[**https://blog.csdn.net/u011279240/article/details/80532555**](https://blog.csdn.net/u011279240/article/details/80532555)

**几个大厂的面试题目目录：**

java基础（40题）

多线程（51题）

设计模式（8点）

JVM（12题）

数据结构与算法（17题）

数据库（22题）

Spring （13题）

Netty（7大题）

缓存（9题）

技术框架（8题）

技术深度（12题）

分布式（33题）

系统架构（18题）

linux（9大题）

TCP/IP（19点）

软能力（12点)

自我问答（44点）

目录

[**java 基础** 3](#_Toc522023573)

[**多线程** 12](#_Toc522023574)

[**设计模式** 24](#_Toc522023575)

[**JVM** 31](#_Toc522023576)

[**数据结构与算法** 36](#_Toc522023577)

[**数据库** 44](#_Toc522023578)

[**Spring** 53](#_Toc522023579)

[**Netty** 58](#_Toc522023580)

[**缓存** 64](#_Toc522023581)

[**技术框架** 68](#_Toc522023582)

[**技术深度** 72](#_Toc522023583)

[**分布式** 78](#_Toc522023584)

[**系统架构** 91](#_Toc522023585)

[**LINUX** 95](#_Toc522023586)

[**TCP/IP** 97](#_Toc522023587)

[**软能力** 108](#_Toc522023588)

[**自我问答总结** 109](#_Toc522023589)

**java 基础**

1. 八种基本数据类型的大小，以及他们的封装类

double ---Double 8位 0.0d

float ---Float 4位 0.0f

long ---Long 8位 0L

int ---Integer 4位 0

short ---Short 2位 (short)0

byte ---byte 1位 (byte)0

char ---Character 2位 null\u0000

boolean ---Boolean -- false

1. 引用数据类型

数组，类，接口

1. Switch能否用string做参数

以前只能支持byte、short、char、int，可以强转

Jdk7.0以后可以，整型、枚举类型、boolean、字符串都可以

1. **equals与==的区别\***

==比较的是2个对象的地址，而equals比较的是2个对象的内容

**== 比较的是变量(栈)内存中存放的对象的(堆)内存地址，用来判断两个对象的地址是否相同，即是否是指相同一个对象。比较的是真正意义上的指针操作。**

**Equals用来比较的是两个对象的内容是否相等，由于所有的类都是继承自java.lang.Object类的，所以适用于所有对象，如果没有对该方法进行覆盖的话，调用的仍然是Object类中的方法，而Object中的equals方法返回的却是==的判断。**

1. 自动装箱，常量池

自动装箱：**基本数据类型🡪对象类型**  
自动拆箱：**对象类型🡪基本数据类型**

常量池：Byte,Short,Integer,Long,Character在自动装箱时对于值从**–128到127**之间的值（**共享**），会存在内存中被重用

字符串常量池

常量池在java用于保存在编译期已确定的，已编译的class文件中的一份数据。它包括了关于类，方法，接口等中的常量，也包括[字符串](https://baike.baidu.com/item/%E5%AD%97%E7%AC%A6%E4%B8%B2)常量，如String s = “java”这种申明方式；当然也可扩充，[执行器](https://baike.baidu.com/item/%E6%89%A7%E8%A1%8C%E5%99%A8/2404958)产生的常量也会放入常量池，故认为常量池是[JVM](https://baike.baidu.com/item/JVM)的一块特殊的内存空间。

1. Object有哪些公用方法

clone(),hashCode(),equals(),notify(),notifyAll(),wait(),getClass(),toString,finalize()

1. Java的四种引用，强弱软虚，用到的场景

**强引用：**使用普遍的引用，内存空间不足了，一般垃圾回收器绝不会回收它

**软引用：**软引用可用来实现内存敏感的高速缓存，内存空间不足了，就会回收这些对象的内存。

**弱引用：**具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存

**虚引用：**虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。

1. **Hashcode的作用\***

利用哈希算法，配合基于散列的集合一起正常运行，Java中的hashCode方法就是根据一定的规则将与对象相关的信息（比如对象的存储地址，对象的字段等）映射成一个数值，这个数值称作为散列值，**降低equals的调用**，实现存放的值不会重复。

Note:重写equals必须重写hashcode方法，equals相等，hashcode也必须相等。

**一般对于存放到Set集合或者Map中键值对的元素，需要按需要重写hashCode与equals方法，以保证唯一性！**

**例如hashset存放多个对象，重写equals和hashcode**

两个对象相等，其HashCode一定相同;

两个对象不相等，其HashCode有可能相同;

HashCode相同的两个对象，不一定相等;

