个partition。broker 是消息的代理，Producers往Brokers里面的指定Topic中写消息，Consumers从Brokers里面拉取指定Topic的消息，然后进行业务处理，broker在中间起到一个代理保存消息的中转

### Topic

每条发布到Kafka集群的消息都有一个类别，这个类别被称为Topic，（物理上不同Topic的消息分开存储，逻辑上一个Topic的消息虽然保存于一个或多个broker上但用户**只需指定消息的Topic**即可生产或消费数据而不必关心数据存于何处）

### Partition

topic中的数据分割为一个或多个partition。每个topic至少有一个partition。每个partition（物理是上存储这个partition所有消息和索引文件）中的数据使用多个segment文件存储。partition中的数据是有序的，不同partition间的数据丢失了数据的顺序。如果topic有多个partition，消费数据时就不能保证数据的顺序。在需要严格保证消息的消费顺序的场景下，需要将partition数目设为1。

**每个Partition只能被属于同一个Group的Consumer中的一个消费，而一个消费者可以消费多个Partition的数据，**

### Producer

生产者即数据的发布者，该角色将消息发布到Kafka的topic中。broker接收到生产者发送的消息后，broker将该消息追加到当前用于追加数据的segment文件中。生产者发送的消息，**存储到一个partition中**，生产者也可以指定数据存储的partition。

### Consumer

消费者可以从broker中读取数据。消费者可以消费多个topic中的数据。

### Consumer Group

每个Consumer属于一个特定的Consumer Group（可为每个Consumer指定group name，若不指定group name则属于默认的group）。这是kafka用来实现一个topic消息的广播（**发给所有的consume**r）和单播（发给任意一个consumer）的手段。一个**topic可以有多个CG**。topic的消息会复制-给consumer。如果需要实现广播，只要每个consumer有一个独立的CG就可以了。要实现单播只要所有的consumer在同一个CG。用CG还可以将consumer进行自由的分组而不需要多次发送消息到不同的topic。

### Leader

每个partition有多个副本，其中有且仅有一个作为Leader，Leader是当前负责数据的读写的partition。

### Follower

Follower跟随Leader，所有**写请求都通过Leader路由**，数据变更会**广播**给所有Follower，Follower与Leader保持数据同步。如果Leader失效，则从Follower中**选举**出一个新的Leader。当Follower与Leader**挂掉、卡住或者同步太慢**，leader会把这个follower从“in sync replicas”（ISR）列表中删除，重新创建一个Follower。

### Offset

kafka的存储文件都是按照offset.kafka来命名，用offset做名字的好处是方便查找。例如你想找位于2049的位置，只要找到2048.kafka的文件即可。当然the first offset就是00000000000.kafka

### Ack

* 1（默认） 数据发送到Kafka后，**经过leader成功接收消息的的确认**，就算是发送成功了。在这种情况下，如果leader宕机了，则会丢失数据。
* 0 生产者将数据发送出去就不管了，不去等待任何返回。这种情况下数据传输效率最高，但是数据可靠性确是最低的。
* -1 producer需要等待ISR中的所有follower都确认接收到数据后才算一次发送完成，可靠性最高。当ISR中所有Replica都向Leader发送ACK时，leader才commit，这时候producer才能认为一个请求中的消息都commit了。

### 消息删除

Kafka集群会保留所有的消息，无论其被消费与否。当然，因为磁盘限制，不可能永久保留所有数据（实际上也没必要），因此Kafka提供两种策略删除旧数据。一是**基于时间**，二是**基于Partition文件大小**。例如可以通过配置$KAFKA\_HOME/config/server.properties，让Kafka删除一周前的数据，也可在Partition文件超过1GB时删除旧数据

### follower如何与leader同步数据

异步复制方式下，Follower异步的从Leader复制数据，数据只要被Leader写入log就被认为已经commit，这种情况下，如果leader挂掉，会丢失数据，kafka使用ISR的方式很好的均衡了确保数据不丢失以及吞吐率。Follower可以**批量**的从Leader复制数据，而且Leader充分利用**磁盘顺序读**以及send file(zero copy)机制，这样极大的提高复制性能，内部**批量写磁盘**，大幅减少了Follower与Leader的消息量差。

### 什么情况下一个 broker 会从 isr中踢出去

leader会维护一个与其基本保持同步的Replica列表，该列表称为ISR(in-sync Replica)，每个Partition都会有一个ISR，而且是由leader动态维护 ，如果一个follower比一个leader落后太多，或者超过一定时间未发起数据复制请求，则leader将其重ISR中移除 。

## Linux

### 操作命令：

#### 目录切换

Cd / 切换到根目录

Cd ../ 切换到上级目录

Cd ~ 切换到home目录

Cd - 切换到上次访问的目录

#### 目录查看

Ls 查看当前目录下所有目录和文件

Ls -a 查看当前目录下所有目录和文件（包括隐藏文件）

Ls -l 或者ll  列表查看当前目录下的所有目录和文件（列表查看详细信息）

Ls /dir 查看指定目录下所有目录和文件

#### 创建目录

Mkdir aaa 当前目录下创建一aaa目录

Mkdir aaa bbb 当前目录下创建两个目录

Mkdir /usr/aaa 指定目录下创建一个aaa的目录

#### 删除目录或文件

删除文件：

Rm 文件 删除当前目录下的文件

Rm -f 文件 删除当前目录的文件(不询问)

删除目录：

Rm -r aaa 递归删除当前目录下的aaa目录

Rm -rf aaa 递归删除当前目录下的aaa目录（不询问）

全部删除

Rm -rf \* 将当前目录下所有文件和目录全部删除

Rm -rf /\* 将根目录下所有文件全部删除

#### 目录修改 mv和cp

##### 重命名目录：

Mv aaa bbb 将aaa重命名为bbb（可以使文件，目录压缩包）

##### 剪切目录：

mv /usr/tmp/aaa /usr 将/usr/tmp目录下的aaa目录剪切到 /usr目录下面

##### 拷贝目录：

cp -r 目录名称 目录拷贝的目标位置   -r代表递归

cp /usr/tmp/aaa  /usr 将/usr/tmp目录下的aaa目录复制到 /usr目录下面       
 注意：cp命令不仅可以拷贝目录还可以拷贝文件，压缩包等，拷贝文件和压缩包时不    用写-r递归

