# 1. 基础篇

## 1.1. Java基础

### 面向对象的特征：继承、封装和多态

多态体现: 多态有四种体现形式：

1.接口和接口的继承。

2.类和类的继承。

3.重载。

4.重写。

### final, finally, finalize 的区别

### Exception、Error、运行时异常与一般异常有何异同

Throwable

/ \

Error Exception

/ / \

xxxxxx xxxxxx RuntimeException

### 请写出5种常见到的runtime exception

### <https://www.cnblogs.com/zhaoleigege/p/5809798.html>

NullPointerException - 空指针引用异常

ClassCastException - 类型强制转换异常。

IllegalArgumentException - 传递非法参数异常。

ArithmeticException - 算术运算异常

IndexOutOfBoundsException - 下标越界异常

NegativeArraySizeException - 创建一个大小为负数的数组错误异常

NumberFormatException - 数字格式异常

SecurityException - 安全异常

UnsupportedOperationException - 不支持的操作异常

### int 和 Integer 有什么区别，Integer的值缓存范围

<https://www.cnblogs.com/zwxblog/p/7788573.html>

### 包装类，装箱和拆箱

### <https://www.cnblogs.com/wang-yaz/p/8516151.html>

当一个基础数据类型与封装类进行==、+、-、\*、/运算时，会将封装类进行拆箱，对基础数据类型进行运算。

它必须满足两个条件才为true：   
1、类型相同   
2、内容相同

Integer integer100=null;

int int100=integer100;

### 这两行代码是完全合法的，完全能够通过编译的，但是在运行时，就会抛出空指针异常。其中，integer100为Integer类型的对象，它当然可以指向null。但在第二行时，就会对integer100进行拆箱，也就是对一个null对象执行intValue()方法，当然会抛出空指针异常。所以，有拆箱操作时一定要特别注意封装类对象是否为null。

### String、StringBuilder、StringBuffer

### 重载和重写的区别

### 抽象类和接口有什么区别

### 说说反射的用途及实现

<https://blog.csdn.net/snn1410/article/details/44978457>

* 什么是反射机制？
* 简单说，反射机制值得是程序在运行时能够获取自身的信息。在java中，只要给定类的名字，那么就可以通过反射机制来获得类的所有信息。

<https://www.jianshu.com/p/d6035d5d4d12>

Java反射框架提供以下功能：

①、在运行时判断任意一个对象所属的类  
②、在运行时构造任意一个类的对象  
③、在运行时判断任意一个类所具有的成员变量和方法（通过反射设置可以调用 private）  
④、在运行时调用人一个对象的方法

**可以通过配置文件来动态配置和加载类**，以实现**软件工程理论**里所提及的类与类，模块与模块之间的**解耦**。反射最经典的应用是spring框架。

**最重要的用途就是开发各种通用框架。**

### 说说自定义注解的场景及实现

<https://www.cnblogs.com/liangweiping/p/3837332.html>

1.注解的定义：Java文件叫做Annotation，用@interface表示。

2.元注解：@interface上面按需要注解上一些东西，包括@Retention、@Target、@Document、@Inherited四种。

3.注解的保留策略：

　　@Retention(RetentionPolicy.SOURCE)   // 注解仅存在于源码中，在class字节码文件中不包含

　　@Retention(RetentionPolicy.CLASS)     // 默认的保留策略，注解会在class字节码文件中存在，但运行时无法获得

　　@Retention(RetentionPolicy.RUNTIME)  // 注解会在class字节码文件中存在，在运行时可以通过反射获取到

4.注解的作用目标：

　　@Target(ElementType.TYPE)                      // 接口、类、枚举、注解

　　@Target(ElementType.FIELD)                     // 字段、枚举的常量

　　@Target(ElementType.METHOD)                 // 方法

　　@Target(ElementType.PARAMETER)            // 方法参数

　　@Target(ElementType.CONSTRUCTOR)       // 构造函数

　　@Target(ElementType.LOCAL\_VARIABLE)   // 局部变量

　　@Target(ElementType.ANNOTATION\_TYPE) // 注解

　　@Target(ElementType.PACKAGE)               // 包

5.注解包含在javadoc中：

　　@Documented

6.注解可以被继承：

　　@Inherited

7.注解解析器：用来解析自定义注解。

### HTTP请求的GET与POST方式的区别

### Session与Cookie区别

### 列出自己常用的JDK包

### MVC设计思想

### equals与==的区别

### hashCode和equals方法的区别与联系

### 什么是Java序列化和反序列化，如何实现Java序列化？或者请解释Serializable 接口的作用

* **What---什么是序列化和反序列化：**
* Serialization-序列化：可以看做是将一个对象转化为二进制流的过程
* Deserialization-反序列化：可以看做是将对象的二进制流重新读取转换成对象的过程

若类仅仅实现了Serializable接口，则可以按照以下方式进行序列化和反序列化  
ObjectOutputStream采用默认的序列化方式，对对象的非transient的实例变量进行序列化。  
ObjcetInputStream采用默认的反序列化方式，对对象的非transient的实例变量进行反序列化。

情况2：若类不仅实现了Serializable接口，并且还定义了readObject(ObjectInputStream in)和writeObject(ObjectOutputSteam out)，则采用以下方式进行序列化与反序列化。  
*ObjectOutputStream调用对象的writeObject(ObjectOutputStream out)的方法进行序列化。*  
*ObjectInputStream会调用对象的readObject(ObjectInputStream in)的方法进行反序列化。*

情况3：若类实现了Externalnalizable接口，且类必须实现readExternal(ObjectInput in)和writeExternal(ObjectOutput out)方法，则按照以下方式进行序列化与反序列化。  
*ObjectOutputStream调用对象的writeExternal(ObjectOutput out))的方法进行序列化。*  
*ObjectInputStream会调用对象的readExternal(ObjectInput in)的方法进行反序列化。*

a. 被transient和static修饰的成员变量不会被序列化

b. 有个需要注意的点来自 Serializable 接口的说明文档，简单说明如下：

假设A实现了Serializable接口，且A为B的子类，B没有实现Serializable接口。那么在序列化和反序列话的时候，B的无参构造函数负责B的相关属性的序列化和反序列化。特殊的，当B没有无参构造函数的时候，将A对象进行序列化时不会报错，但是反序列化获取A的时候报错。

序列化算法一般会按步骤做如下事情：

◆将对象实例相关的类元数据输出。

◆递归地输出类的超类描述直到不再有超类。

◆类元数据完了以后，开始从最顶层的超类开始输出对象实例的实际数据值。

◆从上至下递归输出实例的数据

<https://blog.csdn.net/onceing/article/details/80969369>

|  |
| --- |
| 1、序列化时，只对对象的状态进行保存，而不管对象的方法；  2、当一个父类实现序列化，子类自动实现序列化，不需要显式实现Serializable接口；  3、当一个对象的实例变量引用其他对象，序列化该对象时也把引用对象进行序列化；  4、并非所有的对象都可以序列化，至于为什么不可以，有很多原因了，比如：  安全方面的原因，比如一个对象拥有private，public等field，对于一个要传输的对象，比如写到文件，或者进行RMI传输等等，在序列化进行传输的过程中，这个对象的private等域是不受保护的；  资源分配方面的原因，比如socket，thread类，如果可以序列化，进行传输或者保存，也无法对他们进行重新的资源分配，而且，也是没有必要这样实现；  5、声明为static和transient类型的成员数据不能被序列化。因为static代表类的状态，transient代表对象的临时数据。  6、序列化运行时使用一个称为 serialVersionUID 的版本号与每个可序列化类相关联，该序列号在反序列化过程中用于验证序列化对象的发送者和接收者是否为该对象加载了与序列化兼容的类。为它赋予明确的值。显式地定义serialVersionUID有两种用途：  在某些场合，希望类的不同版本对序列化兼容，因此需要确保类的不同版本具有相同的serialVersionUID；  在某些场合，不希望类的不同版本对序列化兼容，因此需要确保类的不同版本具有不同的serialVersionUID。  7、Java有很多基础类已经实现了serializable接口，比如String,Vector等。但是比如HashTable就没有实现serializable接口；  8、如果一个对象的成员变量是一个对象，那么这个对象的数据成员也会被保存！这是能用序列化解决深拷贝的重要原因； |

* Object类中常见的方法，为什么wait  notify会放在Object里边？
* **为什么这些操作线程的方法要定义在object类中呢？**
* 简单说：因为synchronized中的这把锁可以是任意对象，所以任意对象都可以调用wait()和notify()；所以wait和notify属于Object。
* 专业说：因为这些方法在操作同步线程时，都必须要标识它们操作线程的锁，只有同一个锁上的被等待线程，可以被同一个锁上的notify唤醒，不可以对不同锁中的线程进行唤醒。
* 也就是说，等待和唤醒必须是同一个锁。而锁可以是任意对象，所以可以被任意对象调用的方法是定义在object类中。

### 线程之间通信

<https://www.cnblogs.com/hapjin/p/5492619.html>

1. **同步**
2. **while轮询的方式**
3. **wait/notify机制**
4. **管道通信**就是使用java.io.PipedInputStream 和 java.io.PipedOutputStream进行通信

### Java的平台无关性如何体现出来的

### JDK和JRE的区别

### Java 8有哪些新特性

<http://www.runoob.com/java/java8-new-features.html>

* **Lambda 表达式** − Lambda允许把函数作为一个方法的参数（函数作为参数传递进方法中。
* **方法引用** − 方法引用提供了非常有用的语法，可以直接引用已有Java类或对象（实例）的方法或构造器。与lambda联合使用，方法引用可以使语言的构造更紧凑简洁，减少冗余代码。
* **默认方法** − 默认方法就是一个在接口里面有了一个实现的方法。
* **新工具** − 新的编译工具，如：Nashorn引擎 jjs、 类依赖分析器jdeps。
* **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。
* **Date Time API** − 加强对日期与时间的处理。
* **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。
* **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。

## Java常见集合

### List 和 Set 区别

### Set和hashCode以及equals方法的联系

### List 和 Map 区别

### Arraylist 与 LinkedList 区别

### ArrayList 与 Vector 区别

### HashMap 和 Hashtable 的区别

### HashSet 和 HashMap 区别

### HashMap 和 ConcurrentHashMap 的区别

HashMap不是线程安全的，ConcurrentHashMap则在某一个方法的执行上是线程安全的。

1. 数据结构

都是数组+拉链实现的哈希表，但是具体实现上差别大了

2. 并发

Hashtable全表锁

HashMap多线程不安全，需要自己封装

ConcurrentHashMap加细粒度锁，读不加锁，如果读到空值再加锁。注意这三个关键字final,transient,volatile

3. null

Hashtable不允许用 null作为键和值

HashMap允许全局最多一个null键，但是允许无数个null值

ConcurrentHashMap不允许用 null作为键和值

### HashMap 的工作原理及代码实现，什么时候用到红黑树

链表程度大于 8

### 多线程情况下HashMap死循环的问题

### HashMap出现Hash DOS攻击的问题

### ConcurrentHashMap 的工作原理及代码实现，如何统计所有的元素个数

<https://blog.csdn.net/u012129558/article/details/52268194>

**不lock所有的Segment，遍历所有Segment，累加各个Segment的大小得到整个Map的大小，如果某相邻的两次计算获取的所有Segment的更新的次数是一样的，说明计算过程中没有更新操作，则直接返回这个值。如果这三次不加锁的计算过程中Map的更新次数有变化，则之后的计算先对所有的Segment加锁，再遍历所有Segment计算Map大小，最后再解锁所有Segment**

### 手写简单的HashMap

### 看过那些Java集合类的源码

### 进程和线程

* 线程和进程的概念、并行和并发的概念
* 创建线程的方式及实现
* 进程间通信的方式

1. 管道pipe：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

2. 命名管道FIFO：有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。

4. 消息队列MessageQueue：消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

5. 共享存储SharedMemory：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。

6. 信号量Semaphore：信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。

7. 套接字Socket：套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。

8. 信号 ( sinal ) ： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。

### 说说 CountDownLatch、CyclicBarrier 原理和区别

CountDownLatch：

场景1.某线程需要等待多个线程执行完毕，再执行。  
场景2.实现多个线程共同等待，同时开始执行任务，不可重用。(此类似于CyclicBarrier，可重用)

CyclicBarrier：

CyclicBarrier是一个同步工具类，它允许一组线程互相等待，直到到达某个公共屏障点。  
与CountDownLatch不同的是该barrier在释放等待线程后可以重用，所以称它为循环（Cyclic）的屏障

### 说说 Semaphore 原理

* Semaphore是一个计数信号量，它的本质是一个”共享锁”。
* 信号量维护了一个信号量许可集。线程可以通过调用acquire()来获取信号量的许可；当信号量中有可用的许可时，线程能获取该许可；否则线程必须等待，直到有可用的许可为止。 线程可以通过release()来释放它所持有的信号量许可。

<http://www.importnew.com/21889.html>

|  |
| --- |
| 下面对上面说的三个辅助类进行一个总结：  1）CountDownLatch和CyclicBarrier都能够实现线程之间的等待，只不过它们侧重点不同：  CountDownLatch一般用于某个线程A等待若干个其他线程执行完任务之后，它才执行；  而CyclicBarrier一般用于一组线程互相等待至某个状态，然后这一组线程再同时执行；  另外，CountDownLatch是不能够重用的，而CyclicBarrier是可以重用的。  2）Semaphore其实和锁有点类似，它一般用于控制对某组资源的访问权限。 |

### 说说 Exchanger 原理

 Exchanger（交换者）是一个用于线程间协作的工具类。Exchanger用于进行线程间的数据交换。它提供一个同步点，在这个同步点两个线程可以交换彼此的数据。这两个线程通过exchange方法交换数据， 如果第一个线程先执行exchange方法，它会一直等待第二个线程也执行exchange，当两个线程都到达同步点时，这两个线程就可以交换数据，将本线程生产出来的数据传递给对方。因此使用Exchanger的重点是成对的线程使用exchange()方法，当有一对线程达到了同步点，就会进行交换数据。因此该工具类的线程对象是成对的。

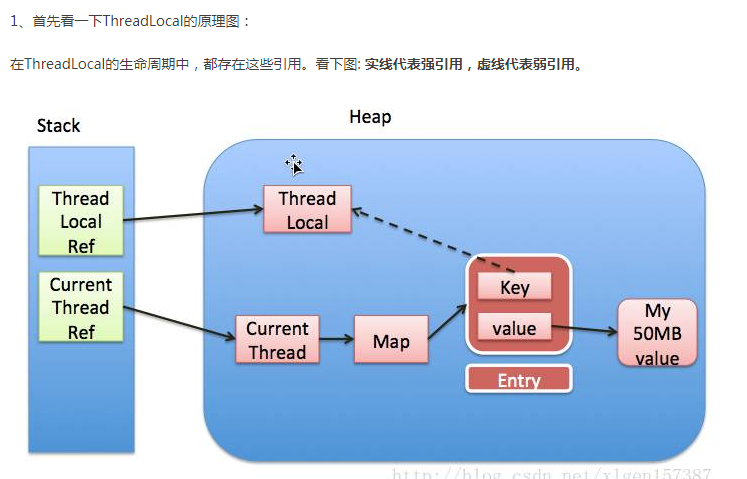
       Exchanger类提供了两个方法，String exchange(V x):用于交换，启动交换并等待另一个线程调用exchange；String exchange(V x,long timeout,TimeUnit unit)：用于交换，启动交换并等待另一个线程调用exchange，并且设置最大等待时间，当等待时间超过timeout便停止等待。

### ThreadLocal 原理分析，ThreadLocal为什么会出现OOM，出现的深层次原理

<https://blog.csdn.net/u012088516/article/details/84067841>

ThreadLocalMap使用ThreadLocal的弱引用作为key，如果一个ThreadLocal没有外部强引用来引用它，那么系统 GC 的时候，这个ThreadLocal势必会被回收，这样一来，ThreadLocalMap中就会出现key为null的Entry，就没有办法访问这些key为null的Entry的value，如果当前线程再迟迟不结束的话，这些key为null的Entry的value就会一直存在一条强引用链：**Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value**永远无法回收，造成内存泄漏。

<https://blog.csdn.net/bntx2jsqfehy7/article/details/78315161>



|  |
| --- |
| ThreadLocal的实现是这样的：每个Thread 维护一个 ThreadLocalMap 映射表，这个映射表的 key 是 ThreadLocal实例本身，value 是真正需要存储的 Object。  3、也就是说 ThreadLocal 本身并不存储值，它只是作为一个 key 来让线程从 ThreadLocalMap 获取 value。值得注意的是图中的虚线，表示 ThreadLocalMap 是使用 ThreadLocal 的弱引用作为 Key 的，弱引用的对象在 GC 时会被回收。  4、ThreadLocalMap使用ThreadLocal的弱引用作为key，如果一个ThreadLocal没有外部强引用来引用它，那么系统 GC 的时候，这个ThreadLocal势必会被回收，这样一来，ThreadLocalMap中就会出现key为null的Entry，就没有办法访问这些key为null的Entry的value，如果当前线程再迟迟不结束的话，这些key为null的Entry的value就会一直存在一条强引用链：Thread Ref -> Thread -> ThreaLocalMap -> Entry -> value永远无法回收，造成内存泄漏。  5、总的来说就是，ThreadLocal里面使用了一个存在弱引用的map, map的类型是ThreadLocal.ThreadLocalMap. Map中的key为一个threadlocal实例。这个Map的确使用了弱引用，不过弱引用只是针对key。每个key都弱引用指向threadlocal。 当把threadlocal实例置为null以后，没有任何强引用指向threadlocal实例，所以threadlocal将会被gc回收。  但是，我们的value却不能回收，而这块value永远不会被访问到了，所以存在着内存泄露。因为存在一条从current thread连接过来的强引用。只有当前thread结束以后，current thread就不会存在栈中，强引用断开，Current Thread、Map value将全部被GC回收。最好的做法是将调用threadlocal的remove方法，这也是等会后边要说的。  6、其实，ThreadLocalMap的设计中已经考虑到这种情况，也加上了一些防护措施：在ThreadLocal的get(),set(),remove()的时候都会清除线程ThreadLocalMap里所有key为null的value。这一点在上一节中也讲到过！  7、但是这些被动的预防措施并不能保证不会内存泄漏：  （1）使用static的ThreadLocal，延长了ThreadLocal的生命周期，可能导致内存泄漏。  （2）分配使用了ThreadLocal又不再调用get(),set(),remove()方法，那么就会导致内存泄漏，因为这块内存一直存在。 |