HashCode不相同的两个对象，一定不相等;

1. **HashMap的hashcode的作用\***

<https://blog.csdn.net/baidu_31657889/article/details/52298367>

Java中的集合（Collection）有两类，一类是List，再有一类是Set。前者集合内的元素是有序的，元素可以重复；后者元素无序，但元素不可重复。 equals方法可用于保证元素不重复，但是，如果每增加一个元素就检查一次，如果集合中现在已经有1000个元素，那么第1001个元素加入集合时，就要调用1000次equals方法。这显然会大大降低效率。

HashMap的数据结构是 数组+链表形式存储数据，继承AbstractMap，实现Map接口，主要用于查找的快捷性。

1. **为什么重载hashCode方法？\***

一般的地方不需要重载hashCode，只有当类需要放在HashTable、HashMap、HashSet等等hash结构的集合时才会重载hashCode，那么为什么要重载hashCode呢？就HashMap来说，好比HashMap就是一个大内存块，里面有很多小内存块，小内存块里面是一系列的对象，可以利用hashCode来查找小内存块hashCode%size(小内存块数量)，所以当equal相等时，hashCode必须相等，而且如果是object对象，必须重载hashCode和equal方法。

1. **ArrayList、LinkedList、Vector的区别\***

ArrayList： 线程不安全，数组，适合查找，可自动扩容50%

三个构造器，无参，容量，Collection接口，**transient** Object[] **elementData**;不被序列化。

LinkedList：线程不安全，链表，审核插入，删除

Vector： 线程安全，数组，适合查找，可自动扩容100%

1. **String、StringBuffer与StringBuilder的区别\***

String 是final修饰的，字符串常量，**String对象一旦创建之后该对象是不可更改的**

StringBuffer 字符串变量，**对象可变**，线程安全，适合多线程下字符缓冲区大量操作

StringBuider 字符串变量，**对象可变**，线程不安全，适用单线程下载字符缓冲区进行大量操作的情况，都是继承AbstractStringBuilder super.容量为16

1. Map、Set、List、Queue、Stack的特点与用法。

**Map** map集合，k-v键值对存储

HashTable 和 HashMap 是 Map 的实现类     
**HashTable** 是线程安全的，不能存储 null 值     
**HashMap** 不是线程安全的，可以存储 null 值

**TreeMap**可以保证顺序，HashMap不保证顺序，即为无序的。

**Set** 集合，无序，不重复

**List** 数组集合，ArrayList ， Vector ， LinkedList 是 List 的实现类

**ArrayList** 是线程不安全的， Vector 是线程安全的，这两个类底层都是由数组实现的

**LinkedList** 是线程不安全的，底层是由链表实现的

**Queue** 队列，提供了几个基本方法，offer、poll、peek等。已知实现类有LinkedList、PriorityQueue等。

**Stack** 栈，继承自Vector，实现一个后进先出的栈。提供了几个基本方法，push、pop、peak、empty、search等。

HashMap和HashTable的区别

1. **JDK7与JDK8中HashMap的实现\***

JDK8在JDK7的基础上引入了**红黑树**-b，因为链表过长，会导致效率很低，将原来链表数组的每一个链表分成**奇偶两个子链表**分别挂在新链表数组的散列位置，这样就减少了每个链表的长度，增加查找效率

1. **HashMap和ConcurrentHashMap的区别，HashMap的底层源码\***

HashMap是线程不安全的，ConcurrentHashMap是线程安全的，适用于高并发，ConcurrentHashMap就是一个分段的hashtable，根据自定的hashcode算法生成的对象来获取对应hashcode的分段块进行加锁，不用整体加锁，提高了效率。

<https://blog.csdn.net/qpzkobe/article/details/78948197>

**HashMap的get（key）方法**是获取key的hash值，计算hash&（n-1）得到在链表数组中的位置first=tab[hash&(n-1)],先判断first的key是否与参数key相等，不等就遍历后面的链表找到相同的key值返回对应的Value值即可。

**HashMap的put（key）方法是**判断键值对数组tab[]是否为空或位null，否则以默认大小resize()；根据键值key计算hash值得到插入的数组索引i，如果tab[i]==null,直接新建节点添加，否则判断当前数组中处理hash冲突的方式为链表还是红黑树(check第一个节点类型即可)，分别处理。

**构造hash表时，**如果不指明初始大小，默认大小为**16**（即Node数组大小16），如果Node[]数组中的元素达到（填充比\*Node.length）重新调整HashMap大小 变为原来2倍大小,扩容很耗时

1. ConcurrentHashMap能完全替代HashTable吗？

Hash table虽然性能上不如ConcurrentHashMap，但并不能完全被取代，两者的迭代器的一致性不同的，**ConcurrentHashMap由于分段锁**，弱一致性主要是为了提升效率。