##### 搜索目录（或者文件）find：

find 目录 参数 文件名称

find /usr/tmp -name 'a\*' 查找/usr/tmp目录下的所有以a开头的目录或文件

find /home/user1 -name \\*.bin 在目录 '/ home/user1' 中搜索带有'.bin' 结尾的文件

find / -user user1 搜索属于用户 'user1' 的文件和目录

#### 文件操作

##### 新增文件 touch

Touch aaa.txt 当前目录创建一个aaa.txt的文件

##### 删除文件 mv

Rm -rf 文件名

##### 修改文件 vi vim

###### 命令行模式

1. 进入编辑模式

i:在光标所在字符前开始插入  
a:在光标所在字符后开始插入  
o:在光标所在行的下面另起一新行插入

1. 进入底行模式 ：
2. 查找 /字符
3. 删除当前行 dd

###### 编辑模式

Esc回到命令行模式

###### 底行模式

1. 退出 Q
2. 强制退出 q！
3. 保存并退出 wq

##### 文件查看

1. Cat 看最后一屏
2. More 百分比显示 回车向下一行 空格向下一页 q退出
3. Less 翻页查看 上下键翻页 q退出
4. Tail 指定行数或者动态查看

tail -10 sudo.conf 查看/etc/sudo.conf文件的后10行，Ctrl+C结束

#### Find

find [路径] [选项] [操作]

find ./ -name goods\* -print 当前目录下goods开头文件并打印

##### 选项：

-name 根据文件名查找

-perm 根据权限查找

-prune 排除某些查找目录

-user 根据文件属主查找

-mtime -n | +n 根据文件更改时间查找 +7表示七天以上

-size 根据文件大小查找

-type 根据文件类型查找 f表示文件

##### 操作：

###### -print 打印

###### -exec：

对搜索到的文件执行特定的操作，固定的格式为：-exec 'commond' {} \;   注意：{} 表示查询的结果。

举例1: 搜索 /etc 目录下的**文件（f）**（非目录），文件以 conf 结尾，且大于 10k，然后将其删除。

find /etc -type **f** -name '\*.conf' -size +10k -exec rm -f {} \;

举例2: 将 /data/log/ 目录下以 .log 结尾的文件，且更改时间在 **7 天以上**的删除。

find /data/log -name '\*.log' -mtime **+7** -exec rm -f \;

举例3: 搜索条件同 例1 一样，但是不删除，只是将其复制到 /root/conf 目录下

find /etc -type f -name '\*.conf' -size +10k -exec cp {} /root/conf/ \;

###### -ok：

和 -exec 的功能一样，只是每次操作都会给用户提示。

#### 根据关键字查找日志

cat -n goods.out | grep "SSLE" -C 10 |more

-n 显示行号

SSLE：关键字

-C 10 前后10行

-A（after） 后

-B 前

More：按空格翻页

#### [mysql：](https://www.cnblogs.com/ming-4/p/11690894.html)

* 登陆：mysql -uroot -p
* 查看mysql是否启动：service mysql status

#### 防火墙：

1:查看防火状态

systemctl status firewalld

service  iptables status

2:暂时关闭防火墙

systemctl stop firewalld

service  iptables stop

3:永久关闭防火墙

systemctl disable firewalld

chkconfig iptables off

4.停止并禁用开机启动

sytemctl disable firewall

#### 文件：

修改文件名：mv test.txt wbk.txt

创建文件夹： mkdir test

创建文件：touch test/a.txt

#### 启动服务：

nohup java -jar changgou-eureka.jar &

nohup java -jar changgou-eureka.jar >nohup.out 2>&1 &

nohup 意思是不挂断运行命令,当账户退出或终端关闭时,程序仍然运行，&代表在后台运行，nohup.out代表将日志打印在当前目录下nohup.out文件中

tail -fn 50 nohup.out：查看日志文件，-f表示尾部

停止：查询该项目的进程PID：ps aux|grep XXX （ps aux|grep java）  
结束进程：kill -9 PID

#### 环境变量：

vim /etc/profile

立即生效：source /etc/profile

#### 查看linux系统内存大小的详细信息

cat /proc/meminfo

#### 性能调优

https://blog.csdn.net/lvdeqing/article/details/54381716

##### Top

查看平均负载和CPU使用率

##### Vmstat

查看CPU上下文切换频次

vmstat 1 10 （数字表示命令执行频率和频次，1则每秒打印一次，执行10次）

Procs（进程）

| 字符列 | 描述 | 意义 |
| --- | --- | --- |
| r | 在就绪状态等待的进程数 | 如果这个数字超过CPU数量就要注意了！ |
| b | 在等待状态等待的进程数 | 说白了就是阻塞状态的进程数，比CPU数目高肯定有问题 |

Memory（内存）

| **字符列** | **描述** | **意义** |
| --- | --- | --- |
| swpd | 已使用虚拟内存大小 | 如果大于0就表示你的机器物理内存不足了（真的不足还是内存泄露？） |
| free | 空闲的物理内存的大小 |  |
| buff | 存储目录内容，权限等的缓存 |  |
| cache | cache直接用来存储打开文的缓冲 | sync后，使用echo 1(2,3) > /proc/sys/vm/drop\_caches分别进行释放吧! |

Swap（交换区）

| **字符列** | **描述** | **意义** |
| --- | --- | --- |
| si | 每秒从磁盘读入虚拟内存的大小 | 如果这个值大于0，表示物理内存不够用或者内存泄露了 |
| so | 每秒虚拟内存写入磁盘的大小 | 如果这个值大于0，同上。一般情况下，si、so的值都为0，如果si、so的值长期不为0，则表示系统内存不足，需要增加系统内存 |

IO（输入输出）

| **字符列** | **描述** | **意义** |
| --- | --- | --- |
| bi | 块设备每秒接收的块数量 | 这里的块设备是指系统上所有的磁盘和其他块设备 |
| bo | 块设备每秒发送的块数量 | 例如：我们读取文件，bo就要大于0。bi和bo一般都要接近0，不然就意味着IO操作过于频繁，需要进行调整。 |