### 讲讲线程池的实现原理

<https://www.cnblogs.com/dongguacai/p/6030187.html>

ThreadPoolExecutor

corePoolSize：线程池核心线程数量

maximumPoolSize:线程池最大线程数量

keepAliverTime：当活跃线程数大于核心线程数时，空闲的多余线程最大存活时间

unit：存活时间的单位

workQueue：存放任务的队列，

<https://www.cnblogs.com/szwh/p/7761171.html>

1、ArrayBlockingQueue

      基于数组实现的一个阻塞队列，在创建ArrayBlockingQueue对象时必须制定容量大小。并且可以指定公平性与非公平性，默认情况下为非公平的，即不保证等待时间最长的队列最优先能够访问队列。

2、LinkedBlockingQueue

     基于链表实现的一个阻塞队列，在创建LinkedBlockingQueue对象时如果不指定容量大小，则默认大小为Integer.MAX\_VALUE。

3、PriorityBlockingQueue[praɪˈɒrəti]

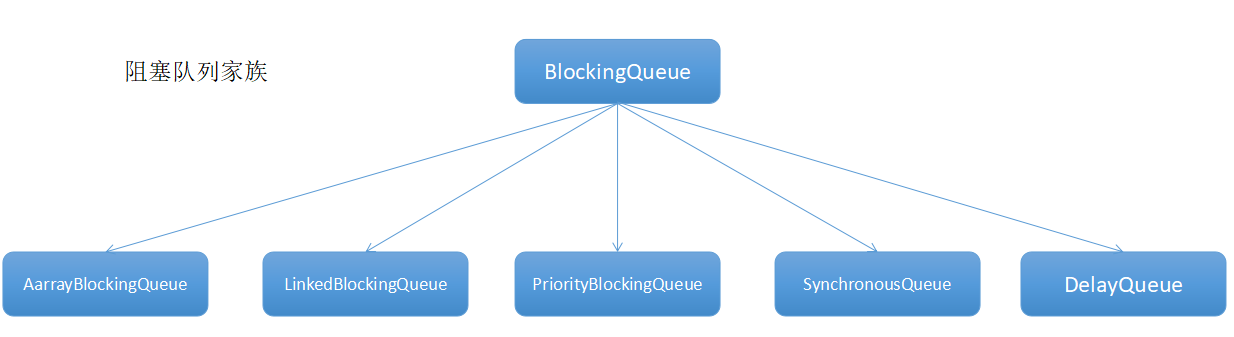
       以上2种队列都是先进先出队列，而PriorityBlockingQueue却不是，它会按照元素的优先级对元素进行排序，按照优先级顺序出队，每次出队的元素都是优先级最高的元素。注意，此阻塞队列为无界阻塞队列，即

容量没有上限（通过源码就可以知道，它没有容器满的信号标志），前面2种都是有界队列。

4、DelayQueue

       基于PriorityQueue，一种延时阻塞队列，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也是一个无界队列，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。



threadfactory: 线程创建工厂

handler：超出线程范围和队列容量的任务的处理程序

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

ThreadPoolExecutor.CallerRunsPolicy：由调用线程处理该任务

线程结束：

<https://justsee.iteye.com/blog/999189>

shutDown()

当线程池调用该方法时,线程池的状态则立刻变成SHUTDOWN状态。此时，则不能再往线程池中添加任何任务，否则将会抛出RejectedExecutionException异常。但是，此时线程池不会立刻退出，直到添加到线程池中的任务都已经处理完成，才会退出。

shutdownNow()

根据JDK文档描述，大致意思是：执行该方法，线程池的状态立刻变成STOP状态，并试图停止所有正在执行的线程，不再处理还在池队列中等待的任务，当然，它会返回那些未执行的任务。

它试图终止线程的方法是通过调用Thread.interrupt()方法来实现的，但是大家知道，这种方法的作用有限，如果线程中没有sleep 、wait、Condition、定时锁等应用, interrupt()方法是无法中断当前的线程的。所以，ShutdownNow()并不代表线程池就一定立即就能退出，它可能必须要等待所有正在执行的任务都执行完成了才能退出。

线程提交过程：：

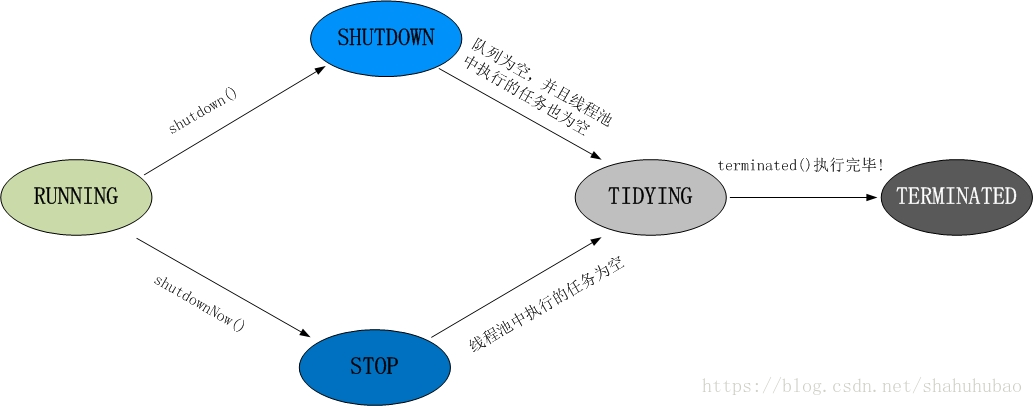
1、判断**线程池里的核心线程**是否都在执行任务，如果不是（核心线程空闲或者还有核心线程没有被创建）则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则进入下个流程。

2、线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。

3、判断**线程池里的线程**是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。

<https://blog.csdn.net/shahuhubao/article/details/80311992>

线程池的5种状态：Running、ShutDown、Stop、Tidying、Terminated。



### 线程池的几种实现方式



### 线程的生命周期，状态是如何转移的

### <https://www.cnblogs.com/happy-coder/p/6587092.html>

### 

* 可参考：《[Java多线程编程核心技术](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247484881&idx=2&sn=b0ecf85cd7c9e543c84e7a9859c20a26&chksm=e9c5fc60deb27576a6a9c453dabc585f43d9f29fd8a8f37ed0e7cc2f012c86b23fbd21763a39&scene=21#wechat_redirect)》

## 锁机制

### 说说线程安全问题，什么是线程安全，如何保证线程安全

### 重入锁的概念，重入锁为什么可以防止死锁

### 产生死锁的四个条件（互斥、请求与保持、不剥夺、循环等待）

### 如何检查死锁（通过jConsole检查死锁）

参考链接：<https://blog.csdn.net/u014039577/article/details/52351626>

<https://www.cnblogs.com/flyingeagle/articles/6853167.html>

###### Jconsole查看死锁：

进入java安装的位置，输入Jconsole，然后弹出界面（或者进入安装目录/java/jdk1.70\_80/bin/，点击Jconsole.exe）

点击线程进入

然后点击检测死锁

然后可以看到造成死锁的两个线程，以及死锁原因

###### Jstack查看死锁：

同样，也是进入jdk安装目录的bin下面，输入jps，先查看我们要检测死锁的进程

然后可以看到进程Test的进程号：8384，然后执行：Jstack -l 8384

查看死锁信息(Java-level deadlock)

### volatile 实现原理（禁止指令重排、刷新内存）

参考链接：

<http://www.importnew.com/23520.html>

<http://ifeve.com/volatile/>

java编程语言允许线程访问共享变量，为了确保共享变量能被准确和一致的更新，线程应该确保通过排他锁单独获得这个变量。Java语言提供了volatile，在某些情况下比锁更加方便。如果一个字段被声明成volatile，java线程内存模型确保所有线程看到这个变量的值是一致的。

Volatile变量修饰符如果使用**恰当**的话，它比synchronized的**使用和执行成本会更低**，因为它不会引起线程上下文的切换和调度

**Lock前缀指令会引起处理器缓存回写到内存**

**一个处理器的缓存回写到内存会导致其他处理器的缓存无效**

### synchronized 实现原理（对象监视器）

**参考链接：**

<https://www.cnblogs.com/paddix/p/5367116.html>

<https://blog.csdn.net/javazejian/article/details/72828483>

**Synchronized是Java中解决并发问题的一种最常用的方法，也是最简单的一种方法。Synchronized的作用主要有三个：**

**（1）确保线程互斥的访问同步代码**

**（2）保证共享变量的修改能够及时可见**

**（3）有效解决重排序问题。**

**从语法上讲，Synchronized总共有三种用法：**

**（1）修饰普通方法**

**（2）修饰静态方法**

**（3）修饰代码块**

当一个线程正在访问一个对象的 synchronized 实例方法，那么其他线程不能访问该对象的其他 synchronized 方法，毕竟一个对象只有一把锁，当一个线程获取了该对象的锁之后，其他线程无法获取该对象的锁，所以无法访问该对象的其他synchronized实例方法，但是其他线程还是可以访问该实例对象的其他非synchronized方法

### synchronized 与 lock 的区别

参考链接：

<https://www.cnblogs.com/baizhanshi/p/6419268.html>

<https://blog.csdn.net/tinyDolphin/article/details/79273103>

如果一个代码块被synchronized修饰了，当一个线程获取了对应的锁，并执行该代码块时，其他线程便只能一直等待，等待获取锁的线程释放锁，而这里获取锁的线程释放锁只会有两种情况：

1）获取锁的线程执行完了该代码块，然后线程释放对锁的占有；

2）线程执行发生异常，此时JVM会让线程自动释放锁。

当有多个线程读写文件时，读操作和写操作会发生冲突现象，写操作和写操作会发生冲突现象，但是读操作和读操作不会发生冲突现象。

但是采用synchronized关键字来实现同步的话，就会导致一个问题：

如果多个线程都只是进行读操作，所以当一个线程在进行读操作时，其他线程只能等待无法进行读操作。

因此就需要一种机制来使得多个线程都只是进行读操作时，线程之间不会发生冲突，通过Lock就可以办到。

另外，通过Lock可以知道线程有没有成功获取到锁。这个是synchronized无法办到的。

总结一下，也就是说Lock提供了比synchronized更多的功能。但是要注意以下几点：

1）Lock不是Java语言内置的，synchronized是Java语言的关键字，因此是内置特性。Lock是一个类，通过这个类可以实现同步访问；

2）Lock和synchronized有一点非常大的不同，采用synchronized不需要用户去手动释放锁，当synchronized方法或者synchronized代码块执行完之后，系统会自动让线程释放对锁的占用；而Lock则必须要用户去手动释放锁，如果没有主动释放锁，就有可能导致出现死锁现象。

### AQS同步队列

<https://www.cnblogs.com/waterystone/p/4920797.html>

<https://blog.csdn.net/zcw4237256/article/details/78552741>

一脸懵逼

* CAS无锁的概念、乐观锁和悲观锁
* 常见的原子操作类

<https://www.cnblogs.com/boothsun/p/8010692.html>

<https://blog.csdn.net/u011116672/article/details/51068828>

原子操作类共有13个类，在java.util.concurrent.atomic包下，可以分为四种类型的原子更新类：原子更新基本类型、原子更新数组类型、原子更新引用和原子更新属性。

* 什么是ABA问题，出现ABA问题JDK是如何解决的

CAS操作可能带来ABA问题，因为CAS操作需要在操作值的时候，检查值有没有发生变化，如果没有发发生变化则更新。如果一个值原理是A，变成了B，又变成了A，那么使用CAS进行检查时会认为它的值没有变化，但是实际上却变了。

ABA问题的解决办法就是使用版本号，在变量前面追加版本号，每次变量更新时把版本号加1，那么A-B-A就会变成1A-2B-3A。

从jdk1.5开始，jdk中的Atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法的作用首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果都相等，则以原子方式将该引用和标志的值设为给定的更新值。

### 乐观锁的业务场景及实现方式

### Java 8并法包下常见的并发类

### 偏向锁、轻量级锁、重量级锁、自旋锁的概念

* 可参考：《[Java多线程编程核心技术](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247484881&idx=2&sn=b0ecf85cd7c9e543c84e7a9859c20a26&chksm=e9c5fc60deb27576a6a9c453dabc585f43d9f29fd8a8f37ed0e7cc2f012c86b23fbd21763a39&scene=21#wechat_redirect)》

## JVM

### JVM运行时内存区域划分

参考链接：

<https://www.cnblogs.com/dolphin0520/p/3613043.html>

<https://www.cnblogs.com/wangjzh/p/5258254.html>

根据《Java虚拟机规范》的规定，运行时数据区通常包括这几个部分：程序计数器(Program Counter Register)、Java栈(VM Stack)、本地方法栈(Native Method Stack)、方法区(Method Area)、堆(Heap)。





1）程序计数器（Program Counter Register）：

它保存的是程序当前执行的指令的地址（也可以说保存下一条指令的所在存储单元的地址），当CPU需要执行指令时，需要从程序计数器中得到当前需要执行的指令所在存储单元的地址，然后根据得到的地址获取到指令，在得到指令之后，程序计数器便自动加1或者根据转移指针得到下一条指令的地址，如此循环，直至执行完所有的指令。

2）Java栈也称作虚拟机栈（Java Vitual Machine Stack）：

当线程执行一个方法时，就会随之创建一个对应的栈帧，并将建立的栈帧压栈。当方法执行完毕之后，便会将栈帧出栈。因此可知，线程当前执行的方法所对应的栈帧必定位于Java栈的顶部。

3）本地方法栈：

本地方法栈与Java栈的作用和原理非常相似。区别只不过是Java栈是为执行Java方法服务的，而本地方法栈则是为执行本地方法（Native Method）服务的。

4）堆：

Java中的堆是用来存储对象本身的以及数组（当然，数组引用是存放在Java栈中的）。Java的垃圾回收机制会自动进行处理。因此这部分空间也是Java垃圾收集器管理的主要区域。另外，堆是被所有线程共享的，在JVM中只有一个堆。

5）方法区：

方法区在JVM中也是一个非常重要的区域，它与堆一样，是被线程共享的区域。在方法区中，存储了每个类的信息（包括类的名称、方法信息、字段信息）、静态变量、常量以及编译器编译后的代码等

* 内存溢出OOM和堆栈溢出SOE的示例及原因、如何排查与解决

OOM:

<https://www.cnblogs.com/ThinkVenus/p/6805495.html>（什么是java OOM？如何分析及解决oom问题？）

<https://www.jianshu.com/p/41ffbf31b20c>(什么是OOM？如何解决OOM问题!)