强一致性就如hashtable一样，锁整个map。

1. **为什么HashMap是线程不安全的\***

在某一时刻同时操作HashMap并执行put操作，而有大于两个key的hash值相同，如图中a1、a2，这个时候需要解决碰撞冲突，而解决冲突的办法上面已经说过，对于链表的结构在这里不再赘述，暂且不讨论是从链表头部插入还是从尾部初入，这个时候**两个线程如果恰好都取到了对应位置的头结点e1**，而最终的结果可想而知，a1、a2两个数据中势必会有一个会丢失

当多个线程同时检测到总数量超过门限值的时候就会同时调用resize操作，各自生成新的数组并rehash后赋给该map底层的数组table，结果最终只有最后一个线程生成的新数组被赋给table变量，其他线程的均会丢失。而且当某些线程已经完成赋值而其他线程刚开始的时候，就会用已经被赋值的table作为原始数组，这样也会有问题。

1. **多并发情况下HashMap是否还会产生死循环\***

<https://blog.csdn.net/u010412719/article/details/52049347>

不会，jdk1.8版本以后已经没有这个问题了，没有transfer这个函数了do while可能造成的死循环，对原有造成死锁的关键原因点（新table复制在头端添加元素）改进为依次在末端添加新的元素

1. **TreeMap、HashMap、LindedHashMap的区别\***

**LinkedHashMap**可以保证HashMap集合有序。存入的顺序和取出的顺序一致。

**TreeMap**实现SortMap接口，能够把它保存的记录根据键排序,默认是按键值的升序排序，也可以指定排序的比较器，当用Iterator 遍历TreeMap时，得到的记录是排过序的。

**HashMap**不保证顺序，即为无序的，具有很快的访问速度。HashMap最多只允许一条记录的键为Null;允许多条记录的值为 Null;HashMap不支持线程的同步。

1. Collection包结构，与Collections的区别

Collection是个java.util下的接口，它是各种集合结构的父接口。

Collections是个java.util下的类，它包含有各种有关集合操作的静态方法。

1. try?catch?finally，try里有return，finally还执行么

如果finally没有return 相同返回值变量，则返回try里面的return，否则finally 的return 值会影响 try里面return结果。

finally还是会执行的，除非中途遇到jvm退出。

1. Excption与Error包结构，OOM你遇到过哪些情况，SOF你遇到过哪些情况

都是Throwable的子类，Exception指出了合理的应用程序想要捕获的条件。Error 它用于指示合理的应用程序不应该试图捕获的严重问题，大多数这样的错误都是异常条件

Java Heap 溢出，虚拟机栈和本地方法栈溢出，运行时常量池溢出，方法区溢出

StackOverflowError 的定义：当应用程序递归太深而发生堆栈溢出时，抛出该错误。

1. Java(OOP)面向对象的三个特征与含义

**封装：**可见性封装，setget读写，将类的某些特征隐藏在类的内部，不允许外部程序直接访问，而是通过该类提供的方法来实现对隐藏信息的操作和访问。

**继承：**子类继承父类，可以得到父类的全部属性和方法（除了父类中的构造方法），java中的多继承可以通过接口来实现。

**多态：**一种是编译时多态，另外一种是运行时多态，编译时多态是通过方法的重载来实现的，运行时多态是通过方法的重写来实现的。

1. Override和Overload的含义去区别

**重写**，是子类覆盖父类方法，重新定义，但是，返回类型，参数，参数类型，抛出异常都和父类一致，被覆盖的方法不能private，子类函数访问权限要大于等于父类的，

子类无法覆盖父类的static方法或private方法。

**重载**，是一个类中，方法名同名，但是具有不同程度参数类型，不同的参数个数，不同的参数顺序。

1. Interface与abstract类的区别

1.抽象类可以有**构造方法**，接口中不能有构造方法。

2.抽象类中可以有**普通成员变量**，接口中没有普通成员变量

3.抽象类中可以包含**非抽象的普通方法**，接口中的所有方法必须都是抽象的，不能有非抽象的普通方法。

4. 抽象类中的抽象方法的访问类型可以是public，protected和（默认类型,虽然eclipse下不报错，但应该也不行），但接口中的抽象方法只能是public类型的，并且默认即为public abstract类型。

5. 抽象类中可以包含**静态方法**，接口中不能包含静态方法

6. 抽象类和接口中都可以包含**静态成员变量**，抽象类中的静态成员变量的访问类型可以任意，但接口中定义的变量只能是public static final类型，并且默认即为public static final类型。