System（系统）

| **字符列** | **描述** | **意义** |
| --- | --- | --- |
| in | 每秒CPU的中断次数，包括时间中断 | 中断太多了肯定有问题 |
| cs | 每秒上下文切换次数 | 这个越小越好。太大了，要考虑调低线程或者进程的数目,例如在apache服务器中，做性能测试时会进行几千并发甚至几万并发的测试，这时就要设置合适的进程数。系统层面的调用也是同样的道理。CS值过大表示上下文切换过于频繁，会消耗很多资源（CPU浪费在上下文切换没有充分利用） |

CPU

| **字符列** | **描述** | **意义** |
| --- | --- | --- |
| us | 用户CPU时间(百分比)。 |  |
| sy | 系统CPU时间(百分比) | 如果太高，表示系统调用时间长，例如是IO操作频繁。 us+sy的参考值为80% |
| id | 空闲CPU时间(百分比) | 一般来说，id + us + sy = 100 |
| wt | 等待IO的CPU时间(百分比) | 过大就意味着CPU和IO操作需要调整。 wt参考值35% |

**vmstat 可以用来确定一个系统的工作是受限于CPU还是受限于内存：如果CPU的sy和us值相加的百分比接近100%，或者运行队列(r) 中等待的进程数总是不等于 0，则该系统受限于CPU；如果pi、po的值总是不等于0，则该系统受限于内存。**

上下文切换次数发生的场景主要有如下几种：  
1）时间片用完，CPU 正常调度下一个任务；2）被其它优先级更高的任务抢占；3）执行任务碰到 I/O 阻塞，挂起当前任务，切换到下一个任务；4）用户代码主动挂起当前任务让出 CPU；5）多任务抢占资源，由于没有抢到被挂起；6）硬件中断。

##### Iostat

对系统的磁盘操作活动进行监视

%util: 一秒中有百分之多少的时间用于 I/O 操作，即被io消耗的cpu百分比  
备注：如果 %util 接近 100%，说明产生的I/O请求太多，I/O系统已经满负荷，该磁盘可能存在瓶颈。如果 svctm 比较接近 await，说明 I/O 几乎没有等待时间；如果 await 远大于 svctm，说明I/O 队列太长，io响应太慢，则需要进行必要优化。如果avgqu-sz比较大，也表示有当量io在等待。

##### 定位线程

top -Hp pid -H 可以显示进程下各线程的情况，其中可获取到有问题的线程id  
比如top -Hp 405，得知408线程cpu消耗大。  
通过printf “%x\n” 408，这个线程id转成16进制数198

jstack 405 | grep 198  
利用jstack命令获取线程的堆栈信息，看线程在哪里阻塞了。可定位到阻塞的对象，在通过代码查找，可定位到具体函数。

##### **[JVM调优之jstack找出最耗cpu的线程并定位代码](https://www.cnblogs.com/chengJAVA/p/5821218.html)**

https://www.cnblogs.com/chengJAVA/p/5821218.html

1.先找出Java进程ID 21711

ps -ef | grep changgou-goods | grep -v grep

2.找出该进程内最耗费CPU的线程

1）ps -Lfp pid

2）ps -mp pid -o THREAD, tid, time

3）top -Hp pid

用第三个

3.打印线程的16进制 54ee

printf "%x\n" 21742

1. 输出进程21711的堆栈信息，然后根据线程ID的十六进制值grep

jstack 21711 | grep 54ee

即可定位代码

## 虚拟机

### JVM参数设置

控制参数

* -Xms设置堆的最小空间大小。
* -Xmx设置堆的最大空间大小。
* -XX:NewSize设置新生代最小空间大小。
* -XX:MaxNewSize设置新生代最大空间大小。
* -XX:PermSize设置永久代最小空间大小。
* -XX:MaxPermSize设置永久代最大空间大小。
* -Xss设置每个线程的堆栈大小。

没有直接设置老年代的参数，但是可以设置堆空间大小和新生代空间大小两个参数来间接控制。

老年代空间大小=堆空间大小-年轻代大空间大小

### Java堆大小设置

Xms 和 Xmx设置为老年代存活对象的3-4倍，即FullGC之后的老年代内存占用的3-4倍  
永久代 PermSize和MaxPermSize(元空间)设置为老年代存活对象的1.2-1.5倍。  
年轻代Xmn的设置为老年代存活对象的1-1.5倍。  
老年代的内存大小设置为老年代存活对象的2-3倍。

如老年代Full Gc后为45M，则设置

-XX:MetaspaceSize=64M

-XX:MaxMetaspaceSize=64M

-Xms180m

-Xmx180m

### Java虚拟机中的内存模型？

方法区，虚拟机栈，本地方法栈，堆，程序计数器

问题分析：  
考官主要是对Java程序中各个变量的访问规则的考核。Java虚拟节内存空间分为方法区，Java堆，Java栈，本地方法栈。

核心答案讲解：  
Java虚拟机运行时内存所有的类的实例（不包括局部变量与方法参数）都存储在Java堆中，每条线程有自己的工作内存（Java栈），不同线程之间无法直接访问对方工作内存中的变量。方法区用于存储被虚拟机加载的类信息、常量、static变量等数据，堆用于存储对象实例，比如通过 new创建的对象实例就保存在堆中，堆中的对象的由垃圾回收器负责回收。Java栈用于实现方法调用，每次方法调用就对应栈中的一个栈帧，栈帧包含局部变量表、操作数栈、方法接口等于方法相关的信息，栈中的数据当没有引用指向数据时，这个数据就会消失。本地方法栈保存的是本地方法的调用。