<https://www.cnblogs.com/kongzhongqijing/articles/7283599.html>（常见OOM现象）

SOE：

参考链接：

<https://www.cnblogs.com/csniper/p/5486828.html>

<https://blog.csdn.net/yanghw117/article/details/80889298>

<https://blog.csdn.net/sinat_29912455/article/details/51125748>(异常、堆内存溢出、OOM的几种情况)

  -Xss128k：设置每个线程的堆栈大小。 JDK5.0以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。根据应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更 多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。

### 如何判断对象是否可以回收或存活

引用计数算法

给对象添加一个引用计数器，每当有一个地方引用它时，计数器的值就+1；当引用失效时，值就-1；任何时刻计数器为0的对象就是不能再被使用的。

可达性分析算法

在主流的商用程序语言[java，C#]的主流实现中都是通过可达性分析来判定对象是否存在的。这个算法的基本思路就是通过一系列的成为”GC Roots”的对象作为起始点，从这些结点开始向下搜索，搜索所经过的路径成为“引用链”,当一个对象到GC Roots没有任何引用链时，则证明此对象是不可用的

在Java语言中， 可作为GC Roots的对象包括下面几种：

虚拟机栈中引用的对象

方法区中类静态属性引用的对象

方法区中常量引用的对象

本地方法栈中JNI[即一般说的Native]引用的对象

### 常见的GC回收算法及其含义

标记-清除算法、复制算法、标记-整理算法

Serial：新生代收集器，采用复制算法，单线程。

ParNew：新生代收集器，采用复制算法，多线程。

Parallel Scavenge：新生代收集器，采用复制算法，多线程，注重吞吐量。

Serial Old：老年代收集器，采用标记-整理算法，单线程。

Parallel Old：老年代收集器，采用标记-整理算法，多线程，与Parallel Scavenge结合使用。

CMS：老年代收集器，采用标记-清除算法，相比以上收集器收集过去相对复杂，停止时间短。

G1：年轻代和老年代收集器，基本采用标记-整理算法，局部采用复制算法，收集过程跟CMS相当，但概念差异，是目前最新的收集器之一，使用范围暂时有待检验。

* 常见的JVM性能监控和故障处理工具类：jps、jstat、jmap、jinfo、jconsole等

当Jconsole连接成功后，它从JMX获取信息，我们便可以在里面监控具体的内容。  
Jconsole能捕获到以下信息：

1. 概述 － JVM概述和一些监控变量的信息
2. 内存 － 内存的使用信息
3. 线程 － 线程的使用信息
4. 类 － 加载java类的信息
5. VM － JVM摘要
6. MBeans － 所有MBeans的信息

### JVM如何设置参数

<https://www.cnblogs.com/marcotan/p/4256885.html>

-Xmx3550m：设置JVM最大堆内存为3550M。  
-Xms3550m：设置JVM初始堆内存为3550M。此值可以设置与-Xmx相同，以避免每次垃圾回收完成后JVM重新分配内存。  
-Xss128k：设置每个线程的栈大小。JDK5.0以后每个线程栈大小为1M，之前每个线程栈大小为256K。应当根据应用的线程所需内存大小进行调整。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。需要注意的是：当这个值被设置的较大（例如>2MB）时将会在很大程度上降低系统的性能。  
-Xmn2g：设置年轻代大小为2G。在整个堆内存大小确定的情况下，增大年轻代将会减小年老代，反之亦然。此值关系到JVM垃圾回收，对系统性能影响较大，官方推荐配置为整个堆大小的3/8。  
-XX:NewSize=1024m：设置年轻代初始值为1024M。  
-XX:MaxNewSize=1024m：设置年轻代最大值为1024M。  
-XX:PermSize=256m：设置持久代初始值为256M。  
-XX:MaxPermSize=256m：设置持久代最大值为256M。  
-XX:NewRatio=4：设置年轻代（包括1个Eden和2个Survivor区）与年老代的比值。表示年轻代比年老代为1:4。  
-XX:SurvivorRatio=4：设置年轻代中Eden区与Survivor区的比值。表示2个Survivor区（JVM堆内存年轻代中默认有2个大小相等的Survivor区）与1个Eden区的比值为2:4，即1个Survivor区占整个年轻代大小的1/6。  
-XX:MaxTenuringThreshold=7：表示一个对象如果在Survivor区（救助空间）移动了7次还没有被垃圾回收就进入年老代。如果设置为0的话，则年轻代对象不经过Survivor区，直接进入年老代，对于需要大量常驻内存的应用，这样做可以提高效率。如果将此值设置为一个较大值，则年轻代对象会在Survivor区进行多次复制，这样可以增加对象在年轻代存活时间，增加对象在年轻代被垃圾回收的概率，减少Full GC的频率，这样做可以在某种程度上提高服务稳定性。

### JVM性能调优

<https://www.cnblogs.com/csniper/p/5592593.html>

**1.**根据Java虚拟机规范，**JVM将内存划分为：**

* New（年轻代）
* Tenured（年老代）
* 永久代（Perm）

  其中New和Tenured属于堆内存，堆内存会从JVM启动参数（-Xmx:3G）指定的内存中分配，**Perm不属于堆内存，有虚拟机直接分配**，但可以通过**-XX:PermSize -XX:MaxPermSize** 等参数调整其大小。

**内存泄漏及解决方法**

程序及JVM进行调优。从以下几个方面进行：

* **线程池**：解决用户响应时间长的问题
* **连接池**
* **JVM启动参数**：调整各代的内存比例和垃圾回收算法，提高吞吐量
* **程序算法**：改进程序逻辑算法提高性能

**调优方法**

一切都是为了这一步，调优，在调优之前，我们需要记住下面的原则：

1、多数的Java应用不需要在服务器上进行GC优化；

2、多数导致GC问题的Java应用，都不是因为我们参数设置错误，而是代码问题；

3、在应用上线之前，先考虑将机器的JVM参数设置到最优（最适合）；

4、减少创建对象的数量；

5、减少使用全局变量和大对象；

6、GC优化是到最后不得已才采用的手段；

7、在实际使用中，分析GC情况优化代码比优化GC参数要多得多；

### 类加载器、双亲委派模型、一个类的生命周期、类是如何加载到JVM中的

双亲委派模型是指：

当一个类加载器收到类加载请求时，不会直接加载这个类，而是把这个加载请求委派给自己父加载器去完成。如果父加载器无法加载时，子加载器才会去尝试加载。

**类加载器种类**

从Java虚拟机的角度来说，只存在两种不同的类加载器：一种是启动类加载器（Bootstrap ClassLoader），这个类加载器使用C++语言实现（HotSpot虚拟机中），是虚拟机自身的一部分；另一种就是所有其他的类加载器，这些类加载器都有Java语言实现，独立于虚拟机外部，并且全部继承自java.lang.ClassLoader。

从开发者的角度，类加载器可以细分为：

* 启动（Bootstrap）类加载器：负责将 Java\_Home/lib下面的类库加载到内存中（比如rt.jar）。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作。
* 标准扩展（Extension）类加载器：是由 Sun 的 ExtClassLoader（sun.misc.Launcher$ExtClassLoader）实现的。它负责将Java\_Home /lib/ext或者由系统变量 java.ext.dir指定位置中的类库加载到内存中。开发者可以直接使用标准扩展类加载器。
* 应用程序（Application）类加载器：是由 Sun 的 AppClassLoader（sun.misc.Launcher$AppClassLoader）实现的。它负责将系统类路径（CLASSPATH）中指定的类库加载到内存中。开发者可以直接使用系统类加载器。由于这个类加载器是ClassLoader中的getSystemClassLoader()方法的返回值，因此一般称为**系统（System）加载器**。

<https://blog.csdn.net/ma345787383/article/details/77098976>

1、类加载过程：加载、验证、准备、解析、初始化

|  |  |
| --- | --- |
| 加载 | 在加载阶段，虚拟机主要完成三件事：       1.通过一个类的全限定名来获取定义此类的二进制字节流。        2.将这个字节流所代表的静态存储结构转化为方法区域的运行时[**数据结构**](http://lib.csdn.net/base/datastructure)。        3.在[**Java**](http://lib.csdn.net/base/java)堆中生成一个代表这个类的java.lang.Class对象，作为方法区域数据的访问入口 |
| 验证 | 验证阶段作用是保证Class文件的字节流包含的信息符合JVM规范，不会给JVM造成危害。如果验证失败，就会抛出一个java.lang.VerifyError异常或其子类异常。验证过程分为四个阶段       1.文件格式验证：验证字节流文件是否符合Class文件格式的规范，并且能被当前虚拟机正确的处理。       2.元数据验证：是对字节码描述的信息进行语义分析，以保证其描述的信息符合Java语言的规范。       3.字节码验证：主要是进行数据流和控制流的分析，保证被校验类的方法在运行时不会危害虚拟机。       4.符号引用验证：符号引用验证发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在解析阶段中发生。 |
| 准备 | 准备阶段为变量分配内存并设置类变量的初始化。在这个阶段分配的仅为类的变量（static修饰的变量），而不包括类的实例变量。对已非final的变量，JVM会将其设置成“零值”，而不是其赋值语句的值：          pirvate static int size = 12;          那么在这个阶段，size的值为0，而不是12。 final修饰的类变量将会赋值成真实的值。 |
| 解析 | 解析过程是将常量池内的符号引用替换成直接引用。主要包括四种类型引用的解析。类或接口的解析、字段解析、方法解析、接口方法解析。 |
| 初始化 | 在准备阶段，类变量已经经过一次初始化了，在这个阶段，则是根据程序员通过程序制定的计划去初始化类的变量和其他资源。这些资源有static{}块，构造函数，父类的初始化等。          至于使用和卸载阶段阶段，这里不再过多说明，使用过程就是根据程序定义的行为执行，卸载由GC完成。 |
| 使用 | 新线程---程序计数器----jvm栈执行（对象引用）-----堆内存（直接引用）----方法区 |
| 卸载 | GC垃圾回收 |

2、双亲委派机制、意义、方法

|  |  |
| --- | --- |
| 机制 | 启动（Bootstrap）类加载器----->标准扩展（Extension）类加载器--->系统（System）类加载器---->上下文(Custom)类加载器 从左到右加载：首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。 |
| 意义 | 防止内存中出现多份同样的字节码  使用委托机制，会递归的向父类查找，如果类A中引用了类B，Java虚拟机将使用加载类A的类加载器去加载类B，如果A加载器已加载类A，那么B使用A的类加载器进行加载时，就不会在加载类A的字节码了 |
| 方法 | 《1》启动（Bootstrap）类加载器 《2》标准扩展（Extension）类加载器 《3》应用程序类加载器（Application ） 《4》上下文(Custom)类加载器 |

破坏双亲委托机制

<https://www.zhihu.com/question/45022217>

其实很简单，类加载器的这个机制是由java代码实现的。(不过bootstrap怎么实现就不知道了)  
具体来说在方法 load(String,boolean)中  
注释和代码都很清楚  
1.寻找已加载的类  
2.若没有找到，则去parent中找  
3.调用findClass方法  
4.要是还没有则抛出异常  
所以有两个方法  
1.在构造类加载器时将parent设为null(构造函数参数为null即可，可以看下源码)  
2.重写load(String,boolean)方法，改变类的查找机制。

### 类加载的过程：加载、验证、准备、解析、初始化

<https://www.cnblogs.com/ygj0930/p/6536048.html>

类的生命周期包括7个部分：加载——验证——准备——解析——初始化——使用——卸载

加载阶段，

虚拟机需要完成三件事：通过类名字获取类的二进制字节流——将字节流的内容转存到方法区——在内存中生成一个Class对象作为该类方法区数据的访问入口。

验证

       确保class文件的二进制字节流中包含的信息符号虚拟机要求，包括：文件格式验证、元数据验证（数据语义分析）、字节码验证（数据流语义合法性）、符号引用验证（符号引用的匹配性校验，确保解析能正确执行）

准备

       为**类变量（静态变量）**在**方法区**分配内存，并设置**零值**。注意：这里是类变量，不是实例变量，实例变量是对象分配到**堆内存**时根据运行时动态生成的

### 强引用、软引用、弱引用、虚引用

Java四种引用包括强引用，软引用，弱引用，虚引用。

强引用：

只要引用存在，垃圾回收器永远不会回收

Object obj = new Object();

//可直接通过obj取得对应的对象 如obj.equels(new Object());

而这样 obj对象对后面new Object的一个强引用，只有当obj这个引用被释放之后，对象才会被释放掉，这也是我们经常所用到的编码形式。

软引用：

非必须引用，内存溢出之前进行回收，可以通过以下代码实现

Object obj = new Object();

SoftReference<Object> sf = new SoftReference<Object>(obj);

obj = null;

sf.get();//有时候会返回null

这时候sf是对obj的一个软引用，通过sf.get()方法可以取到这个对象，当然，当这个对象被标记为需要回收的对象时，则返回null；

软引用主要用户实现类似缓存的功能，在内存足够的情况下直接通过软引用取值，无需从繁忙的真实来源查询数据，提升速度；当内存不足时，自动删除这部分缓存数据，从真正的来源查询这些数据。

弱引用：

第二次垃圾回收时回收，可以通过如下代码实现

Object obj = new Object();

WeakReference<Object> wf = new WeakReference<Object>(obj);

obj = null;

wf.get();//有时候会返回null

wf.isEnQueued();//返回是否被垃圾回收器标记为即将回收的垃圾

弱引用是在第二次垃圾回收时回收，短时间内通过弱引用取对应的数据，可以取到，当执行过第二次垃圾回收时，将返回null。

弱引用主要用于监控对象是否已经被垃圾回收器标记为即将回收的垃圾，可以通过弱引用的isEnQueued方法返回对象是否被垃圾回收器标记。

虚引用：

垃圾回收时回收，无法通过引用取到对象值，可以通过如下代码实现

Object obj = new Object();

PhantomReference<Object> pf = new PhantomReference<Object>(obj);

obj=null;

pf.get();//永远返回null

pf.isEnQueued();//返回是否从内存中已经删除

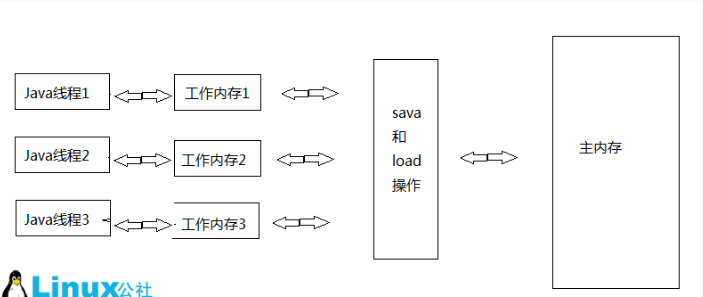
虚引用是每次垃圾回收的时候都会被回收，通过虚引用的get方法永远获取到的数据为null，因此也被成为幽灵引用。

虚引用主要用于检测对象是否已经从内存中删除。

### Java内存模型JMM

<https://www.linuxidc.com/Linux/2016-03/129549.htm>

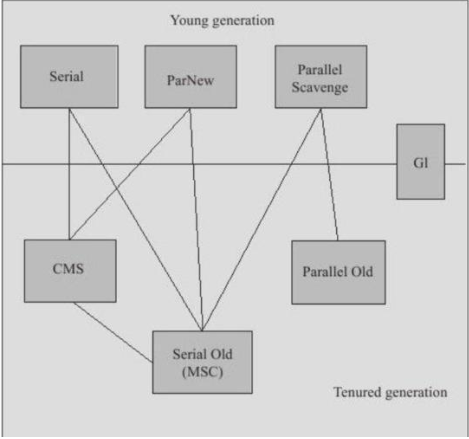
JMM规定了所有的变量都存储在主内存（Main Memory）中。每个线程还有自己的工作内存（Working Memory）,线程的工作内存中保存了该线程使用到的变量的主内存的副本拷贝，线程对变量的所有操作（读取、赋值等）都必须在工作内存中进行，而不能直接读写主内存中的变量（volatile变量仍然有工作内存的拷贝，但是由于它特殊的操作顺序性规定，所以看起来如同直接在主内存中读写访问一般）。不同的线程之间也无法直接访问对方工作内存中的变量，线程之间值的传递都需要通过主内存来完成。



### 垃圾回收器

<http://baijiahao.baidu.com/s?id=1579653447835626526&wfr=spider&for=pc>

<https://www.cnblogs.com/cxxjohnson/p/8625713.html>



**Serial收集器**

* 在JDK1.3之前唯一的收集器
* serial串行，它只会使用一个CPU和一个线程去完成垃圾收集工作，所以它是单线程的
* 在它运行的时候，必须暂停掉所有其他的线程，直到它工作结束，这项工作实际上是由虚拟机在后台自动发起和自动完成的，用户是不能控制的，在运行过程中，一旦垃圾收集器需要启动，就会停掉所有其他的线程
* 对于serial收集器今天仍然在使用，因为有它存在的用处，它相对其他更复杂的收集器而言
* 有着简单而高效的优势。它不需要去考虑跟其他线程的交互，抢资源，专心做自己的事就可以了
* Serial：用于新生代垃圾收集，复制算法；SerialOld：用于老年代垃圾收集，标记整理算法

从Serial收集器到Parallel收集器，再到Concurrent Mark Sweep（CMS）乃至GC收集器的最前沿成果Garbage First（G1）收集器，我们看到了一个个越来越优秀（也越来越复杂）的收集器的出现，用户线程的停顿时间在不断缩短，但是我们依然没办法完全克服这个问题，所以工程师们依旧在不断的为之努力

**ParNew收集器**

* ParNew收集器其实就是Serial收集器的多线程版本，多线程运行，采用并行策略
* 除了Serial收集器外，目前只有它能与CMS收集器配合工作
* ParNew收集器在单CPU的环境中绝对不会有比Serial收集器更好的效果，由于多线程存在竞争交互资源消耗，所以它即使在超线程的情况下也未必能超过serial
* 默认开启的收集线程数与CPU的数量相同，我们也可以使用-XX：ParallelGCThreads参数来限制垃圾收集的线程数

**CMS收集器**

* CMS（Concurrent Mark Sweep）基于“标记—清除”算法，用于老年代,
* 所以其关注点在于减少“pause time”也即因垃圾回收导致的stop the world时间。
* 因为CMS是“并发”运行的，也即垃圾收集线程可以和用户线程同时运行。
* CMS最主要解决了pause time，但是会占用CPU资源，牺牲吞吐量。CMS默认启动的回收线程数是（CPU数量+3）/ 4，当CPU<4个时，会影响用户线程的执行。
* 内存碎片问题，碎片会给大对象的内存分配造成麻烦，如果老年代的可用的连续空间也无法分配时，会触发full gc。并且full gc时如果发生young gc会被young gc打断，执行完young gc之后再继续执行full gc。
* CMS的回收分为几个阶段：

初始标记：标记一下GC Roots能直接关联到的对象，会“Stop The World”

并发标记：GC Roots Tracing，可以和用户线程并发执行。

重新标记：标记期间产生的对象存活的再次判断，修正对这些对象的标记，执行时间相对并发标记短，会“Stop The World”。

并发清除：清除对象,可以和用户线程并发执行。

**G1垃圾收集器**

* G1（Garbage-First）是JDK7-u4才推出商用的收集器
* 并行与并发，能充分利用多CPU、多核环境，可以并行来缩短"Stop The World"停顿时间，也可以并发让垃圾收集与用户程序同时进行
* 分代收集，收集范围包括新生代和老年代
* 结合多种垃圾收集算法，空间整合，不产生碎片；从整体看，是基于标记-整理算法，从局部（两个Region间）看，是基于复制算法
* 可预测的停顿：低停顿的同时实现高吞吐量
* 在下面的情况时，使用G1可能比CMS好：