7. 一个类可以实现多个接口，但只能继承一个抽象类。

1. Static?class?与non?static?class的区别

内部静态类**不需要**有指向外部类的**引用**。但非静态内部类**需要**持有对外部类的**引用**。

非静态内部类能够**访问外部类的静态和非静态成员**。静态类**不能访问外部类的非静态成员**。他只能访问外部类的静态成员。一个非静态内部类不能脱离外部类实体被创建，一个非静态内部类**可以访问外部类的数据和方法**，因为他就在外部类里面。java多态的实现原理。

1. foreach与正常for循环效率对比

对于数组来说，for和foreach循环效率差不多，但是对于链表来说，for循环效率明显比foreach低。

1. **Java?IO与NIO\***

NIO是为了弥补IO操作的不足而诞生的，NIO的一些新特性有：非阻塞I/O，选择器，缓冲以及管道。管道（Channel），缓冲（Buffer） ，选择器（ Selector）是其主要特征

IO是面向流的，NIO是面向块（缓冲区）的。

IO是阻塞的，NIO是非阻塞的。

多连接，少数据可以用NIO

少连接，大数据可以用IO

1. **java反射的作用与原理\***

<https://blog.csdn.net/sinat_38259539/article/details/71799078>

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

在JAVA中，只有给定类的名字，就可以通过反射机制来获取类的所有信息，可以动态的创建对象和编译。

JAVA语言编译之后会生成一个.class文件，反射就是通过**字节码文件**找到某一个类、类中的方法以及属性等。

1. 泛型常用特点

泛型是具有占位符（类型参数）的类、结构、接口和方法，这些占位符是类、结构、接口和方法所存储或使用的一个或多个类型的占位符。

使用泛型类型可以最大限度地**重用代码**、**保护类型的安全**以及**提高性能**。

通过**类型擦除**来实现

1. 解析XML的几种方式的原理与特点：DOM、SAX

DOM分析器是把**整个XML文档**转化为**DOM树**放在内存中

SAX解析采用**事件驱动**，通过事件**处理函数**实现对xml文档的访问。

1. Java1.7与1.8,1.9,10 新特性

**1.7**  
**1.switch中可以使用字串了**  
2.运用List tempList = new ArrayList<>(); 即泛型实例化类型自动推断  
3.语法上支持集合，而不一定是数组  
4.新增一些取环境信息的工具方法  
5.Boolean类型反转，空指针安全,参与位运算  
6.两个char间的equals  
7.安全的加减乘除  
8.map集合支持并发请求，且可以写成 Map map = {name:"xxx",age:18};  
  
**1.8**  
1. 允许在接口中有默认方法实现  
**2. Lambda表达式**  
3. 函数式接口  
4. 方法和构造函数引用  
5. Lambda的范围  
6. 内置函数式接口  
7. Streams  
8. Parallel Streams  
**9. Map**  
10. 时间日期API  
11. Annotations  
  
**1.9**  
1. Jigsaw 项目;模块化源码  
2. 简化进程API  
3. 轻量级 JSON API  
4. 钱和货币的API  
5. 改善锁争用机制  
6. 代码分段缓存  
7. 智能Java编译, 第二阶段  
**8. HTTP 2.0客户端**  
9. Kulla计划: Java的REPL实现

1. 设计模式：单例、工厂、适配器、责任链、观察者等等

单例：

public class Singleton1 {

public static final Singleton1 instance = new Singleton1();

private Singleton1(){

}

public static Singleton1 getInstance(){

return instance;

}

public class Singleton3 {

private static class SingletonHolder {

*//静态初始化器，由JVM来保证线程安全*

private static Singleton3 instance = new Singleton3();

}

private Singleton3() {

}

public static Singleton3 getInstance() {

return SingletonHolder.instance;

}

1. **JNI的使用\***

**jni是一种协议**，这个协议用来**沟通java代码和外部的本地代码（c/c++）**，通过这个协议，java代码就可以调用外部的c++代码。

1、在java本地代码中声明一个native方法：例如：public native String helloJni();

2、在eclipse中创建一个文件夹，名称必须命名为jni；

3、在jni这个文件夹下创建一个.c文件，按照c代码的规范来写

4、ndk-build.cmd指令编译c代码（注意：如果不配置Android.mk文件的话就会报错）；

5、配置Android.mk文件；

6、编译过后，自动生成一个.so的动态链接库；

7、在java代码中，把动态链接库加载到jvm虚拟机中加入一个静态代码块

8、像调用java代码一样，调用native方法；

1. **AOP是什么\***

面向切面编程，不影响功能的情况下添加内容扩展，比如添加log，权限等，通过**Aspect切面**，把**业务共同调用的逻辑或者责任封装起来**，减少重复代码，降低模块之间的耦合度。