问题扩展：  
线程安全问题就是，多个线程的工作内存同时对堆中同一个数据的修改，使用Java锁避免线程安全问题。

结合项目中使用：  
多线程消费同一个商品

### 谈谈你对垃圾回收机制的了解？

GC回收的是堆区和方法区的内存,当要进行垃圾回收时候，不管何种GC算法，除了垃圾回收的线程之外其他任何线程都将停止运行

问题解析：  
考官主要针对你对GC方面的考核，比如：什么是垃圾回收机制、JVM回收特点等。

核心答案解析  
什么是垃圾回收机制：在系统运行过程中，会产生一些无用的对象，这些对象占据着一定的内存，如果不对这些对象清理回收无用对象的内存，可能会导致内存的耗尽，所以垃圾回收机制回收的是内存。同时GC回收的是堆区和方法区的内存。  
JVM回收特点：(stop-the-world)当要进行垃圾回收时候，不管何种GC算法，除了垃圾回收的线程之外其他任何线程都将停止运行。被中断的任务将会在垃圾回收完成后恢复进行。GC不同算法或是GC调优就是减少stop-the-world的时间。à(为何非要stop-the-world)，就像是一个同学的聚会，地上有很多垃圾，你去打扫，边打扫边丢垃圾怎么都不可能打扫干净的哈。当在垃圾回收时候不暂停所有的程序，在垃圾回收时候有new一个新的对象B，此时对象A是可达B的，但是没有来及标记就把B当成无用的对象给清理掉了，这就会导致程序的运行会出现错误。

问题扩展：  
如何判断哪些对象需要回收呢：  
1.引用计数算法(java中不是使用此方法)：每个对象中添加一个引用计数器，当有别人引用它的时候，计数器就会加1，当别人不引用它的时候，计数器就会减1，当计数器为0的时候对象就可以当成垃圾。算法简单，但是最大问题就是在循环引用的时候不能够正确把对象当成垃圾。  
2.根搜索方法（这是后面垃圾搜集算法的基础）：这是JVM一般使用的算法，设立若干了根对象，当上述若干个跟对象对某一个对象都不可达的时候，这个对象就是无用的对象。对象所占的内存可以回收。  
根搜索算法的基础上，现代虚拟机的实现当中，垃圾搜集的算法主要有三种，分别是标记-清除算法、复制算法、标记-整理算法。

### 标记-清除算法

首先标记出需要回收的对象，然后在标记完成后对所标记的对象进行垃圾回收。

缺点：

1.效率执行不稳定，如果Java堆中存放的对象绝大多数都是要回收的，那么该算法需要进行大量的标记和清除动作，导致标记和清除两个过程随着对象数量的增长效率持续降低；

2.空间碎片化的问题，在进行标记—清除算法后，产生大量不连续的内存碎片，如果内存碎片过多，导致下一次为对象分配内存时找不到合适的内存大小存储对象，导致不得不提前触发一次垃圾收集动作。或者说对象分配内存时，找不到合适的对象造成大量内存浪费

### 标记-复制算法：

主要解决了标记—清除算法之后内存过于碎片化的问题，它是将可用的内存区域划分为大小相等的两块内存（当然也不一定非要相等），在进行垃圾收集时，现将存活的对象标记，然后将标记的存活对象复制到另一半空闲的内存区域，最后将之前的一半区域中所有对象完全清除用来存放之后新生的对象。这样就解决了使用标记—回收算法之后内存碎片化的问题，而且在对象内存分配时也极为简单，只需要移动堆顶指针，按顺序分配即可，简单高效。

缺点：

1.如果某块内存区域中在进行垃圾回收时，有大量的存活的对象，那么就会产生大量的对象复制的开销。

2.本来一块完整的可使用的内存区域被分为两份区域，一次只能使用其中一份区域，无疑极大的浪费了内存。

现在的商业虚拟机都采用这种算法来回收新生代，新生代中的对象 98% 都是「朝生夕死」，所以并不需要按照 1:1 的比例来划分空间，而是将内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor。当回收时，将 Eden 和 Survivor 中还存活的对象一次性复制到另一块 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 空间。

HotSpot 默认 Eden 和 Survivor 的大小比例是 **8:1**，也就是每次新生代中可用的内存为整个新生代容量的 90%（80%+10%），只有 10% 会被浪费。如果另外一块 Survivor 空间没有足够空间存放上一次新生代收集下来存活的对象时，这些对象将直接通过分配担保机制进入老年代。

### 标记-整理算法

适用于老年代的对象，存活时间久且就数量多

标记-整理算法的标记过程与「标记-清除」算法一样，但是后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

<https://www.cnblogs.com/code-duck/p/13573579.html>

### JVM内存模型

程序计数器(线程私有)：  
是当前线程锁执行字节码的行号治时期，每条线程都有一个独立的程序计数器，这类内存也称为“线程私有”的内存。正在执行java方法的话，计数器记录的是虚拟机字节码指令的地址(当前指令的地址)。如果是Natice方法，则为空。

java 虚拟机栈（线程私有）  
每个方法在执行的时候也会创建一个栈帧，存储了**局部变量**，操作数，动态链接，方法返回地址。每个方法从调用到执行完毕，对应一个栈帧在虚拟机栈中的入栈和出栈。通常所说的栈，一般是指在虚拟机栈中的局部变量部分。局部变量所需内存在编译期间完成分配，如果线程请求的栈深度大于虚拟机所允许的深度，则抛出StackOverflowError。如果虚拟机栈可以动态扩展，扩展到无法申请足够的内存，则抛出OutOfMemoryError。  
本地方法栈（线程私有）  
和虚拟机栈类似，主要为虚拟机使用到的**Native方法服务**。也会抛出StackOverflowError 和OutOfMemoryError。

Java堆（线程共享）  
被所有线程共享的一块内存区域，在虚拟机启动的时候创建，用于存放**对象实例**。  
对可以按照可扩展来实现（通过-Xmx 和-Xms 来控制）  
当队中没有内存可分配给实例，也无法再扩展时，则抛出OutOfMemoryError异常。  
方法区（线程共享）  
被所有方法线程共享的一块内存区域。  
用于存储已经被虚拟机加载的**类信息，常量，静态变量**等。  
这个区域的内存回收目标主要针对常量池的回收和堆类型的卸载。