（1）、超过50％的Java堆被活动数据占用；

（2）、对象分配频率或年代提升频率变化很大；

（3）、GC停顿时间过长（长于0.5至1秒）。

## 设计模式

### 常见的设计模式

* 设计模式的的六大原则及其含义

<https://www.cnblogs.com/dolphin0520/p/3919839.html>

<https://blog.csdn.net/q291611265/article/details/48465113>

<https://blog.csdn.net/wuxianzhenjia/article/details/80263619>

|  |  |  |
| --- | --- | --- |
| **原则** | **含义** | **具体方法** |
| 开闭原则 | 对扩展开放,对修改关闭 | 多使用抽象类和接口 |
| 里氏代换原则 | 基类可以被子类替换 | 使用抽象类继承,不使用具体类继承 |
| 依赖倒转原则 | 要依赖于抽象,不要依赖于具体 | 针对接口编程,不针对实现编程 |
| 接口隔离原则 | 使用多个隔离的接口,比使用单个接口好 | 建立最小的接口 |
| 迪米特法则 | 一个软件实体应当尽可能少地与其他实体发生相互作用 | 通过中间类建立联系 |
| 合成复用原则 | 尽量使用合成/聚合,而不是使用继承 | 尽量使用合成/聚合,而不是使用继承 |

一、单一职责原则

该原则是指一个类只负责一个功能领域中的相应职责，或者可以定义为：就一个类而言，应该只有一个引起它变化的原因。它是实现高内聚、低耦合的指导方针，它是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，而发现类的多重职责需要设计人员具有较强的分析设计能力和相关实践经验。

二、开闭原则

该原则是面向对象的可复用设计的第一块基石，它是最重要的面向对象设计原则。一个软件实体应当对扩展开放，对修改关闭。即软件实体应尽量在不修改原有代码的情况下进行扩展，软件实体可以指一个软件模块、一个由多个类组成的局部结构或一个独立的类。

三、里氏替换原则

该原则是指所有引用基类（父类）的地方必须能透明地使用其子类的对象。在软件中将一个基类对象替换成它的子类对象，程序将不会产生任何错误和异常，反过来则不成立，如果一个软件实体使用的是一个子类对象的话，那么它不一定能够使用基类对象。

四、依赖倒置原则

抽象不应该依赖于细节，细节应当依赖于抽象。换言之，要针对接口编程，而不是针对实现编程。  
在实现依赖倒转原则时，我们需要针对抽象层编程，而将具体类的对象通过依赖注入(DI)的方式注入到其他对象中，依赖注入是指当一个对象要与其他对象发生依赖关系时，通过抽象来注入所依赖的对象。常用的注入方式有三种，分别是：构造注入，设值注入（Setter注入）和接口注入。构造注入是指通过构造函数来传入具体类的对象，设值注入是指通过Setter方法来传入具体类的对象，而接口注入是指通过在接口中声明的业务方法来传入具体类的对象。这些方法在定义时使用的是抽象类型，在运行时再传入具体类型的对象，由子类对象来覆盖父类对象。

五、接口隔离原则

使用多个专门的接口，而不使用单一的总接口，即客户端不应该依赖那些它不需要的接口。  
在使用接口隔离原则时，我们需要注意控制接口的粒度，接口不能太小，如果太小会导致系统中接口泛滥，不利于维护；接口也不能太大，太大的接口将违背接口隔离原则，灵活性较差，使用起来很不方便。一般而言，接口中仅包含为某一类用户定制的方法即可，不应该强迫客户依赖于那些它们不用的方法。

六、迪米特法则

一个软件实体应当尽可能少地与其他实体发生相互作用。迪米特法则可降低系统的耦合度，使类与类之间保持松散的耦合关系。迪米特法则要求我们在设计系统时，应该尽量减少对象之间的交互，如果两个对象之间不必彼此直接通信，那么这两个对象就不应当发生任何直接的相互作用，如果其中的一个对象需要调用另一个对象的某一个方法的话，可以通过第三者转发这个调用。简言之，就是通过引入一个合理的第三者来降低现有对象之间的耦合度。

在将迪米特法则运用到系统设计中时，要注意下面的几点：

1）在类的划分上，应当尽量创建松耦合的类，类之间的耦合度越低，就越有利于复用，一个处在松耦合中的类一旦被修改，不会对关联的类造成太大波及；

2）在类的结构设计上，每一个类都应当尽量降低其成员变量和成员函数的访问权限；

3）在类的设计上，只要有可能，一个类型应当设计成不变类；在对其他类的引用上，一个对象对其他对象的引用应当降到最低。

* 常见的单例模式以及各种实现方式的优缺点，哪一种最好，手写常见的单利模式

**饿汉模式、懒汉模式（线程不安全）、懒汉模式（线程安全）、双重检查模式 （DCL）**

**静态内部类单例模式**

public class Singleton {

private Singleton(){

}

public static Singleton getInstance(){

return SingletonHolder.sInstance;

}

private static class SingletonHolder {

private static final Singleton sInstance = new Singleton();

}

}

**枚举单例**

public enum Singleton {

INSTANCE;

public void doSomeThing() {

}

}

### 设计模式在实际场景中的应用

### Spring中用到了哪些设计模式

* 1.工厂模式，这个很明显，在各种BeanFactory以及ApplicationContext创建中都用到了；
* 2.模版模式，这个也很明显，在各种BeanFactory以及ApplicationContext实现中也都用到了；
* 3.代理模式，在Aop实现中用到了JDK的动态代理；
* 4.单例模式，这个比如在创建bean的时候。
* 5.Tomcat中有很多场景都使用到了外观模式，因为Tomcat中有很多不同的组件，每个组件需要相互通信，但又不能将自己内部数据过多地暴露给其他组件。用外观模式隔离数据是个很好的方法。
* 6.策略模式在Java中的应用，这个太明显了，因为Comparator这个接口简直就是为策略模式而生的。[Comparable和Comparator的区别](http://www.cnblogs.com/baizhanshi/p/6201875.html)一文中，详细讲了Comparator的使用。比方说Collections里面有一个sort方法，因为集合里面的元素有可能是复合对象，复合对象并不像基本数据类型，可以根据大小排序，复合对象怎么排序呢？基于这个问题考虑，Java要求如果定义的复合对象要有排序的功能，就自行实现Comparable接口或Comparator接口.
* 7.原型模式：使用原型模式创建对象比直接new一个对象在性能上好得多，因为Object类的clone()方法是一个native方法，它直接操作内存中的二进制流，特别是复制大对象时，性能的差别非常明显。
* 8.迭代器模式：Iterable接口和Iterator接口这两个都是迭代相关的接口，可以这么认为，实现了Iterable接口，则表示某个对象是可被迭代的；Iterator接口相当于是一个迭代器，实现了Iterator接口，等于具体定义了这个可被迭代的对象时如何进行迭代的

### MyBatis中用到了哪些设计模式

装饰模式

最明显的就是cache包下面的实现

Cahe、LoggingCache、LruCache、TransactionalCahe...等

建造者模式

BaseBuilder、XMLMapperBuilder

工厂方法

SqlSessionFactory

适配器模式

Log、LogFactory

动态代理

Plugin

### 你项目中有使用哪些设计模式

### 说说常用开源框架中设计模式使用分析

### 动态代理很重要！！！

## 数据结构

### 树（二叉查找树、平衡二叉树、红黑树、B树、B+树）

二叉查找树：

<https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%8E%92%E5%BA%8F%E6%A0%91/10905079?fromtitle=%E4%BA%8C%E5%8F%89%E6%9F%A5%E6%89%BE%E6%A0%91&fromid=7077965&fr=aladdin>

平衡二叉树：

<https://baike.baidu.com/item/%E5%B9%B3%E8%A1%A1%E4%BA%8C%E5%8F%89%E6%A0%91/10421057?fr=aladdin>

红黑树：

<https://baike.baidu.com/item/%E7%BA%A2%E9%BB%91%E6%A0%91/2413209?fr=aladdin>

<http://www.cnblogs.com/skywang12345/p/3245399.html>

**红黑树的特性**:  
**（1）每个节点或者是黑色，或者是红色。**  
**（2）根节点是黑色。**  
**（3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]**  
**（4）如果一个节点是红色的，则它的子节点必须是黑色的。**  
**（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。**

[](https://images0.cnblogs.com/i/497634/201403/251733282013849.jpg)

对x进行左旋，意味着"将x变成一个左节点"。

B树:

<https://baike.baidu.com/item/B%E6%A0%91/5411672>

定义

**B 树**可以看作是对2-3查找树的一种扩展，即他允许每个节点有M-1个子节点。

* 根节点至少有两个子节点
* 每个节点有M-1个key，并且以升序排列
* 位于M-1和M key的子节点的值位于M-1 和M key对应的Value之间
* 其它节点至少有M/2个子节点

B+树:

<https://www.cnblogs.com/vincently/p/4526560.html>

### 深度有限算法、广度优先算法

### 克鲁斯卡尔算法、普林母算法、迪克拉斯算法

### 什么是一致性Hash及其原理、Hash环问题

### 常见的排序算法和查找算法：快排、折半查找、堆排序等

## 网络/IO基础

### BIO、NIO、AIO的概念

### 什么是长连接和短连接

### Http1.0和2.0相比有什么区别，可参考《[Http 2.0](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247484611&idx=1&sn=66c875392eedff8150633ddcd5d83e7a&chksm=e9c5fd72deb274648a607b9bc39bac34adadd768577b77354f6dc85422691605e210b69eeb7b&scene=21#wechat_redirect)》

### Https的基本概念

### <https://www.cnblogs.com/wqhwe/p/5407468.html>

### 

**四、HTTPS的优点**

　　尽管HTTPS并非绝对安全，掌握根证书的机构、掌握加密算法的组织同样可以进行中间人形式的攻击，但HTTPS仍是现行架构下最安全的解决方案，主要有以下几个好处：

　　（1）使用HTTPS协议可认证用户和服务器，确保数据发送到正确的客户机和服务器；

　　（2）HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全，可防止数据在传输过程中不被窃取、改变，确保数据的完整性。

　　（3）HTTPS是现行架构下最安全的解决方案，虽然不是绝对安全，但它大幅增加了中间人攻击的成本。

　　（4）谷歌曾在2014年8月份调整搜索引擎算法，并称“比起同等HTTP网站，采用HTTPS加密的网站在搜索结果中的排名将会更高”。

**五、HTTPS的缺点**

　　虽然说HTTPS有很大的优势，但其相对来说，还是存在不足之处的：

　　（1）HTTPS协议握手阶段比较费时，会使页面的加载时间延长近50%，增加10%到20%的耗电；

　　（2）HTTPS连接缓存不如HTTP高效，会增加数据开销和功耗，甚至已有的安全措施也会因此而受到影响；

　　（3）SSL证书需要钱，功能越强大的证书费用越高，个人网站、小网站没有必要一般不会用。

　   （4）SSL证书通常需要绑定IP，不能在同一IP上绑定多个域名，IPv4资源不可能支撑这个消耗。

　　（5）HTTPS协议的加密范围也比较有限，在黑客攻击、拒绝服务攻击、服务器劫持等方面几乎起不到什么作用。最关键的，SSL证书的信用链体系并不安全，特别是在某些国家可以控制CA根证书的情况下，中间人攻击一样可行。

### 三次握手和四次挥手、为什么挥手需要四次

<https://blog.csdn.net/zixiaomuwu/article/details/60965466>

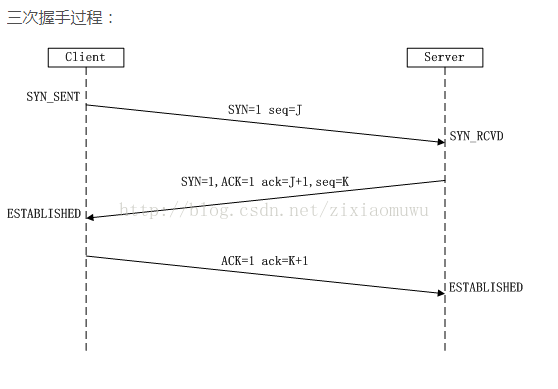
答：三次握手的目的是建立可靠的通信信道，说到通讯，简单来说就是数据的发送与接收，而三次握手最主要的目的就是双方确认自己与对方的发送与接收机能正常。

        第一次握手：Client什么都不能确认；Server确认了对方发送正常

        第二次握手：Client确认了：自己发送、接收正常，对方发送、接收正常；Server确认了：自己接收正常，对方发送正常

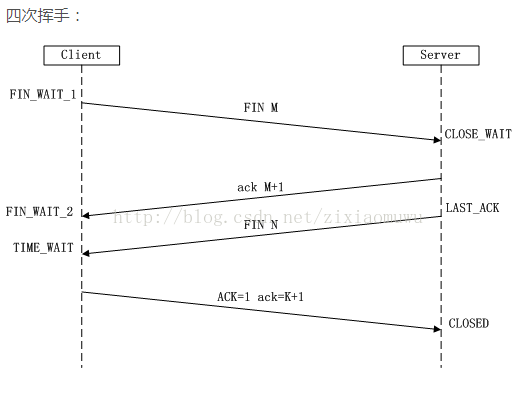
        第三次握手：Client确认了：自己发送、接收正常，对方发送、接收正常；Server确认了：自己发送、接收正常，对方发送接收正常

所以三次握手就能确认双发收发功能都正常，缺一不可。



答：根本原因是，一方发送FIN只表示自己发完了所有要发的数据，但还允许对方继续把没发完的数据发过来。

        举个例子：A和B打电话，通话即将结束后，A说“我没啥要说的了”，B回答“我知道了”，但是B可能还会有要说的话，A不能要求B跟着自己的节奏结束通话，于是B可能又巴拉巴拉说了一通，最后B说“我说完了”，A回答“知道了”，这样通话才算结束。



* 从游览器中输入URL到页面加载的发生了什么？可参考《[从输入URL到页面加载发生了什么](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247483724&idx=1&sn=e58dd30d124971c795584e8673d6cc71&chksm=e9c5f8fddeb271ebebbb6c350ed1abc252f1f26b4f35c4ce36e10bde9659a37520feabed2290&scene=21#wechat_redirect)》

可以分为这几个大的过程：

1. DNS解析
2. TCP连接
3. 客户端发送HTTP请求
4. 服务器处理请求并返回HTTP报文
5. 浏览器解析渲染页面
6. 结束

# 数据存储和消息队列

## 数据库

* MySQL 索引使用的注意事项

<https://www.cnblogs.com/heyonggang/p/6610526.html>

<https://blog.csdn.net/qq_26963433/article/details/78631487>

<https://www.cnblogs.com/xyhero/p/b0ad525c6a6a5ed2bd7f40918c5dbd98.html>

索引虽然好处很多，但过多的使用索引可能带来相反的问题，索引也是有缺点的：

虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT,UPDATE和DELETE。因为更新表时，mysql不仅要保存数据，还要保存一下索引文件

建立索引会占用磁盘空间的索引文件。一般情况这个问题不太严重，但如果你在要给大表上建了多种组合索引，索引文件会膨胀很宽

索引只是提高效率的一个方式，如果mysql有大数据量的表，就要花时间研究建立最优的索引，或优化查询语句。

使用索引时，有一些技巧：

1.索引不会包含有NULL的列

只要列中包含有NULL值，都将不会被包含在索引中，复合索引中只要有一列含有NULL值，那么这一列对于此符合索引就是无效的。

2.使用短索引

对串列进行索引，如果可以就应该指定一个前缀长度。例如，如果有一个char（255）的列，如果在前10个或20个字符内，多数值是唯一的，那么就不要对整个列进行索引。短索引不仅可以提高查询速度而且可以节省磁盘空间和I/O操作。

3.索引列排序

mysql查询只使用一个索引，因此如果where子句中已经使用了索引的话，那么order by中的列是不会使用索引的。因此数据库默认排序可以符合要求的情况下不要使用排序操作，尽量不要包含多个列的排序，如果需要最好给这些列建复合索引。

4.like语句操作

一般情况下不鼓励使用like操作，如果非使用不可，注意正确的使用方式。like ‘%aaa%’不会使用索引，而like ‘aaa%’可以使用索引。

5.不要在列上进行运算

6.不使用NOT IN 、<>、！=操作，但<,<=，=，>,>=,BETWEEN,IN是可以用到索引的

7.索引要建立在经常进行select操作的字段上。

这是因为，如果这些列很少用到，那么有无索引并不能明显改变查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

8.索引要建立在值比较唯一的字段上。

9.对于那些定义为text、image和bit数据类型的列不应该增加索引。因为这些列的数据量要么相当大，要么取值很少。

10.在where和join中出现的列需要建立索引。

11.where的查询条件里有不等号(where column != …),mysql将无法使用索引。

12.如果where字句的查询条件里使用了函数(如：where DAY(column)=…),mysql将无法使用索引。

13.在join操作中(需要从多个数据表提取数据时)，mysql只有在主键和外键的数据类型相同时才能使用索引，否则及时建立了索引也不会使用。

* DDL、DML、DCL分别指什么

<https://www.cnblogs.com/dato/p/7049343.html>

<https://blog.csdn.net/sinat_25059791/article/details/69666318>

DML（data manipulation language）数据操纵语言：

就是我们最经常用到的 SELECT、UPDATE、INSERT、DELETE。 主要用来对数据库的数据进行一些操作。

SELECT 列名称 FROM 表名称[也可以叫做**DQL**]

UPDATE 表名称 SET 列名称 = 新值 WHERE 列名称 = 某值

INSERT INTO table\_name (列1, 列2,...) VALUES (值1, 值2,....)

DELETE FROM 表名称 WHERE 列名称 = 值

DDL（data definition language）数据库定义语言：

其实就是我们在创建表的时候用到的一些sql，比如说：CREATE、ALTER、DROP等。DDL主要是用在定义或改变表的结构，数据类型，表之间的链接和约束等初始化工作上

CREATE TABLE 表名称

(....

)

ALTER TABLE table\_name

ALTER COLUMN column\_name datatype

DROP TABLE 表名称

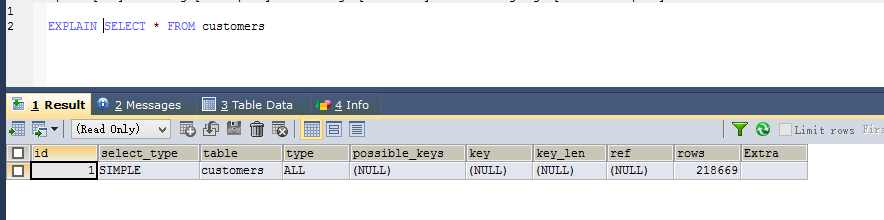
DROP DATABASE 数据库名称

DCL（Data Control Language）数据库控制语言：

是用来设置或更改数据库用户或角色权限的语句，包括（grant,deny,revoke等）语句。这个比较少用到。

* explain命令

<https://www.cnblogs.com/gomysql/p/3720123.html>



1. id：

包含一组数字，表示查询中执行select子句或操作表的顺序

Example（id相同，执行顺序由上至下）

2. select\_type

表示查询中每个select子句的类型（简单OR复杂）

a. SIMPLE：查询中不包含子查询或者UNION

b. 查询中若包含任何复杂的子部分，最外层查询则被标记为：PRIMARY

c. 在SELECT或WHERE列表中包含了子查询，该子查询被标记为：SUBQUERY

d. 在FROM列表中包含的子查询被标记为：DERIVED（衍生）用来表示包含在from子句中的子查询的select，mysql会递归执行并将结果放到一个临时表中。服务器内部称为"派生表"，因为该临时表是从子查询中派生出来的

e. 若第二个SELECT出现在UNION之后，则被标记为UNION；若UNION包含在FROM子句的子查询中，外层SELECT将被标记为：DERIVED

f. 从UNION表获取结果的SELECT被标记为：UNION RESULT

SUBQUERY和UNION还可以被标记为DEPENDENT和UNCACHEABLE。

DEPENDENT意味着select依赖于外层查询中发现的数据。

UNCACHEABLE意味着select中的某些 特性阻止结果被缓存于一个item\_cache中。

3. type

表示MySQL在表中找到所需行的方式，又称“访问类型”，常见类型如下:

ALL, index, range, ref, eq\_ref, const, system, NULL

从左到右，性能从最差到最好

**a. ALL：Full Table Scan， MySQL将遍历全表以找到匹配的行**

**EXPLAIN SELECT \* FROM customers**

**b. index：Full Index Scan，index与ALL区别为index类型只遍历索引树**

**c. range:索引范围扫描，对索引的扫描开始于某一点，返回匹配值域的行。显而易见的索引范围扫描是带有between或者where子句里带有<, >查询。当mysql使用索引去查找一系列值时，例如IN()和OR列表，也会显示range（范围扫描）,当然性能上面是有差异的。**

**EXPLAIN SELECT \* FROM customers WHERE id = '209825' OR id = '209827' ;**

**EXPLAIN SELECT \* FROM customers WHERE id BETWEEN 209825 AND 209827**

**EXPLAIN SELECT \* FROM customers WHERE id > 209825 AND id < 209829**

**d. ref：使用非唯一索引扫描或者唯一索引的前缀扫描，返回匹配某个单独值的记录行**

**e. eq\_ref：类似ref，区别就在使用的索引是唯一索引，对于每个索引键值，表中只有一条记录匹配，简单来说，就是多表连接中使用primary key或者 unique key作为关联条件**

**f. const、system：当MySQL对查询某部分进行优化，并转换为一个常量时，使用这些类型访问。如将主键置于where列表中，MySQL就能将该查询转换为一个常量**

**EXPLAIN SELECT \* FROM customers WHERE id = '209825';**

**EXPLAIN SELECT \* FROM customers WHERE id = (SELECT MIN(id) FROM customers );**

**g. NULL：MySQL在优化过程中分解语句，执行时甚至不用访问表或索引，例如从一个索引列里选取最小值可以通过单独索引查找完成。**

**EXPLAIN SELECT MIN(id) FROM customers**

* left join，right join，inner join

参考链接：<https://www.cnblogs.com/pcjim/articles/799302.html>

left join(左联接) 返回包括左表中的所有记录和右表中联结字段相等的记录

right join(右联接) 返回包括右表中的所有记录和左表中联结字段相等的记录

inner join(等值连接) 只返回两个表中联结字段相等的行

* 数据库事物ACID（原子性、一致性、隔离性、持久性）

<https://www.2cto.com/database/201710/691297.html>

<https://baike.baidu.com/item/acid/10738>

ACID，指数据库事务正确执行的四个基本要素的缩写。包含：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。一个支持事务（Transaction）的数据库，必须要具有这四种[特性](https://baike.baidu.com/item/%E7%89%B9%E6%80%A7/3128227)，否则在事务过程（Transaction processing）当中无法保证数据的正确性，交易过程极可能达不到交易方的要求。

原子性

整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被[回滚](https://baike.baidu.com/item/%E5%9B%9E%E6%BB%9A)（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。

一致性

一个事务可以封装状态改变（除非它是一个只读的）。事务必须始终保持系统处于一致的状态，不管在任何给定的时间[并发](https://baike.baidu.com/item/%E5%B9%B6%E5%8F%91)事务有多少。

也就是说：如果事务是[并发](https://baike.baidu.com/item/%E5%B9%B6%E5%8F%91)多个，系统也必须如同串行事务一样操作。其主要特征是保护性和不变性(Preserving an Invariant)，以转账[案例](https://baike.baidu.com/item/%E6%A1%88%E4%BE%8B)为例，假设有五个账户，每个账户余额是100元，那么五个账户总额是500元，如果在这个5个账户之间同时发生多个转账，无论[并发](https://baike.baidu.com/item/%E5%B9%B6%E5%8F%91)多少个，比如在A与B账户之间转账5元，在C与D账户之间转账10元，在B与E之间转账15元，五个账户总额也应该还是500元，这就是保护性和不变性。

隔离性

隔离状态执行事务，使它们好像是系统在给定时间内执行的唯一操作。如果有两个事务，运行在相同的时间内，执行相同的功能，事务的隔离性将确保每一事务在系统中认为只有该事务在使用系统。这种属性有时称为[串行](https://baike.baidu.com/item/%E4%B8%B2%E8%A1%8C)化，为了防止事务操作间的混淆，必须串行化或序列化请求，使得在同一时间仅有一个请求用于同一数据。

持久性

在事务完成以后，该事务对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

* 事物的隔离级别（读未提交、读以提交、可重复读、可序列化读）

<https://www.2cto.com/database/201710/691297.html>

<https://blog.csdn.net/Somhu/article/details/78775198>

隔离性的四个级别

READ UNCOMMITTED（未提交读）

在READ UNCOMMITTED级别，事务中的修改，即使没有提交，对其他事务也都是可见的。事务可以读取未提交的数据，这也被称为脏读（Dirty Read）。这个级别会导致很多问题，从性能上来说，READ UNCOMMITTED不会比其他的级别好太多，但却缺乏其他级别的很多好处，除非真的有非常必要的理由，在实际应用中一般很少使用。

READ COMMITTED（提交读）

大多数数据库系统的默认隔离级别都是READ COMMTTED（但MySQL不是）。READ COMMITTED满足前面提到的隔离性的简单定义：一个事务开始时，只能”看见”已经提交的事务所做的修改。换句话说，一个事务从开始直到提交之前，所做的任何修改对其他事务都是不可见的。这个级别有时候叫做不可重复读（nonrepeatble read），因为两次执行同样的查询，可能会得到不一样的结果。

REPEATABLE READ(可重复读)

REPEATABLE READ解决了脏读的问题。该隔离级别保证了在同一个事务中多次读取同样记录结果是一致的。但是理论上，可重复读隔离级别还是无法解决另外一个幻读（Phantom Read）的问题。所谓幻读，指的是当某个事务在读取某个范围内的记录时，另一个事务又在该范围内插入了新的记录，当之前的事务再次读取该范围的记录时，会产生幻行（Phantom Row）。InnoDB和XtraDB存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）解决了幻读的问题。

SERIALIZABLE（串行化）

SERIALIZABLE是最高的隔离级别。它通过强制事务串行执行，避免了前面说的幻读的问题。简单来说，SERIALIZABLE会在读取每一行数据都加锁，所以可能导致大量的超时和锁争用问题。实际应用中也很少用到这个隔离级别，只有在非常需要确保数据的一致性而且可以接受没有并发的情况下，才考虑采用该级别。

* 脏读、幻读、不可重复读

<https://blog.csdn.net/Somhu/article/details/78775198>

也许有很多读者会对上述隔离级别中提及到的 **脏读、不可重复读、幻读** 的理解有点吃力，我在这里尝试使用通俗的方式来解释这三种语义：

**脏读：**所谓的脏读，其实就是读到了别的事务回滚前的脏数据。比如事务B执行过程中修改了数据X，在未提交前，事务A读取了X，而事务B却回滚了，这样事务A就形成了脏读。

也就是说，当前事务读到的数据是别的事务想要修改成为的但是没有修改成功的数据。

**不可重复读：**事务A首先读取了一条数据，然后执行逻辑的时候，事务B将这条数据改变了，然后事务A再次读取的时候，发现数据不匹配了，就是所谓的不可重复读了。

也就是说，当前事务先进行了一次数据读取，然后再次读取到的数据是别的事务修改成功的数据，导致两次读取到的数据不匹配，也就照应了不可重复读的语义。

**幻读：**事务A首先根据条件索引得到N条数据，然后事务B改变了这N条数据之外的M条或者增添了M条符合事务A搜索条件的数据，导致事务A再次搜索发现有N+M条数据了，就产生了幻读。

也就是说，当前事务读第一次取到的数据比后来读取到数据条目少。

不可重复读和幻读比较：

两者有些相似，但是前者针对的是update或delete，后者针对的insert。

* 数据库的几大范式

<https://blog.csdn.net/qq_29232943/article/details/62421776>

<https://www.cnblogs.com/lca1826/p/6601395.html>

 第一范式（1NF）无重复的列

    所谓第一范式（1NF）是指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。在第一范式（1NF）中表的每一行只包含一个实例的信息。简而言之，第一范式就是无重复的列。

说明：在任何一个关系数据库中，第一范式（1NF）是对关系模式的基本要求，不满足第一范式（1NF）的数据库就不是关系数据库。

**第一范式（1NF）：属性不可分。**

在前面已经介绍了属性值的概念，我们说，它是“不可分的”。而第一范式要求属性也不可分

第二范式（2NF）属性完全依赖于主键 [ 消除部分子函数依赖 ]

如果关系模式R为第一范式，并且R中每一个非主属性完全函数依赖于R的某个候选键， 则称为第二范式模式。

第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。第二范式（2NF）要求数据库表中的每个实例或行必须可以被惟一地区分。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。这个惟一属性列被称为主关键字或主键、主码。

**第二范式（2NF）：符合1NF，并且，非主属性完全依赖于码。**（注意是完全依赖不能是部分依赖，设有函数依赖W→A，若存在XW，有X→A成立，那么称W→A是局部依赖，否则就称W→A是完全函数依赖）

第三范式（3NF）属性不依赖于其它非主属性 [ 消除传递依赖 ]

如果关系模式R是第二范式，且每个非主属性都不传递依赖于R的候选键，则称R为第三范式模式。

满足第三范式（3NF）必须先满足第二范式（2NF）。第三范式（3NF）要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。

**第三范式（3NF）：符合2NF，并且，消除传递依赖**（也就是每个非主属性都不传递依赖于候选键，判断传递函数依赖，指的是如果存在"A → B → C"的决定关系，则C传递函数依赖于A。）

考虑这样一个表：【联系人】（姓名，性别，电话）

如果在实际场景中，一个联系人有家庭电话和公司电话，那么这种表结构设计就没有达到 1NF。要符合 1NF 我们只需把列（电话）拆分，即：【联系人】（姓名，性别，家庭电话，公司电话）。1NF 很好辨别，但是 2NF 和 3NF 就容易搞混淆。

◆ 第二范式（2NF）：首先是 1NF，另外包含两部分内容，一是表必须有一个主键；二是没有包含在主键中的列必须完全依赖于主键，而不能只依赖于主键的一部分。

考虑一个订单明细表：【OrderDetail】（OrderID，ProductID，UnitPrice，Discount，Quantity，ProductName）。

因为我们知道在一个订单中可以订购多种产品，所以单单一个 OrderID 是不足以成为主键的，主键应该是（OrderID，ProductID）。显而易见 Discount（折扣），Quantity（数量）完全依赖（取决）于主键（OderID，ProductID），而 UnitPrice，ProductName 只依赖于 ProductID。所以 OrderDetail 表不符合 2NF。不符合 2NF 的设计容易产生冗余数据。

可以把【OrderDetail】表拆分为【OrderDetail】（OrderID，ProductID，Discount，Quantity）和【Product】（ProductID，UnitPrice，ProductName）来消除原订单表中UnitPrice，ProductName多次重复的情况。

◆ 第三范式（3NF）：首先是 2NF，另外非主键列必须直接依赖于主键，不能存在传递依赖。即不能存在：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况。

考虑一个订单表【Order】（OrderID，OrderDate，CustomerID，CustomerName，CustomerAddr，CustomerCity）主键是（OrderID）。

其中 OrderDate，CustomerID，CustomerName，CustomerAddr，CustomerCity 等非主键列都完全依赖于主键（OrderID），所以符合 2NF。不过问题是 CustomerName，CustomerAddr，CustomerCity 直接依赖的是 CustomerID（非主键列），而不是直接依赖于主键，它是通过传递才依赖于主键，所以不符合 3NF。

通过拆分【Order】为【Order】（OrderID，OrderDate，CustomerID）和【Customer】（CustomerID，CustomerName，CustomerAddr，CustomerCity）从而达到 3NF。

* 数据库常见的命令

<https://blog.csdn.net/u010028461/article/details/79722736>

SQL：Structured Query Language，结构化查询语言（数据以查询为主，99% 都是在进行查询操作）。

SQL 主要分为三种：

• DDL：Data Definition Language，数据定义语言，用来维护存储数据的结构（数据库、表），代表指令为create、drop和alter等。

• DML：Data Manipulation Language，数据操作语言，用来对数据进行操作（表中的内容）代表指令为insert、delete和update等，不过在 DML 内部又单独进行了一个分类，即 DQL（Data Query Language），数据查询语言，代表指令为select.

• DCL：Data Control Language，数据控制语言，主要是负责（用户）权限管理，代表指令为grant和revoke等。

* 说说分库与分表设计
* 分库与分表带来的分布式困境与应对之策（如何解决分布式下的分库分表，全局表？）

##### 说说 SQL 优化之道

<https://www.cnblogs.com/yycc/p/7338894.html>

explain解释

<https://blog.csdn.net/jie_liang/article/details/77340905>

1.对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

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.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。

4.应尽量避免在 where 子句中使用 or 来连接条件，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=10 or num=20

可以这样查询：

select id from t where num=10

union all

select id from t where num=20

5.in 和 not in 也要慎用，否则会导致全表扫描，如：

select id from t where num in(1,2,3)

对于连续的数值，能用 between 就不要用 in 了：

select id from t where num between 1 and 3

6.下面的查询也将导致全表扫描：

select id from t where name like '%abc%'

7.应尽量避免在 where 子句中对字段进行表达式操作，这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/2=100

应改为:

select id from t where num=100\*2

8.应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where substring(name,1,3)='abc'--name以abc开头的id

应改为:

select id from t where name like 'abc%'

9.不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

10.在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，

否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

11.不要写一些没有意义的查询，如需要生成一个空表结构：

select col1,col2 into #t from t where 1=0

这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：

create table #t(...)