### GC是什么时候触发的（面试最常见的问题之一）

  由于对象进行了分代处理，因此垃圾回收区域、时间也不一样。GC有两种类型：Scavenge GC和Full GC。

#### Scavenge GC

  一般情况下，当新对象生成，并且在Eden申请空间失败时，就会触发Scavenge GC，对Eden区域进行GC，清除非存活对象，并且把尚且存活的对象移动到Survivor区。然后整理Survivor的两个区。这种方式的GC是对年轻代的Eden区进行，不会影响到年老代。因为大部分对象都是从Eden区开始的，同时Eden区不会分配的很大，所以Eden区的GC会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使Eden去能尽快空闲出来。

#### Full GC

  对整个堆进行整理，包括Young、Tenured和Perm。Full GC因为需要对整个堆进行回收，所以比Scavenge GC要慢，因此应该尽可能减少Full GC的次数。在对JVM调优的过程中，很大一部分工作就是对于Full GC的调节。有如下原因可能导致Full GC：

a) 年老代（Tenured）被写满；

b) 持久代（Perm）被写满；

c) System.gc()被显示调用；

d) 上一次GC之后Heap的各域分配策略动态变化；

https://www.cnblogs.com/1024Community/p/honery.html#%E4%BA%94gc%E6%98%AF%E4%BB%80%E4%B9%88%E6%97%B6%E5%80%99%E8%A7%A6%E5%8F%91%E7%9A%84%E9%9D%A2%E8%AF%95%E6%9C%80%E5%B8%B8%E8%A7%81%E7%9A%84%E9%97%AE%E9%A2%98%E4%B9%8B%E4%B8%80

结合项目中使用：  
尽量不要new很大的object，大对象（>=85000Byte）直接归为G2代，GC回收算法从来不对大对象堆（LOH）进行内存压缩整理，因为在堆中下移85000字节或更大的内存块会浪费太多CPU时间；不要频繁的new生命周期很短object，这样频繁垃圾回收频繁压缩有可能会导致很多内存碎片，可以使用设计良好稳定运行的对象池（ObjectPool）技术来规避这种问题；之前在维护一个系统的时候，发现有很多大数据量的处理逻辑，但竟然都没有批量和分页处理，随着数据量的不断膨胀，隐藏的问题会不断暴露。然后我在重写的时候，都按照批量多次的思路设计实现，有了多线程、多进程和分布式集群技术，再大的数据量也能很好处理，而且性能不会下降，系统也会变得更加稳定可靠。

### 类加载机制有了解嘛？

问题分析：  
Java源代码被编译成class字节码，最终需要加载到虚拟机中才能运行。整个生命周期包括：加载、验证、准备、解析、初始化、使用和卸载7个阶段。

Class文件由类装载器装载后，在JVM中将形成一份描述Class结构的元信息对象，通过该元信息对象可以获知Class的结构信息：如构造函数，属性和方法等，Java允许用户借由这个Class相关的元信息对象间接调用Class对象的功能。  
虚拟机把描述类的数据从class文件加载到内存，并对数据进行校验，转换解析和初始化，最终形成可以被虚拟机直接使用的Java类型，这就是虚拟机的类加载机制。

核心答案讲解：  
类装载器就是寻找类的字节码文件，并构造出类在JVM内部表示的对象组件。在Java中，类装载器把一个类装入JVM中，要经过以下步骤：  
(1)装载：查找和导入Class文件；  
(2)链接：把类的二进制数据合并到JRE中；  
(a)校验：检查载入Class文件数据的正确性；  
(b)准备：给类的静态变量分配存储空间；  
(c)解析：将符号引用转成直接引用；  
(3)初始化：对类的静态变量，静态代码块执行初始化操作  
Java程序可以动态扩展是由运行期动态加载和动态链接实现的；比如：如果编写一个使用接口的应用程序，可以等到运行时再指定其实际的实现(多态)，解析过程有时候还可以在初始化之后执行；比如：动态绑定(多态)。

问题扩展  
双亲委派机制:父类加载了子类就不会加载了  
如果没有使用双亲委派模型，由各个类加载器自行加载的话，如果攻击者编写了一个称为java.lang.Object的类，并放在程序的ClassPath中，那系统将会出现多个不同的Object类， Java类型体系中最基础的行为就无法保证。应用程序也将会变得一片混乱。

结合项目中的使用

Java 类加载机制(阿里面试题)-何时初始化类  
[https://www.cnblogs.com/aspirant/p/7200523.html](https://www.cnblogs.com/aspirant/p/7200523.html" \t "https://www.showdoc.cc/_blank)

为什么要自定义Java类加载器  
[https://blog.csdn.net/u011490072/article/details/81560295](https://blog.csdn.net/u011490072/article/details/81560295" \t "https://www.showdoc.cc/_blank)

JVM类加载的那些事  
[https://www.jianshu.com/p/2133558b4735](https://www.jianshu.com/p/2133558b4735" \t "https://www.showdoc.cc/_blank)

## 事务

什么是事务？事务常见的并发问题及含义

问题分析：  
考官主要相对于事务方面的考核，如什么是事务，事务都有哪些特性；事务并发问题如何解决；事务的隔离级别分别可以解决什么问题；jdbc，spring如何实现事务控制；事务的应用场景等

核心答案讲解：  
事务是指逻辑上的一组操作，组成这组操作的一系列操作要么全部成功，要么一个都不做。因此，事务的结束有两种，当事务中的所有操作全部成功执行时，事务提交。如果其中一个操作失败，将发生回滚操作，撤消到事务开始时的状态。  
事务常见并发问题：  
丢失更新：撤消一个事务时，把其它事务已提交的更新的数据覆盖了。  
脏读：一个事务读到另一个事务未提交的更新数据。  
幻读：一个事务执行两次查询，但第二次查询比第一次查询多出了一些数据行。  
不可重复读：一个事务执行相同的查询两次或两次以上，但是每次都得到不同的数据。  
事务的特性：  
原子性（Atomicity）：事务作为一个整体被执行，包含在其中的对数据库的操作要么全部被执行，要么都不执行。