12.很多时候用 exists 代替 in 是一个好的选择：

select num from a where num in(select num from b)

用下面的语句替换：

select num from a where exists(select 1 from b where num=a.num)

13.并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，

如一表中有字段sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。

14.索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，

因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。

一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

15.尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。

这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

16.尽可能的使用 varchar 代替 char ，因为首先变长字段存储空间小，可以节省存储空间，

其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

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

18.避免频繁创建和删除临时表，以减少系统表资源的消耗。

19.临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。

20.在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，

以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

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

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

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

24.与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。

在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

25.尽量避免大事务操作，提高系统并发能力。26.尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

<https://blog.csdn.net/wuhuagu_wuhuaguo/article/details/72875054>

* MySQL遇到的死锁问题、如何排查与解决

<https://blog.csdn.net/mine_song/article/details/71106410>

MySQL有三种锁的级别：页级、表级、行级。

表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高,并发度最低。

行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低,并发度也最高。

页面锁：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般

Demo:

<https://www.cnblogs.com/sivkun/p/7518540.html>

**死锁的第一种情况**

一个用户A 访问表A(锁住了表A),然后又访问表B；另一个用户B 访问表B(锁住了表B)，然后企图访问表A；这时用户A由于用户B已经锁住表B，它必须等待用户B释放表B才能继续，同样用户B要等用户A释放表A才能继续，这就死锁就产生了。

解决方法：

这种死锁比较常见，是由于程序的BUG产生的，除了调整的程序的逻辑没有其它的办法。仔细分析程序的逻辑，对于数据库的多表操作时，尽量按照相同的顺序进 行处理，尽量避免同时锁定两个资源，如操作A和B两张表时，总是按先A后B的顺序处理， 必须同时锁定两个资源时，要保证在任何时刻都应该按照相同的顺序来锁定资源。

**死锁的第二种情况**

用户A查询一条纪录，然后修改该条纪录；这时用户B修改该条纪录，这时用户A的事务里锁的性质由查询的共享锁企图上升到独占锁，而用户B里的独占锁由于A 有共享锁存在所以必须等A释放掉共享锁，而A由于B的独占锁而无法上升的独占锁也就不可能释放共享锁，于是出现了死锁。这种死锁比较隐蔽，但在稍大点的项 目中经常发生。如在某项目中，页面上的按钮点击后，没有使按钮立刻失效，使得用户会多次快速点击同一按钮，这样同一段代码对数据库同一条记录进行多次操 作，很容易就出现这种死锁的情况。

**死锁的第三种情况**

如果在事务中执行了一条不满足条件的update语句，则执行全表扫描，把行级锁上升为表级锁，多个这样的事务执行后，就很容易产生死锁和阻塞。类似的情 况还有当表中的数据量非常庞大而索引建的过少或不合适的时候，使得经常发生全表扫描，最终应用系统会越来越慢，最终发生阻塞或死锁。

**案例2：**

在开发中，经常会做这类的判断需求：根据字段值查询（有索引），如果不存在，则插入；否则更新。

查看死锁：

show engine innodb status

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 读数据一致性 | 脏读 | 不可重复读 | 幻读 |
| 未提交读 | 最低级别 | 是 | 是 | 是 |
| 已提交读 | 语句级 | 否 | 是 | 是 |
| 可重复读 | 事务级 | 否 | 否 | 是 |
| 可序列化 | 最高级别，事务级 | 否 | 否 | 否 |

* 存储引擎的 InnoDB与MyISAM区别，优缺点，使用场景

<https://blog.csdn.net/qq_27607965/article/details/79925288>

* 索引类别（B+树索引、全文索引、哈希索引）、索引的原理
* 什么是自适应哈希索引（AHI）
* 为什么要用 B+tree作为MySQL索引的数据结构
* 聚集索引与非聚集索引的区别
* 遇到过索引失效的情况没，什么时候可能会出现，如何解决

<https://www.cnblogs.com/areyouready/p/7802885.html>

**隐式转换导致索引失效.这一点应当引起重视.也是开发中经常会犯的错误.**

**对索引列进行运算导致索引失效,我所指的对索引列进行运算包括(+，-，\*，/，! 等)**

* limit 20000 加载很慢怎么解决

1、子查询优化法

在子查询里先从索引获取到较大id，然后倒序排，再取10行结果集

2、表连接优化法

通过索引构造一个小表，然后同数据进行关联，由于在索引中只取想要的ID所以这个会非常快，然后把取出来的很少的ID再同数据表关系，这样能大大提高查询效率。

* 如何选择合适的分布式主键方案

<https://www.cnblogs.com/baiwa/p/5318432.html>

auto\_increment

 UUID

基于redis的分布式ID生成器

MongoDB文档（Document）全局唯一ID

* 选择合适的数据存储方案
* 常见的几种分布式ID的设计方案
* 常见的数据库优化方案，在你的项目中数据库如何进行优化的

<https://www.cnblogs.com/zhyunfe/p/6209074.html>

<https://zhidao.baidu.com/question/67759076.html>

1、调整数据结构的设计。这一部分在开发信息系统之前完成，程序员需要考虑是否使用ORACLE数据库的分区功能，对于经常访问的数据库表是否需要建立索引等。

2、调整应用程序结构设计。这一部分也是在开发信息系统之前完成，程序员在这一步需要考虑应用程序使用什么样的体系结构，是使用传统的Client/Server两层体系结构，还是使用Browser/Web/Database的三层体系结构。不同的应用程序体系结构要求的数据库资源是不同的。

3、调整数据库SQL语句。应用程序的执行最终将归结为数据库中的SQL语句执行，因此SQL语句的执行效率最终决定了ORACLE数据库的性能。ORACLE公司推荐使用ORACLE语句优化器（Oracle Optimizer）和行锁管理器（row-level manager）来调整优化SQL语句。

4、调整服务器内存分配。内存分配是在信息系统运行过程中优化配置的，数据库管理员可以根据数据库运行状况调整数据库系统全局区（SGA区）的数据缓冲区、日志缓冲区和共享池的大小；还可以调整程序全局区（PGA区）的大小。需要注意的是，SGA区不是越大越好，SGA区过大会占用操作系统使用的内存而引起虚拟内存的页面交换，这样反而会降低系统。

5、调整硬盘I/O，这一步是在信息系统开发之前完成的。数据库管理员可以将组成同一个表空间的数据文件放在不同的硬盘上，做到硬盘之间I/O负载均衡。

6、调整操作系统参数，例如：运行在UNIX操作系统上的ORACLE数据库，可以调整UNIX数据缓冲池的大小，每个进程所能使用的内存大小等参数。

## Redis

* Redis 有哪些数据类型，可参考《[Redis常见的5种不同的数据类型详解](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247483987&idx=1&sn=5c5e4cd5bc73a7e6f84e5d6adfab0935&chksm=e9c5fbe2deb272f4b5b75bd2ac92bb27950452623ec83c0e1add7e30c773160421fab1571680&scene=21#wechat_redirect)》

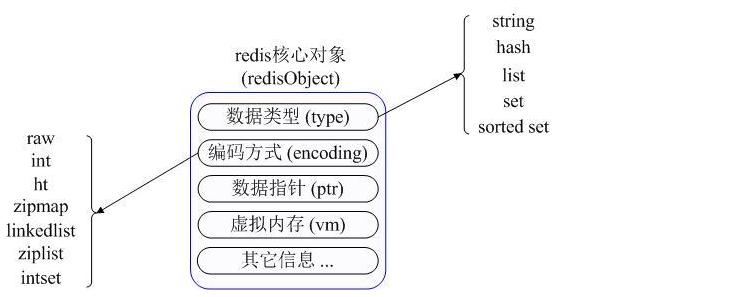
字符串(String), 哈希(Map), 列表(list), 集合(sets) 和 有序集合(sorted sets)等类型。

* Redis 内部结构

<https://www.cnblogs.com/wykCN/p/4533231.html>

<https://www.cnblogs.com/xiaoxi/p/7007695.html>

ht(dict),raw,embstr,intset,sds,ziplist,quicklist,skiplist



* Redis 使用场景

<https://www.cnblogs.com/xiaoxi/p/7007695.html>

**显示最新的项目列表**

**排行榜应用，取TOP N操作**

**删除与过滤**

**按照用户投票和时间排序**

**处理过期项目**

**计数**

**特定时间内的特定项目**

**查找某个值所在的区间(区间无重合) ：(Sorted Set)**

**交集，并集，差集：(Set)**

* Redis 持久化机制，可参考《[使用快照和AOF将Redis数据持久化到硬盘中](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247483992&idx=1&sn=8f554bc490c4db1a78a30144f873e911&chksm=e9c5fbe9deb272fff47483c241e6d2a7aae99dc8f6fe9fee31f2dd214d0cf81b33d51f7a7dbe&scene=21#wechat_redirect)》

<https://www.cnblogs.com/xingzc/p/5988080.html>

**首先我们来看一下数据库在进行写操作时到底做了哪些事，主要有下面五个过程**：

* 客户端向服务端发送写操作（数据在客户端的内存中）。
* 数据库服务端接收到写请求的数据（数据在服务端的内存中）。
* 服务端调用write这个系统调用，将数据往磁盘上写（数据在系统内存的缓冲区中）。
* 操作系统将缓冲区中的数据转移到磁盘控制器上（数据在磁盘缓存中）。
* 磁盘控制器将数据写到磁盘的物理介质中（数据真正落到磁盘上）。

**Redis提供了RDB持久化和AOF持久化**

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘。

　　也是默认的持久化方式，这种方式是就是将内存中数据以快照的方式写入到二进制文件中,默认的文件名为dump.rdb。

可以通过配置设置自动做快照持久化的方式。我们可以配置redis在n秒内如果超过m个key被修改就自动做快照，下面是默认的快照保存配置

save 900 1 #900秒内如果超过1个key被修改，则发起快照保存

save 300 10 #300秒内容如超过10个key被修改，则发起快照保存

save 60 10000

#### 优势

* 一旦采用该方式，那么你的整个Redis数据库将只包含一个文件，这样非常方便进行备份。比如你可能打算没1天归档一些数据。
* 方便备份，我们可以很容易的将一个一个RDB文件移动到其他的存储介质上
* RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。
* RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 fork 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。

#### 劣势

* 如果你需要尽量避免在服务器故障时丢失数据，那么 RDB 不适合你。 虽然 Redis 允许你设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 因为RDB 文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少 5 分钟才保存一次 RDB 文件。 在这种情况下， 一旦发生故障停机， 你就可能会丢失好几分钟的数据。
* 每次保存 RDB 的时候，Redis 都要 fork() 出一个子进程，并由子进程来进行实际的持久化工作。 在数据集比较庞大时， fork() 可能会非常耗时，造成服务器在某某毫秒内停止处理客户端； 如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。 虽然 AOF 重写也需要进行 fork() ，但无论 AOF 重写的执行间隔有多长，数据的耐久性都不会有任何损失。

#### AOF文件保存过程

redis会将每一个收到的写命令都通过write函数追加到文件中(默认是 appendonly.aof)。

当redis重启时会通过重新执行文件中保存的写命令来在内存中重建整个数据库的内容。当然由于os会在内核中缓存 write做的修改，所以可能不是立即写到磁盘上。这样aof方式的持久化也还是有可能会丢失部分修改。不过我们可以通过配置文件告诉redis我们想要 通过fsync函数强制os写入到磁盘的时机。有三种方式如下（默认是：每秒fsync一次）

appendonly yes //启用aof持久化方式

# appendfsync always //每次收到写命令就立即强制写入磁盘，最慢的，但是保证完全的持久化，不推荐使用

appendfsync everysec //每秒钟强制写入磁盘一次，在性能和持久化方面做了很好的折中，推荐

# appendfsync no //完全依赖os，性能最好,持久化没保证

<https://www.cnblogs.com/bamaofan/p/5284014.html>

RDB持久化是指用数据集快照的方式记录redis数据库的所有键值对。

  两个命令：SAVE命令会阻塞主进程来完成写文件，BGSAVE命令会创建子进程来完成写文件，主进程会继续处理命令。

  优点：

　　1.只有一个文件dump.rdb，方便持久化。

　　2.容灾性好，一个文件可以保存到安全的磁盘。

　　3.性能最大化，fork子进程来完成写操作，让主进程继续处理命令，所以是IO最大化。

　　4.相对于数据集大时，比AOF的启动效率更高。

  缺点：

　　1.数据安全性低，通过配置save参数来达到定时的写快照，比如 每900 秒有1个键被修改就进行一次快照，每600秒至少有10个键被修改进行快照，每30秒有至少10000个键被修改进行记录。所以如果当服务器还在等待写快照时出现了宕机，那么将会丢失数据。

　　2.fork子进程时可能导致服务器停机1秒，数据集太大。

AOF持久化是指所有的命令行记录以redis命令请求协议的格式保存为aof文件。

  优点：

　　1.数据安全，aof持久化可以配置appendfsync属性，有always，每进行一次命令操作就记录到aof文件中一次；everySec，就是每秒内进行一次文件的写操作；no就是不进行aof文件的写操作。

　　2.通过append模式写文件，即使中途服务器宕机，可以通过redis-check-aof工具解决数据一致性问题。

　　3.AOF机制的rewrite模式，用来将过大的aof文件缩小，实现原理是将所有的set 通过一句set 命令总结，所有的SADD命令用总结为一句，这样每种命令都概括为一句来执行，就可以减少aof文件的大小了。（注意，在重写的过程中，是创建子进程来完成重写操作，主进程每个命令都会在AOF缓冲区和AOF重写缓冲区进行保存，这样旧版aof文件可以实现数据最新，当更新完后将重写缓冲区中的数据写入新的aof文件中然后就可以将新的文件替换掉旧版的文件。

  缺点：

　　1.文件会比RDB形式的文件大。

　　2.数据集大的时候，比rdb启动效率低。

* Redis 集群方案与实现

<https://github.com/bage2014/study/blob/master/study-redis/README.md>

* Redis 为什么是单线程的

<https://www.cnblogs.com/yuyutianxia/p/6346723.html>

单纯的网络IO来说，量大到一定程度之后，多线程的确有优势, 但并不是单纯的多线程，而是每个线程自己有自己的epoll这样的模型，

也就是多线程和multiplexing混合。

但是。还要考虑Redis操作的是内存中的数据结构。如果在多线程中操作，那就需要为这些对象加锁。

所以使用多线程可以提高性能，但是每个线程的效率严重下降了，而且程序的逻辑严重复杂化。Redis的数据结构并不全是简单的Key-Value，

还有list，hash等复杂的结构，这些结构有可能会进行很细粒度的操作，比如在很长的列表后面添加一个元素，在hash当中添加或者删除

一个对象，这些操作还可以合成MULTI/EXEC的组。这样一个操作中可能就需要加非常多的锁，导致的结果是同步开销大大增加。Redis

在权衡之后的选择是用单线程，突出自己功能的灵活性。在单线程基础上任何原子操作都可以几乎无代价地实现，多么复杂的数据结构都

可以轻松运用，甚至可以使用Lua脚本这样的功能。对于多线程来说这需要高得多的代价。

      并不是所有的KV数据库或者内存数据库都应该用单线程，比如ZooKeeper就是多线程的，最终还是看作者自己的意愿和取舍。单线程

的威力实际上非常强大，每核心效率也非常高，在今天的虚拟化环境当中可以充分利用云化环境来提高资源利用率。多线程自然是可以比单

线程有更高的性能上限，但是在今天的计算环境中，即使是单机多线程的上限也往往不能满足需要了，需要进一步摸索的是多服务器集群化

的方案，这些方案中多线程的技术照样是用不上的，所以单线程、多进程的集群不失为一个时髦的解决方案。

<http://www.cnblogs.com/syyong/p/6231326.html>

单进程单线程好处

代码更清晰，处理逻辑更简单

不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗

不存在多进程或者多线程导致的切换而消耗CPU

单进程单线程弊端

无法发挥多核CPU性能，不过可以通过在单机开多个Redis实例来完善；

* 缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级

<https://www.cnblogs.com/leeSmall/p/8594542.html>

<https://blog.csdn.net/qianshangding0708/article/details/48024239>

**缓存雪崩**

我们可以简单的理解为：由于原有缓存失效，新缓存未到期间(例如：我们设置缓存时采用了相同的过期时间，在同一时刻出现大面积的缓存过期)，所有原本应该访问缓存的请求都去查询数据库了，而对数据库CPU和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。

缓存雪崩的解决方案：

1、碰到这种情况，一般并发量不是特别多的时候，使用最多的解决方案是加锁排队

2、给每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存

3、增加一个时间差去访问数据库

4、二级缓存

1，采用加锁计数，或者使用合理的队列数量来避免缓存失效时对数据库造成太大的压力。这种办法虽然能缓解数据库的压力，但是同时又降低了系统的吞吐量。

2，分析用户行为，尽量让失效时间点均匀分布。避免缓存雪崩的出现。

3，如果是因为某台缓存服务器宕机，可以考虑做主备，比如：redis主备，但是双缓存涉及到更新事务的问题，update可能读到脏数据，需要好好解决。

**缓存穿透**

是指用户查询数据，在数据库没有，自然在缓存中也不会有。这样就导致用户查询的时候，在缓存中找不到，每次都要去数据库再查询一遍，然后返回空（相当于进行了两次无用的查询）。这样请求就绕过缓存直接查数据库，这也是经常提的缓存命中率问题。

缓存穿透解决方案

（1）采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。

（2）如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。通过这个直接设置的默认值存放到缓存，这样第二次到缓存中获取就有值了，而不会继续访问数据库，这种办法最简单粗暴！

2，根据缓存数据Key的规则。例如我们公司是做机顶盒的，缓存数据以Mac为Key，Mac是有规则，如果不符合规则就过滤掉，这样可以过滤一部分查询。在做缓存规划的时候，Key有一定规则的话，可以采取这种办法。这种办法只能缓解一部分的压力，过滤和系统无关的查询，但是无法根治。

**缓存预热**

就是系统上线后，提前将相关的缓存数据直接加载到缓存系统。避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

 缓存预热解决方案：

（1）直接写个缓存刷新页面，上线时手工操作下；

（2）数据量不大，可以在项目启动的时候自动进行加载；

（3）定时刷新缓存或者由用户触发都行。

分布式缓存系统，如Memcached，Redis，比如缓存系统比较大，由十几台甚至几十台机器组成，这样预热会复杂一些。

1，写个程序去跑。

2，单个缓存预热框架。

缓存预热的目标就是在系统上线前，将数据加载到缓存中。

**缓存更新**

除了缓存服务器自带的缓存失效策略之外（Redis默认的有6中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

（1）定时去清理过期的缓存；

（2）当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的key是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

**缓存降级**

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

在进行降级之前要对系统进行梳理，看看系统是不是可以丢卒保帅；从而梳理出哪些必须誓死保护，哪些可降级；比如可以参考日志级别设置预案：

（1）一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；

（2）警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；

（3）错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；

（4）严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

* 使用缓存的合理性问题

<https://blog.csdn.net/diyhzp/article/details/54892358>

热点数据，缓存才有价值

对于冷数据而言，大部分数据可能还没有再次访问到就已经被挤出内存，不仅占用内存，而且价值不大。

对于热点数据，比如我们的某IM产品，生日祝福模块，当天的寿星列表，缓存以后可能读取数十万次。再举个例子，某导航产品，我们将导航信息，缓存以后可能读取数百万次。

频繁修改的数据，看情况考虑使用缓存

数据更新前至少读取两次，缓存才有意义。这个是最基本的策略，如果缓存还没有起作用就失效了，那就没有太大价值了。

对于上面两个例子，寿星列表、导航信息都存在一个特点，就是信息修改频率不高，读取通常非常高的场景。

那存不存在，修改频率很高，但是又不得不考虑缓存的场景呢？有！比如，这个读取接口对数据库的压力很大，但是又是热点数据，这个时候就需要考虑通过缓存手段，减少数据库的压力，比如我们的某助手产品的，点赞数，收藏数，分享数等是非常典型的热点数据，但是又不断变化，此时就需要将数据同步保存到Redis缓存，减少数据库压力。

数据不一致性

一般会对缓存设置失效时间，一旦超过失效时间，就要从数据库重新加载，因此应用要容忍一定时间的数据不一致。还有一种是在数据更新时立即更新缓存，不过这也会更多系统开销和事务一致性问题。

缓存更新机制

使用缓存过程中，我们经常会遇到缓存数据的不一致性和与脏读现象，我们有什么解决策略呢？

一般情况下，我们采取缓存双淘汰机制，在更新数据库的时候淘汰缓存。此外，设定超时时间，例如30分钟。极限场景下，即使有脏数据入cache，这个脏数据也最多存在三十分钟。

缓存可用性

缓存是提高数据读取性能的，缓存数据丢失和缓存不可用不会影响应用程序的处理。因此，一般的操作手段是，如果Redis出现异常，我们手动捕获这个异常，记录日志，并且去数据库查询数据返回给用户。

缓存服务降级

服务降级的目的，是为了防止Redis服务故障，导致数据库跟着一起发生雪崩问题。因此，对于不重要的缓存数据，可以采取服务降级策略，例如一个比较常见的做法就是，Redis出现问题，不去数据库查询，而是直接返回默认值给用户。

缓存预热

在新启动的缓存系统中，如果没有任何数据，在重建缓存数据过程中，系统的性能和数据库复制都不太好，那么最好的缓存系统启动时就把热点数据加载好，例如对于缓存信息，在启动缓存加载数据库中全部数据进行预热。一般情况下，我们会开通一个同步数据的接口，进行缓存预热。

缓存穿透

如果因为不恰当的业务，或者恶意攻击持续地发请求某些不存在的数据，由于缓存没有保存该数据，所有的请求都会落到数据库上，会对数据库造成很大压力，甚至奔溃。一个简单的对策是将不存在的数据也缓存起来。

* Redis常见的回收策略
* volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
* volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
* volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
* allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰
* allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
* no-enviction（驱逐）：禁止驱逐数据

## 消息队列

* 消息队列的使用场景
* MQ的不足是：
* 1）系统更复杂，多了一个MQ组件
* 2）消息传递路径更长，延时会增加
* 3）消息可靠性和重复性互为矛盾，消息不丢不重难以同时保证
* 4）上游无法知道下游的执行结果，这一点是很致命的

【典型场景一：数据驱动的任务依赖】

【典型场景二：上游不关心执行结果】

【典型场景三：上游关注执行结果，但执行时间很长】

<https://www.cnblogs.com/HigginCui/p/6478613.html>

异步处理，应用解耦，流量削锋和消息通讯四个场景。

2.1异步处理

2.2应用解耦

2.3流量削锋

2.4日志处理

2.5消息通讯

<https://www.cnblogs.com/joylee/p/8916460.html>

#### 通知

这里就用到了前文一个重要的特点，发布订阅，下游系统一直在监听MQ的数据，如果MQ有数据，下游系统则会按照 **先进先出** 这样的规则， **逐条进行消费** ，而上游系统只需要将数据存入MQ里，这样就既降低了不同系统之间的耦合度，同时也确保了消息通知的及时性，而且也不影响上游系统的性能。

#### 限流

上文有说了一个非常重要的特性，MQ **数据是只有一条数据在使用中。** 在很多存在并发，而又对数据一致性要求高，而且对性能要求也高的场景，如何保证，那么MQ就能起这个作用了。不管多少流量进来，MQ都会让你遵守规则，排除处理，不会因为其他原因，导致并发的问题，而出现很多意想不到脏数据。

#### 数据分发

MQ的发布订阅肯定不是只是简单的一对一，一个上游和一个下游的关系，MQ中间件基本都是支持一对多或者广播的模式，而且都可以根据规则选择分发的对象。这样上游的一份数据，众多下游系统中，可以根据规则选择是否接收这些数据，这样扩展性就很强了。  
PS:上文中的上游和下游，在MQ更多的是叫做生产者（producer）和消费者（consumer）。

#### 分布式事务

分布式事务是我们开发中一直尽量避免的一个技术点，但是，现在越来越多的系统是基于微服务架构开发，那么分布式事务成为必须要面对的难题，解决分布式事务有一个比较容易理解的方案，就是二次提交。基于MQ的特点，MQ作为二次提交的中间节点，负责存储请求数据，在失败的情况可以进行多次尝试，或者基于MQ中的队列数据进行回滚操作，是一个既能保证性能，又能保证业务一致性的方案，当然，这个方案的主要问题就是定制化较多，有一定的开发工作量。

<https://www.sojson.com/blog/48.html>

RabbitMQ 选型和对比

1.从社区活跃度

按照目前网络上的资料，RabbitMQ 、activeM 、ZeroMQ 三者中，综合来看，RabbitMQ 是首选。

2.持久化消息比较

ZeroMq 不支持，ActiveMq 和RabbitMq 都支持。持久化消息主要是指我们机器在不可抗力因素等情况下挂掉了，消息不会丢失的机制。

3.综合技术实现

可靠性、灵活的路由、集群、事务、高可用的队列、消息排序、问题追踪、可视化管理工具、插件系统等等。

RabbitMq / Kafka 最好，ActiveMq 次之，ZeroMq 最差。当然ZeroMq 也可以做到，不过自己必须手动写代码实现，代码量不小。尤其是可靠性中的：持久性、投递确认、发布者证实和高可用性。

4.高并发

毋庸置疑，RabbitMQ 最高，原因是它的实现语言是天生具备高并发高可用的erlang 语言。

5.比较关注的比较， RabbitMQ 和 Kafka

RabbitMq 比Kafka 成熟，在可用性上，稳定性上，可靠性上， RabbitMq 胜于 Kafka （理论上）。

另外，Kafka 的定位主要在日志等方面， 因为Kafka 设计的初衷就是处理日志的，可以看做是一个日志（消息）系统一个重要组件，针对性很强，所以 如果业务方面还是建议选择 RabbitMq 。

还有就是，Kafka 的性能（吞吐量、TPS ）比RabbitMq 要高出来很多。

* 消息的重发补偿解决思路
* 消息的幂等性解决思路
* 消息的堆积解决思路
* 自己如何实现消息队列
* 如何保证消息的有序性

# 开源框架和容器

## SSM/Servlet

### Servlet的生命周期

参考链接：

<https://blog.csdn.net/paicMis/article/details/52421706>

Servlet 通过调用 init () 方法进行初始化。

Servlet 调用 service() 方法来处理客户端的请求。

Servlet 通过调用 destroy() 方法终止（结束）。

最后，Servlet 是由 JVM 的垃圾回收器进行垃圾回收的。

### 转发与重定向的区别

参考链接：<https://blog.csdn.net/lijun102542/article/details/78518733>

转发在服务器端完成的；重定向是在客户端完成的

转发的速度快；重定向速度慢

转发的是同一次请求；重定向是两次不同请求

转发不会执行转发后的代码；重定向会执行重定向之后的代码

转发地址栏没有变化；重定向地址栏有变化

转发必须是在同一台服务器下完成；重定向可以在不同的服务器下完成

### BeanFactory 和 ApplicationContext 有什么区别

参考链接：<https://blog.csdn.net/lxyso/article/details/45446757>

<https://www.cnblogs.com/aspirant/p/9082858.html>

作用：

1. BeanFactory负责读取bean配置文档，管理bean的加载，实例化，维护bean之间的依赖关系，负责bean的声明周期。

2. ApplicationContext除了提供上述BeanFactory所能提供的功能之外，还提供了更完整的框架功能：

a. 国际化支持

b. 资源访问：Resource rs = ctx. getResource(“classpath:config.properties”), “file:c:/config.properties”

c. 事件传递：通过实现ApplicationContextAware接口

3. 常用的获取ApplicationContext的方法

1.利用MessageSource进行国际化

2.强大的事件机制(Event)

3.底层资源的访问

4.对Web应用的支持

5.其它区别   
1).BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化，这样，我们就不能发现一些存在的Spring的配置问题。而ApplicationContext则相反，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误。

2).BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册  
总结  
1.如果使用ApplicationContext，则配置的bean如果是singleton不管你用还是不用，都被实例化。好处是可以预先加载，坏处是浪费内存。  
2.BeanFactory，当使用BeanFactory实例化对象时，配置的bean不会马上被实例化。当你使用该bean时才会被实例化（getBean）。好处是节约内存，缺点是速度比较慢。多用于移动设备的开发上。  
3.一般没有特殊要求，应当使用ApplicationContext完成。

### Spring Bean 的生命周期

参考链接：<https://www.cnblogs.com/kenshinobiy/p/4652008.html>

实现：/study-spring/src/main/java/com/bage/lifecircle/LifeCircleBean.java

1. 实例化一个Bean，也就是我们通常说的new

2. 按照Spring上下文对实例化的Bean进行配置，也就是IOC注入

3. 如果这个Bean实现了BeanNameAware接口，会调用它实现的setBeanName(String beanId)方法，此处传递的是Spring配置文件中Bean的ID

4. 如果这个Bean实现了BeanFactoryAware接口，会调用它实现的setBeanFactory()，传递的是Spring工厂本身（可以用这个方法获取到其他Bean）

5. 如果这个Bean实现了ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文，该方式同样可以实现步骤4，但比4更好，以为ApplicationContext是BeanFactory的子接口，有更多的实现方法

6. 如果这个Bean关联了BeanPostProcessor接口，将会调用postProcessBeforeInitialization(Object obj, String s)方法，BeanPostProcessor经常被用作是Bean内容的更改，并且由于这个是在Bean初始化结束时调用After方法，也可用于内存或缓存技术

7. 如果这个Bean在Spring配置文件中配置了init-method属性会自动调用其配置的初始化方法

8. 如果这个Bean关联了BeanPostProcessor接口，将会调用postAfterInitialization(Object obj, String s)方法

注意：以上工作完成以后就可以用这个Bean了，那这个Bean是一个single的，所以一般情况下我们调用同一个ID的Bean会是在内容地址相同的实例

9. 当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean接口，会调用其实现的destroy方法

10. 最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法

### Spring中Bean的作用域，默认的是哪一个

singleton: 在Spring的IoC容器中只存在一个对象实例，所有该对象的引用都共享这个实例。Spring 容器只会创建该bean定义的唯一实例，这个实例会被保存到缓存中，并且对该bean的所有后续请求和引用都将返回该缓存中的对象实例，一般情况下，无状态的bean使用该scope。

prototype:每次对该bean的请求都会创建一个新的实例，一般情况下，有状态的bean使用该scope。

request：每次http请求将会有各自的bean实例，类似于prototype。

session：在一个http session中，一个bean定义对应一个bean实例。

global session：在一个全局的http session中，一个bean定义对应一个bean实例。典型情况下，仅在使用portlet context的时候有效。

### 说说 Spring AOP、Spring AOP 实现原理

<https://www.cnblogs.com/lcngu/p/5339555.html>

前面在学习代理模式的时候，了解到代理模式分为动态代理和静态代理。现在我们就以代理模式为基础先实现我们自己的AOP框架，再来研究Spring的AOP的实现原理。

　　先以静态代理实现，静态代理关键是在代理对象和目标对象实现共同的接口，并且代理对象持有目标对象的引用。

　　公共接口代码：

### 动态代理（CGLib 与 JDK）、优缺点、性能对比、如何选择

<https://blog.csdn.net/fighterandknight/article/details/51200470>

### Spring 事务实现方式、事务的传播机制、默认的事务类别

### Spring 事务底层原理

### Spring事务失效（事务嵌套），JDK动态代理给Spring事务埋下的坑，可参考《[JDK动态代理给Spring事务埋下的坑！](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247484940&idx=1&sn=0a0a7198e96f57d610d3421b19573002&chksm=e9c5ffbddeb276ab64ff3b3efde003193902c69acda797fdc04124f6c2a786255d58817b5a5c&scene=21#wechat_redirect)》

### 如何自定义注解实现功能

### Spring MVC 运行流程

<https://www.cnblogs.com/glorywzm/p/6503141.html>

1.用户发送请求到DispatchServlet

2.DispatchServlet根据请求路径查询具体的Handler

3.HandlerMapping返回一个HandlerExcutionChain给DispatchServlet

　HandlerExcutionChain：Handler和Interceptor集合

4.DispatchServlet调用HandlerAdapter适配器

5.HandlerAdapter调用具体的Handler处理业务

6.Handler处理结束返回一个具体的ModelAndView给适配器

   ModelAndView:model-->数据模型，view-->视图名称

7.适配器将ModelAndView给DispatchServlet

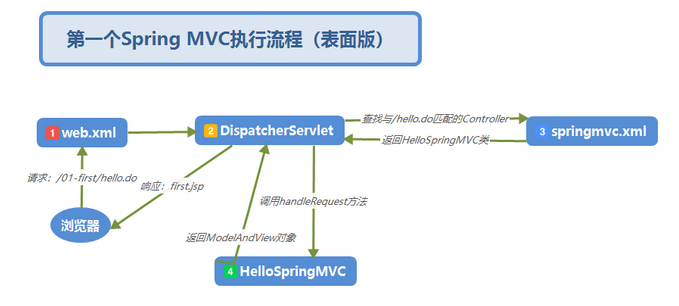
8.DispatchServlet把视图名称给ViewResolver视图解析器

9.ViewResolver返回一个具体的视图给DispatchServlet

10.渲染视图

11.展示给用户





### Spring MVC 启动流程

<https://www.cnblogs.com/mingziday/p/4987058.html>

下面一个基本的运用springMVC的的web.xml的配置，这里要注意两个地方，一个是ContextLoadListener，一个是DispatcherServlet。web容器正是通过这两个配置才和Spring关联起来。这两个配置与web容器的ServletContext关联，为Spring的Ioc容器提供了一个宿主，在建立起Ioc容器体系之后，把DispatcherServlet作为Spring MVC处理web请求的转发器建立起来，从而完成响应Http请求的准备。

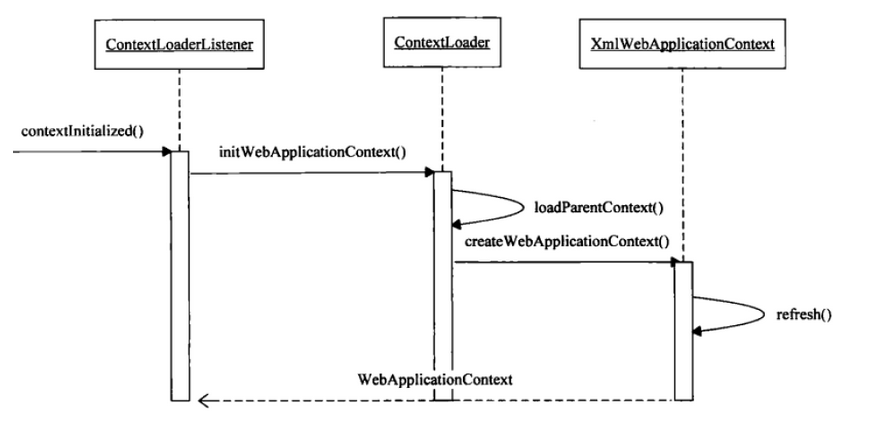
Spring MVC启动过程依据这两个配置大致分为两个过程：

1、ContextLoaderListener初始化，实例化IoC容器，并将此容器实例注册到ServletContext中。ContextLoaderListener实现ServletContextListener，这个接口里面的函数会结合web容器的生命周期被调用。因为ServletContextListener是ServletContext的监听者，如果ServletContext发生变化，会触发相应的事件，而监听者一直对这些事件进行监听，如果接受到了监听的事件，就会作出预先设计好的动作。例如在服务器启动，ServletContext被创建的时候，ServletContextListener的contextInitialized()方法被调用，从而拉开了初始化Spring IOC容器的大幕。

首先从Servlet的启动事件中得到ServletContext，然后读取web.xml中的各个相关的属性值，接着ContextLoader会实例化WebApplicationContext，并完成载入和初始化的过程，这个被初始化的第一个上下文作为根上下文而存在，这个根上下文载入后，被绑定到web应用程序的ServletContext上，这样，IOC容器中的类就可以在任何地方访问到了。

具体的Ioc容器的载入过程在refresh()中实现，这个方法主要干了以下几件事情：

（1）Resource文件的定位，即找到bean的配置文件  
（2）通过特定的reader解析该bean配置文件，抽象成beanDefinition类  
（3）将beanDefinition向容器注册，写入到一个大的HashMap中，供实例化请求的时候使用



2、DispatcherServlet初始化，建立自己的上下文，也注册到ServletContext中。

DispatchServlet本质上是一个Servlet，web容器启动的时候，servlet也会初始化，其init方法被调用，开启初始化之旅。

DispatchServlet会建立自己的上下文来持有Spring MVC特殊的bean对象，在建立这个自己持有的Ioc容器的时候，会从ServletContext中得到根上下文作为DispatchServlet上下文的parent上下文。有了这个根上下文，再对自己持有的上下文进行初始化，最后把自己持有的这个上下文保存到ServletContext中，供以后检索和使用。