一致性（Consistency）：事务应确保数据库的状态从一个一致状态转变为另一个一致状态。一致状态的含义是数据库中的数据应满足完整性约束。

隔离性（Isolation）：多个事务并发执行时，一个事务的执行不应影响其他事务的执行。

持久性（Durability）：已被提交的事务对数据库的修改应该永久保存

事务的隔离级别及可以解决什么问题：

Serializable (串行化) 8 可避免脏读、不可重复读、幻读的发生。但是效率最低  
Repeatable read (可重复读) 4 可避免脏读、不可重复读的发生。（最常用的mysql默认的级别就是4）  
Read committed (读已提交) 2 可避免脏读的发生（oracle默认）。  
Read uncommitted (读未提交) 1 最低级别，任何情况都无法保证，但效率最高，最不安全。

### 事务的并发问题(脏读幻读等)

　　1、脏读（Read uncommitted）：一个事务读取到了另一个事务中尚未提交的数据,事务A读取了事务B更新的数据，然后B回滚操作，那么A读取到的数据是脏数据

　　2、不可重复读（Read committed）：一个事务中两次读取的数据内容不一致，要求的是一个事务中多次读取时数据是一致的，这是事务update时引发的问题: 事务 A 多次读取同一数据，事务 B 在事务A多次读取的过程中，对数据作了更新并提交，导致事务A多次读取同一数据时，结果 不一致。

　　3、幻读（Repeatable read）：一个事务中两次读取的数据的数量不一致，要求在一个事务多次读取的数据的数量是一致的，这是insert或delete时引发的问题: 系统管理员A将数据库中所有学生的成绩从具体分数改为ABCDE等级，但是系统管理员B就在这个时候插入了一条具体分数的记录，当系统管理员A改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

　　小结：不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表

锁  
怎样实现事务的隔离呢？  
隔离机制的实现必须使用锁。  
锁的基本原理  
a.当一个事务访问某个数据库资源时，如果执行的是select语句，必须为资源加上共享锁，如果执行的是insert,update,delete语句，必须为资源加上排他锁，这些锁锁定正在被操作的资源。  
b.当第二个事务也要访问相同的资源时，如果执行的select语句，那么也必须为资源加上共享锁；如果执行的是insert,update,或delete语句，也必须为资源加上排他锁。但此时第二个事务并非就立即能为资源加上锁，当第一个事务为资源加的是共享锁时，第二个事务能够为资源加上共享锁，但当第一个事务为资源加的是排他锁时，第二个事务必须等待第一个事务结束，才能为资源加上排他锁。  
1.共享锁（s锁）

共享锁用于读取数据操作，它允许其他事务同时读取锁定的资源，但不允许其他事务更新它。  
2.排他锁（X锁）  
排他锁用于修改数据的场合，他锁定的资源，其他事务部能读取也不能修改。  
3.更新锁（U锁）  
更新锁在更新操作初始化截断用来锁定可能要被修改的资源，从而避免使用共享锁造成的死锁现象。

表锁（table lock）  
表锁是MySQL中最基本的锁策略，并且是开销最小的策略。表锁会锁定整张表。一个用户在对表进行写操作（插入、删除、更新等）前，需要先获得写锁，这会阻塞其他用户对该表的所有读写操作。当没有写锁时，其他读取的用户才能获得读锁，读锁之间是不相互阻塞的。

行级锁（row lock）  
行级锁可以最大程度地支持并发处理（同时也带来了最大的锁开销）。在InnoDB和XtraDB，以及其他一些存储引擎中实现了行级锁。行级锁只在存储引擎层实现，而MySQL服务器层没有实现。服务器层完全不了解存储引擎中的锁实现。

一次封锁or两段锁？  
因为有大量的并发访问，为了预防死锁，一般应用中推荐使用一次封锁法，就是在方法的开始阶段，已经预先知道会用到哪些数据，然后全部锁住，在方法运行之后，再全部解锁。这种方式可以有效的避免循环死锁，但在数据库中却不适用，因为在事务开始阶段，数据库并不知道会用到哪些数据。  
数据库遵循的是两段锁协议，将事务分成两个阶段，加锁阶段和解锁阶段（所以叫两段锁）

加锁阶段：在该阶段可以进行加锁操作。在对任何数据进行读操作之前要申请并获得S锁（共享锁，其它事务可以继续加共享锁，但不能加排它锁），在进行写操作之前要申请并获得X锁（排它锁，其它事务不能再获得任何锁）。加锁不成功，则事务进入等待状态，直到加锁成功才继续执行。

解锁阶段：当事务释放了一个封锁以后，事务进入解锁阶段，在该阶段只能进行解锁操作不能再进行加锁操作。

悲观锁  
悲观锁是指假设并发更新冲突会发生，所以不管冲突是否真的发生，都会使用锁机制。悲观锁会完成以下功能：锁住读取的记录，防止其它事务读取和更新这些记录。其它事务会一直阻塞，直到这个事务结束.悲观锁是在使用了数据库的事务隔离功能的基础上，独享占用的资源，以此保证读取数据一致性，避免修改丢失。悲观锁可以使用Repeatable Read事务，它完全满足悲观锁的要求。

乐观锁  
乐观锁不会锁住任何东西，也就是说，它不依赖数据库的事务机制，乐观锁完全是应用系统层面的东西。如果使用乐观锁，大多是基于数据版本（ Version ）记录机制实现。何谓数据版本？即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个 “version” 字段来实现。读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

结合项目中使用：  
 spring 的申明式事务：使得我们再也无需要去处理获得连接、关闭连接、事务提交和回滚等这些操作。再也无需要我们在与事务相关的方法中处理大量的try…catch…finally代码。

### 事务的传播行为

A方法调用了B,当前方法指A,当前事务指A开启的事务,该方法指B

PROPAGION\_XXX：事务的传播行为

\*保证同一个事务中

PROPAGATION\_REQUIRED支持当前事务，如果不存在，就新建一个（默认）

PROPAGATION\_SUPPORTS支持当前事务，如果不存在，就不适用事务

PROPAGATION\_MANDATORY 支持当前事务，如果不存在，抛出异常

\*保证没有在同一个事务中

PROPAGATION\_REQUIRES\_NEW如果有事务存在，挂起当前事务，创建一个新的事务

PROPAGATION\_NOT\_SUPPORTED 以非事务方式运行，如果有事务存在，挂起当前事务

PROPAGATION\_NEVER 以非事务方式运行，如果有事务存在，抛出异常

PROPAGATION\_NESTED 如果当前事务存在，则嵌套事务执行(外层事务失败回滚所有,内层事务失败只回滚内层)