在initWebApplicationContext中完成了对自己上下文的初始化，这里面也有一个refresh的过程，和普通的Ioc容器初始化大同小异。

另外一些MVC的特性初始化时在initStrategies()中实现的，包括支持国际化的LocalResolver、支持Request映射的HandlerMappings，以及视图生成的ViewResolver等等。

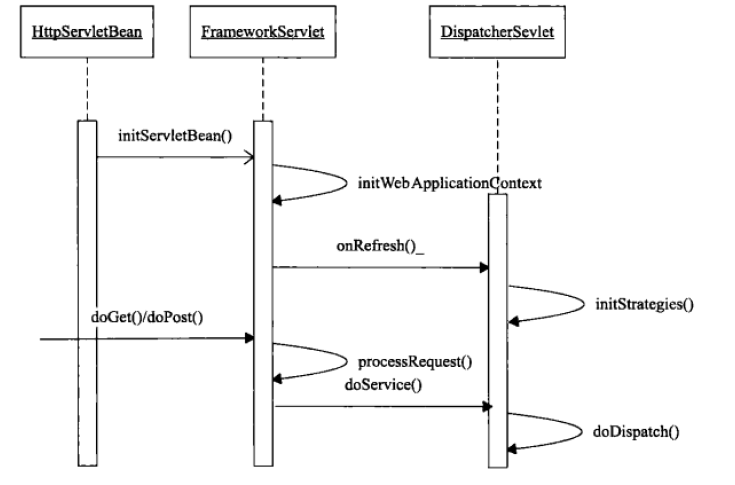
初始化完成后，上下文环境中已经定义的所有HandlerMapping都已经被加载了，这些被加载的HandlerMapping放在List中，存储着Http请求和Controller的映射数据。

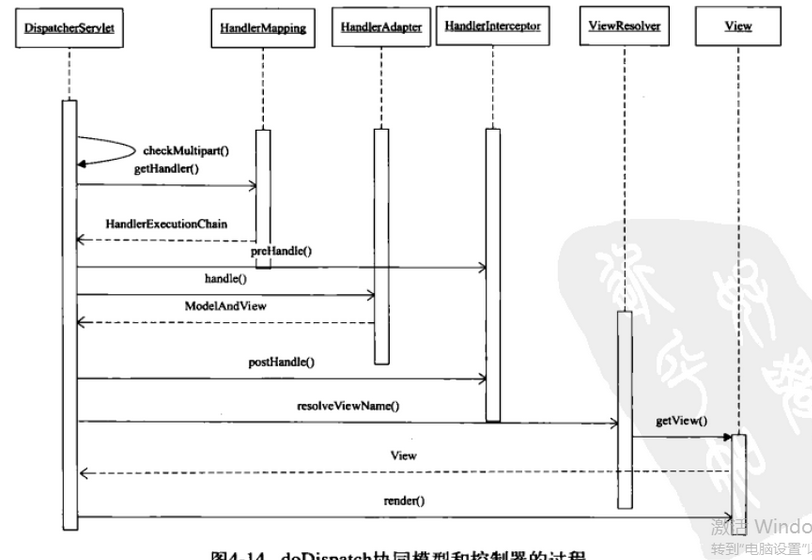
DispatcherServlet是HttpServlet的子类，也是通过doService()来响应HTTP请求的，doService()中采用doDispatch()方法实现的。

（1）首先通过url匹配，通过getHandler找到Handler，Hander中封装了配置的Controller

（2）执行Handler，得到返回的ModelAndView结果

（3）最后把这个ModelAndView对象交给视图对象的render()方法去呈现。





### Spring 的单例实现原理

### Spring 框架中用到了哪些设计模式

### Spring 其他产品（Srping Boot、Spring Cloud、Spring Secuirity、Spring Data、Spring AMQP 等）

### 有没有用到Spring Boot，Spring Boot的认识、原理

### MyBatis的原理

* 可参考《[为什么会有Spring](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247484822&idx=1&sn=6fbee2a12b31b6102a18d3725671d41b&chksm=e9c5fc27deb275319641c3f30d168b85c7c196fd276d47efa35046b5dc54f5b77174c5bf8808&scene=21#wechat_redirect)》
* 可参考《[为什么会有Spring AOP](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247484827&idx=1&sn=b9d82f3fced6a875f8dfc22e5849b28e&chksm=e9c5fc2adeb2753c516ef8fc959c0c9dd84ccacaa40473b64bc58b5137c30562a0b45803ba8e&scene=21#wechat_redirect)》

## Netty

### 为什么选择 Netty

### 说说业务中，Netty 的使用场景

### 原生的 NIO 在 JDK 1.7 版本存在 epoll bug

### 什么是TCP 粘包/拆包

### TCP粘包/拆包的解决办法

### Netty 线程模型

### 说说 Netty 的零拷贝

### Netty 内部执行流程

### Netty 重连实现

## Tomcat

* Tomcat的基础架构（Server、Service、Connector、Container）
* Tomcat如何加载Servlet的
* Pipeline-Valve机制
* 可参考：《[四张图带你了解Tomcat系统架构](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247484905&idx=1&sn=6c8acd89476fadbc4cb9ccfda9c9c2e4&chksm=e9c5fc58deb2754e7519511bb0ed8dcbfa3fe29179663b53f3626643f8b9c82068d9b0464ee6&scene=21#wechat_redirect)！》

# 分布式

## Nginx

* 请解释什么是C10K问题或者知道什么是C10K问题吗？
* Nginx简介，可参考《[Nginx简介](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247483994&idx=1&sn=b6591f62c7ea6b4adc5a5bf1bf4eac40&chksm=e9c5fbebdeb272fdd865a9c61a380f6b909fc988f99d00ce0aa8c3efca501644db46c40bd4f2&scene=21#wechat_redirect)》
* 正向代理和反向代理.
* Nginx几种常见的负载均衡策略
* Nginx服务器上的Master和Worker进程分别是什么
* 使用“反向代理服务器”的优点是什么?

## 分布式其他

* 谈谈业务中使用分布式的场景
* Session 分布式方案

<https://www.cnblogs.com/qin-derella/p/6808436.html>

粘性Session，Session复制或Session共享

* Session 分布式处理
* 分布式锁的应用场景、分布式锁的产生原因、基本概念
* 分布是锁的常见解决方案

基于数据库实现分布式锁

基于缓存（redis，memcached，tair）实现分布式锁

基于Zookeeper实现分布式锁

* 分布式事务的常见解决方案
* 集群与负载均衡的算法与实现
* 说说分库与分表设计，可参考《[数据库分库分表策略的具体实现方案](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247483931&idx=1&sn=6eda41aa81c1243422a603205d2fad22&chksm=e9c5fbaadeb272bc92537803c14a6f55e1170b1a3b8f60160f66417800c0ace960dfe192717a&scene=21#wechat_redirect)》
* 分库与分表带来的分布式困境与应对之策

**4.3、Dubbo**

* 什么是Dubbo，可参考《[Dubbo入门](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247483791&idx=1&sn=49345f1a022734e81e9257f2b8d38a52&chksm=e9c5f83edeb2712805a77c1e1589e8f1d04bd17e55eeb2a45cabddb46d03615636908f058628&scene=21#wechat_redirect)》
* 什么是RPC、如何实现RPC、RPC 的实现原理，可参考《[基于HTTP的RPC实现](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247483900&idx=1&sn=c5ca198a66a701f81c2ab118fe7a734a&chksm=e9c5f84ddeb2715bc574e467cd6537ef81f223453e0989ffd136976b48dcc2d961a75be596de&scene=21#wechat_redirect)》
* Dubbo中的SPI是什么概念
* Dubbo的基本原理、执行流程

**五、微服务**

**5.1、微服务**

* 前后端分离是如何做的？
* 微服务哪些框架
* Spring Could的常见组件有哪些？可参考《[Spring Cloud概述](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247484125&idx=1&sn=ddba9fba6ae900f5ef71a68f70afebe5&chksm=e9c5fb6cdeb2727a1e8d16a05adb7d9df0170e392ae579d1e40075fb488d680f0c061aa45327&scene=21#wechat_redirect)》
* 领域驱动有了解吗？什么是领域驱动模型？充血模型、贫血模型
* JWT有了解吗，什么是JWT，可参考《[前后端分离利器之JWT](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247485183&idx=1&sn=05dac824dbb534710dd99d6c895fbaf5&chksm=e9c5ff4edeb27658173c8b06ad6d1241d3b7822c734ddf6ac064d40e63cb0cb0a0c90804b9c7&scene=21#wechat_redirect)》
* 你怎么理解 RESTful
* 说说如何设计一个良好的 API
* 如何理解 RESTful API 的幂等性

HTTP幂等方法，是指无论调用多少次都不会有不同结果的 HTTP 方法。不管你调用一次，还是调用一百次，一千次，结果都是相同的。

* 如何保证接口的幂等性
* 说说 CAP 定理、BASE 理论
* 怎么考虑数据一致性问题
* 说说最终一致性的实现方案
* 微服务的优缺点，可参考《[微服务批判](http://mp.weixin.qq.com/s?__biz=MzI1NDQ3MjQxNA==&mid=2247485005&idx=1&sn=78a1d286c6a15a81ea5dcf6634a70b54&chksm=e9c5fffcdeb276ea3c766a6e5954685db0e89bea8ff0f47c9a3ff2c0c02991f791a5160287c4&scene=21#wechat_redirect)》
* 微服务与 SOA 的区别
* 如何拆分服务、水平分割、垂直分割
* 如何应对微服务的链式调用异常
* 如何快速追踪与定位问题
* 如何保证微服务的安全、认证

# 安全问题

* 如何防范常见的Web攻击、如何方式SQL注入
* 服务端通信安全攻防
* HTTPS原理剖析、降级攻击、HTTP与HTTPS的对比

# 性能优化

* 性能指标有哪些

<https://baike.baidu.com/item/%E8%BD%AF%E4%BB%B6%E6%80%A7%E8%83%BD/5341633>

**响应时间、系统响应时间和应用延迟时间、吞吐量（**单位时间内处理请求的数量**）、并发用户数、资源利用率（**一段时间内资源平均被占用的情况**）**

* 如何发现性能瓶颈
* 性能调优的常见手段
* 说说你在项目中如何进行性能调优

# Linux

<https://www.cnblogs.com/hystj/p/8552757.html>

<http://www.iplaypy.com/linux/l6824.html>

# 其他

**6.1、设计能力**

* 说说你在项目中使用过的UML图
* 你如何考虑组件化、服务化、系统拆分
* 秒杀场景如何设计

<https://www.zhihu.com/question/54895548>

作者：网易云  
链接：https://www.zhihu.com/question/54895548/answer/259218876  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

答一下网易云架构师总结的秒杀系统设计与实现：

秒杀的核心问题就是**极高并发处理**，由于系统要在瞬时承受平时数十倍甚至上百倍的流量，这往往超出系统上限，因此处理秒杀的**核心思路是流控和性能优化**。

***流控***

* **请求流控**

尽可能在上游拦截和限制请求，限制流入后端的量，保证后端系统正常。

因为无论多少人参与秒杀，实际成交往往是有限的，而且远小于参加秒杀的人数，因此可以通过前端系统进行拦截，限制最终流入系统的请求数量，来保证系统正常进行。

* **客户端流控**

在客户端进行访问限制，较为合适的做法是屏蔽用户高频请求，比如在网页中设置5s一次访问限制，可以防止用户过度刷接口。这种做法较为简单，用户体验也尚可，可以拦截大部分小白用户的异常访问，比如狂刷F5。关键是要明确告知用户，如果像一些抢购系统那样假装提交一个排队页面但又不回应任何请求，就是赤裸裸的欺骗了。

* **Web端流控**

对客户端，特别是页面端的限流，对稍有编程知识或者网络基础的用户而言没有作用（可以简单修改JS或者模拟请求），因此服务端流控是必要的。服务端限流的配置方法有很多种，现在的主流Web服务器一般都支持配置访问限制，可以通过配置实现简单的流控。

但是这种限制一般都在协议层。如果要实现更为精细的访问限制（根据业务逻辑限流），可以在后端服务器上，对不同业务实现访问限制。常见做法是可以通过在内存或缓存服务中加入请求访问信息，来实现访问量限制。

* **后端系统流控**

上述的流控做法只能限制用户异常访问，如果正常访问的用户数量很多，就有后端系统压力过大甚至异常宕机的可能，因此需要后端系统流量控制。

对于后端系统的访问限制可以通过异步处理、消息队列、并发限制等方式实现。核心思路是保证后端系统的压力维持在可以正常处理的水平。对于超过系统负载的请求，可以选择直接拒绝，以此来对系统进行保护，保证在极限压力的情况下，系统有合理范围内的处理能力。

***系统架构优化***

除了流控之外，提高系统的处理能力也是非常重要的，通过系统设计和架构优化，可以提高系统的吞吐量和抗压能力。关于通用系统性能的提升，已经超出本节的范围，这里只会提几点和秒杀相关的优化。

* **读取加速**：在秒杀活动中，数据需求一般都是读多写少。20万人抢2000个商品，最后提交的订单最多也就2000个，但是在秒杀过程中，这20万人会一直产生大量的读取请求。因此可以使用缓存服务对用户请求进行缓存优化，把一些高频访问的内容放到缓存中去。对于更大规模的系统，可以通过静态文件分离、CDN服务等把用户请求分散到外围设施中去，以此来分担系统压力。
* **异步处理和排队**：通过消息队列和异步调用的方式可以实现接口异步处理，快速响应用户请求，在后端有较为充足的时间来处理实际的用户操作，提高对用户请求的响应速度，从而提升用户体验。通过消息队列还可以隔离前端的压力，实现排队系统，在涌入大量压力的情况下保证系统可以按照正常速率来处理请求，不会被流量压垮。
* **无状态服务设计**：相对于有状态服务，无状态服务更容易进行扩展，实现无状态化的服务可以在秒杀活动前进行快速扩容。而云化的服务更是有着先天的扩容优势，一般都可以实现分钟级别的资源扩容。

***系统扩容***

这项内容是在云计算环境下才成为可能，相对于传统的IT行业，云计算提供了快速的系统交付能力（min VS. day），因此可以做到按需分配，在业务需要时实现资源的并行扩展。

对一次成功的秒杀活动来说，无论如何限流，如何优化系统，最终产生数倍于正常请求的压力是很正常的。因此临时性的系统扩容必不可少，系统扩容包括以下3个方面。

* **增加系统规格**：可以预先增加系统容量，比如提高系统带宽、购买更多流量等。
* **服务扩展**：无状态服务+负载均衡可以直接进行水平扩展，有状态的服务则需要进行较为复杂的垂直扩展，增大实例规格。
* **后端系统扩容**：缓存服务和数据库服务都可以进行容量扩展。

**秒杀服务实践**

一般来说，流控的实现，特别是业务层流控，依赖于业务自身的设计，因此云计算提供的服务在于更多、更完善的基础设计，来支持用户进行更简单的架构优化和扩容能力。

***系统架构优化***

通过CDN服务和对象存储服务来分离静态资源，实现静态资源的加速，避免服务器被大量静态资源请求过度占用。要实现异步的消息处理，可以使用队列服务来传输消息，以达到消息异步化和流控。

***系统扩容***

云服务会提供按需计费的资源分配方式和分钟级甚至秒级的资源交付能力，根据需要快速进行资源定制和交付。

内部系统可以通过负载均衡等服务实现并行扩展，在网易云基础服务中，用户可以直接使用Kubernetes的Replication Controller服务实现在线水平扩容。对于对外提供的Web系统，可以通过负载均衡服务实现水平在线扩展。

对于后端系统来说，建议使用云计算服务商提供的基础服务来实现并行扩展。例如，[网易云基础服务](https://link.zhihu.com/?target=https%3A//www.163yun.com/product-cloudcompute%3Fid%3Dzhihu%26tag%3DM_zhihu_54895548)就提供了[分布式缓存服务](https://link.zhihu.com/?target=https%3A//www.163yun.com/product/redis%3Fid%3Dzhihu%26tag%3DM_zhihu_54895548)和[数据库服务](https://link.zhihu.com/?target=https%3A//www.163yun.com/product/rds%3Fid%3Dzhihu%26tag%3DM_zhihu_54895548)，支持在线扩容。

**6.2、业务工程**

* 说说你的开发流程、如何进行自动化部署的
* 你和团队是如何沟通的
* 你如何进行代码评审
* 说说你对技术与业务的理解
* 说说你在项目中遇到感觉最难Bug，是如何解决的
* 介绍一下工作中的一个你认为最有价值的项目，以及在这个过程中的角色、解决的问题、你觉得你们项目还有哪些不足的地方

**6.3、软实力**

* 说说你的优缺点、亮点
* 说说你最近在看什么书、什么博客、在研究什么新技术、再看那些开源项目的源代码
* 说说你觉得最有意义的技术书籍
* 工作之余做什么事情、平时是如何学习的，怎样提升自己的能力
* 说说个人发展方向方面的思考
* 说说你认为的服务端开发工程师应该具备哪些能力
* 说说你认为的架构师是什么样的，架构师主要做什么
* 如何看待加班的问题

当然，一个完整的面试肯定不止上述的知识点，其他的诸如：Linux、数据结构、算法、逻辑思维题、系统设计题、职业规划等等都会有所涉及！总之，充分的准备是应对一切面试的不二法宝！