原文链接：https://blog.csdn.net/nwj1989/article/details/80549760

<https://blog.csdn.net/weixin_39625809/article/details/80707695>

### 分布式架构如何保证数据一致性?

## 网络协议

|  |
| --- |
| 应用层 |
| 表示层 |
| 会话层 |
| 传输层 |
| 网络层 |
| 数据链路层 |
| 物理层 |

这个和邮局寄信是一个概念。应用层就是写信，并且放入信封。传输层就是你把信件投到邮局，网络层就是邮局把你的邮件投递到对方所属的中心邮局，链路层就是对方中心邮局分发到所属的小邮政点，物理层就是绿衣使者骑着自行车把信送到您的手上。

[物理层](https://baike.baidu.com/item/%E7%89%A9%E7%90%86%E5%B1%82" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank)：[以太网](https://baike.baidu.com/item/%E4%BB%A5%E5%A4%AA%E7%BD%91" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank) · [调制解调器](https://baike.baidu.com/item/%E8%B0%83%E5%88%B6%E8%A7%A3%E8%B0%83%E5%99%A8" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank) · 电力线通信(PLC) · [SONET/SDH](https://baike.baidu.com/item/SONET/SDH" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank) · G.709 · [光导纤维](https://baike.baidu.com/item/%E5%85%89%E5%AF%BC%E7%BA%A4%E7%BB%B4" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank) · 同轴电缆 · 双绞线等

**数据链路层**：Wi-Fi([IEEE 802.11](https://baike.baidu.com/item/IEEE%20802.11" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank)) · WiMAX([IEEE 802.16](https://baike.baidu.com/item/IEEE%20802.16" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank)) ·ATM · DTM · [令牌环](https://baike.baidu.com/item/%E4%BB%A4%E7%89%8C%E7%8E%AF" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank) · [以太网](https://baike.baidu.com/item/%E4%BB%A5%E5%A4%AA%E7%BD%91" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank) ·FDDI · [帧中继](https://baike.baidu.com/item/%E5%B8%A7%E4%B8%AD%E7%BB%A7" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank) · GPRS · EVDO ·HSPA · HDLC · [PPP](https://baike.baidu.com/item/PPP/6660214" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank) · L2TP ·PPTP · ISDN·STP · CSMA/CD等

**[网络层](https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%B1%82" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank)协议**：IP (IPv4 · IPv6) · ICMP· ICMPv6·IGMP ·IS-IS · IPsec · ARP · RARP · RIP等

**[传输层](https://baike.baidu.com/item/%E4%BC%A0%E8%BE%93%E5%B1%82" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank)协议**：TCP · UDP · TLS · [DCCP](https://baike.baidu.com/item/DCCP" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank) · SCTP · RSVP · OSPF 等

**[应用层](https://baike.baidu.com/item/%E5%BA%94%E7%94%A8%E5%B1%82" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank)协议**：DHCP ·[DNS](https://baike.baidu.com/item/DNS" \t "https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E5%8D%8F%E8%AE%AE/_blank) · FTP · Gopher · HTTP· IMAP4 · IRC · NNTP · XMPP ·POP3 · SIP · SMTP ·SNMP · SSH ·TELNET RPC · RTCP · RTP ·RTSP· SDP · SOAP · GTP · STUN · NTP· SSDP · BGP 等

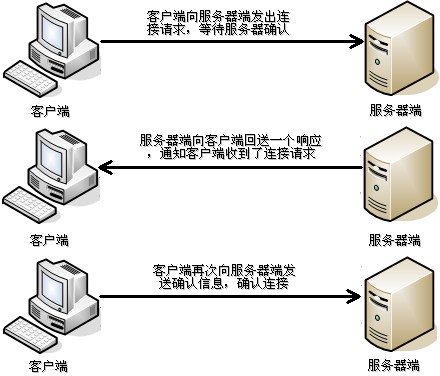
### HTTP（传输数据，应用层）

#### 长连接

HTTP1.1规定了默认保持长连接（HTTP persistent connection ，也有翻译为持久连接），**数据传输完成了保持TCP连接不断开（不发RST包、不四次握手），等待在同域名下继续用这个通道传输数据**；相反的就是短连接。

http是基于TCP连接基础上的，HTTP连接使用的是“请求—响应”的方式，不仅在请求时需要先建立连接，而且需要客户端向服务器发出请求后，服务器端才能回复数据。

### TCP（单纯建立连接，传输层）



#### 为何是三次连接而不是两次连接

一句话，主要防止已经失效的连接请求报文突然又传送到了服务器，从而产生错误。

如果使用的是两次握手建立连接，假设有这样一种场景，客户端发送了第一个请求连接并且没有丢失，只是因为在网络结点中滞留的时间太长了，由于TCP的客户端迟迟没有收到确认报文，以为服务器没有收到，此时重新向服务器发送这条报文，此后客户端和服务器经过两次握手完成连接，传输数据，然后关闭连接。此时此前滞留的那一次请求连接，网络通畅了到达了服务器，这个报文本该是失效的，但是，两次握手的机制将会让客户端和服务器再次建立连接，这将导致不必要的错误和资源的浪费。

如果采用的是三次握手，就算是那一次失效的报文传送过来了，服务端接受到了那条失效报文并且回复了确认报文，但是客户端不会再次发出确认。由于服务器收不到确认，就知道客户端并没有请求连接。

https://blog.csdn.net/qzcsu/article/details/72861891

#### TCP连接的释放（四次挥手）

1. 客户端发送FIN=1表示我要关闭连接了表示不再发送数据
2. 服务器发送ACK=表示我收到了(之后会将剩余数据发送完)
3. 服务器发送FIN=1表示我要关闭连接了
4. 客户端收到发送ACk=1表示收到然后关闭连接服务器收到后关闭连接

#### 为什么建立连接是三次握手，关闭连接确是四次挥手呢？

建立连接的时候， 服务器在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。

而关闭连接时，服务器收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，而自己也未必全部数据都发送给对方了，所以己方可以立即关闭，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送，从而导致多了一次。

#### 如果已经建立了连接，但是客户端突然出现故障了怎么办

TCP还设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75秒发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

原文链接：<https://blog.csdn.net/qzcsu/article/details/72861891>

### Servlet和Socket

Socket是j2se在网络编程这块最基本的东西。把一系列网络io复杂性封装。但是光有io，光能传输数据，不足以完成标准化的网络通信。所以在这个基础上，加入了协议支持。比如web容器(tomcat等)加入了http协议的解析(所谓的协议，就是一系列带有标准格式的字符串)。而把里面可以独立给程序员的模型接口抽取出来，就是Servlet。所以，Servlet可以看做是web容器运行的一部分逻辑(请求和响应，即request和response其实可以看成是dto，最后执行完servlet.service方法后，由容器传递给socket类的outputstream，完成相关操作)。而Socket则是web容器的核心(负责交换协议)。

### 协议总结

处理请求: servlet(tomcat)-->socket(jdk)

发送请求:httpclient(框架)-->HttpUrlConnection(jdk)-->socket(jdk)

<https://blog.csdn.net/Hknock/article/details/74332402>

### HttpUrlConnection底层实现

<http://blog.csdn.net/zhongweijian/article/details/7619453>

## 排序算法

### 冒泡排序：

比较相邻的数据大小，顺序错误就把它们交换过来，重复地进行直到没有再需要交换

### 选择排序:

找到最小的数据放第一位，交换位置，剩余未排序数据中找到最小的放第二位，交换位置，直到所有元素均排序完毕

### 快速排序：

选基准书如第一个数temp，从后往前找到一个比该数小的数a放到temp处，从前往后找到一个比该数大的交换位置放到a处，直到相遇，交换基准数temp与相遇的值，分两个区再做此直到所有的排序玩成

### 计数排序：

找出待排序的数组中最大和最小的元素，统计数组中每个值为i的元素出现的次数，存入数组C的第i项；

对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）；

反向填充目标数组：将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1。

### 归并排序

分为n个子序列，每个子序列排好序，比较子序列第一个值拿出来放到第一位，再比较第二个值与其他的第一位，拿出来放在第二位，以此类推

## 对称加密与非对称加密

base64: 是一种编码方式(加密解密)

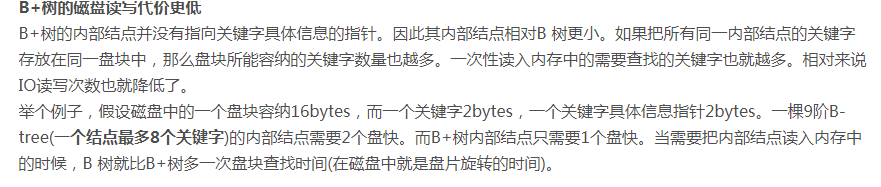
Bcrypt: 通过Bcrypt加密的明文密码即使解密也不是真的“解密”，哪怕是内部人员，也不会看到密码

对称加密:[DES](https://baike.baidu.com/item/DES" \t "https://baike.baidu.com/item/_blank)、[3DES](https://baike.baidu.com/item/3DES" \t "https://baike.baidu.com/item/_blank)、TDEA、[Blowfish](https://baike.baidu.com/item/Blowfish" \t "https://baike.baidu.com/item/_blank)、RC2、RC4、[RC5](https://baike.baidu.com/item/RC5" \t "https://baike.baidu.com/item/_blank)、[IDEA](https://baike.baidu.com/item/IDEA" \t "https://baike.baidu.com/item/_blank)、SKIPJACK

非对称加密:[RSA](https://baike.baidu.com/item/RSA" \t "https://baike.baidu.com/item/%E9%9D%9E%E5%AF%B9%E7%A7%B0%E5%8A%A0%E5%AF%86%E7%AE%97%E6%B3%95/_blank)、[Elgamal](https://baike.baidu.com/item/Elgamal" \t "https://baike.baidu.com/item/%E9%9D%9E%E5%AF%B9%E7%A7%B0%E5%8A%A0%E5%AF%86%E7%AE%97%E6%B3%95/_blank)、背包算法、Rabin、D-H、[ECC](https://baike.baidu.com/item/ECC" \t "https://baike.baidu.com/item/%E9%9D%9E%E5%AF%B9%E7%A7%B0%E5%8A%A0%E5%AF%86%E7%AE%97%E6%B3%95/_blank)（椭圆曲线加密算法）

公钥加密|私钥解密用于防止密文被破解、被第三方得到明文；私钥加密|公钥解密用于防止明文被篡改，确保消息的完整性和正确的发送方。那么既然为了防止明文被篡改，我们是不是直接都可以用公钥加密的方式呢，这样整串都是密文了，其实当然也可以，只不过签名的效率要高的多，而非对称加解密很费时间，所有对于不值得加密的非关键性数据，还是用签名合适

## B树、B-树、B+树与红黑树

内部节点：除了根节点和叶子节点的节点

## 什么是上下文切换

<http://www.mamicode.com/info-detail-2789756.html>

## BIO NIO的区别

<https://blog.csdn.net/weixin_42133940/article/details/88073132>

BIO：同步阻塞式IO，服务器实现模式为一个连接建立一个线程

单线程：只能等待同当前连接的客户端的操作执行完成再执行其他的请求

多线程：为每个客户端请求都单独创建一个线程来单独处理

NIO：同步非阻塞式IO，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器（Selector）上，多路复用器论询到连接有IO请求时才启动一个线程进行处理，NIO和BIO最大的区别就是只需要开启一个线程就可以处理来自多个客户端的IO事件

AIO：异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的IO请求都是由OS先完成了再通知服务器应用区启动线程进行处理

### 应用场景

BIO方式适用于连接数目比较小且固定的场景，这种方式对服务器资源要求比较高，并发局限于应用中

NIO适合处理连接数目特别多，但是连接比较短小的场景，Jetty，Mina，Zookeeper等都是基于java nio实现

AIO方式使用于连接数目比较多且比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