1、横切关注点

对哪些**方法**进行拦截，拦截后怎么处理，这些关注点称之为**横切关注点**

2、切面（aspect）

类是对物体特征的抽象，**切面**就是对**横切关注点的抽象**

3、连接点（joinpoint）

被拦截到的点，因为**Spring只支持方法类型的连接点**，所以在Spring中连接点指的就是被拦截到的方法，实际上**连接点还可以是字段或者构造器**

4、切入点（pointcut）

对**连接点进行拦截**的定义

5、通知（advice）

所谓通知指的就是指**拦截到连接点之后要执行的代码**，通知分为**前置、后置、异常、最终、环绕**通知五类

6、目标对象

代理的目标对象

7、织入（weave）

将**切面应用到目标对象并导致代理**对象创建的过程

8、引入（introduction）

在不修改代码的前提下，引入可以在**运行期为类动态地添加一些方法或字段**

1. OOP是什么

面向对象编程，一种编程思想，万物皆对象

1. **AOP与OOP的区别\***

**AOP:** (Aspect Oriented Programming) **面向切面**编程。是目前软件开发中的一个热点，也是Spring框架中容。利用AOP可以对业务逻辑的**各个部分进行隔离**，从而使得业务逻辑各部分之间的**耦合度降低**，提高程序的**可重用性**，同时提高了**开发的效率**。主要的功能是：**日志记录，性能统计，安全控制，事务处理，异常处理**等等。

AOP、OOP在字面上虽然非常类似，但却是面向不同领域的两种设计思想。OOP（**面向对象**编程）针对业务处理过程的实体及其**属性和行为**进行抽象封装，以获得更加清晰高效的逻辑单元划分。 而AOP则是针**对业务处理过程中的切面进行提取**，它所面对的是处理过程中的某个步骤或阶段，以获得逻辑过程中各部分之间低耦合性的隔离效果。这两种设计思想在目标上有着本质的差异。

**多线程**

1. 什么是线程？

**线程**是进程的最小执行单元，是进程中的一个实体，是被系统独立调度和分派的基本单位，线程自己不拥有系统资源，只拥有一点在运行中必不可少的资源，但它可与同属一个进程的其它线程共享进程所拥有的全部资源。一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。线程有就绪、阻塞和运行三种基本状态。

**1、继承Thread类，并重写run函数**

**2、实现Runnable接口，并重写run函数**

1. 什么是线程安全和线程不安全？

**线程安全**就是在多线程环境下也不会出现数据不一致，而非线程安全就有可能出现数据不一致的情况。

线程安全由于要确保**数据的一致性**，所以对**资源的读写**进行了**控制**，换句话说增加了系统开销。所以在单线程环境中效率比非线程安全的效率要低些，但是如果线程间数据相关，需要保证读写顺序，用线**程安全模式**。

1. **什么是自旋锁？\***

<https://www.zhihu.com/question/66733477/answer/246535792>

**没有锁上，就不断重试，**如果别的线程长期持有该锁，那么你的线程就一直在while地检查是否能够加锁，浪费CPU无用功，

**互斥锁，就是在这个基础上，线程等待别的线程操作完后被唤醒。**

互斥锁：线程会从sleep（加锁）——>running（解锁），过程中有上下文的切换，cpu的抢占，信号的发送等开销。

自旋锁：线程一直是running(加锁——>解锁)，死循环检测锁的标志位，机制不复杂。

1. **什么是Java内存模型？\***

<https://www.cnblogs.com/nexiyi/p/java_memory_model_and_thread.html>

Java内存模型的主要目标是定义程序中**各个变量的访问规则**，即在虚拟机中将变量存储到**内存和从内存中取出变量这样底层细节**。此处的变量与Java编程时所说的变量不一样，指包括了**实例字段、静态字段和构成数组对象**的元素，但是不包括局部变量与方法参数，后者是线程私有的，不会被共享。

**主内存与工作内存**

**内存间交互操作**

**重排序**

**同步机制原子性、可见性与有序性**

**先行发生元则**

1. **什么是CAS？\***

对于并发控制而言，锁是一种**悲观策略**，会阻塞线程执行。而无锁是一种**乐观策略**，它会假设对资源的访问时没有冲突的，既然没有冲突就不需要等待，线程不需要阻塞。那多个线程共同访问临界区的资源怎么办呢，**无锁的策略**采用一种比较交换技术**CAS（compare and swap）**来鉴别线程冲突，一旦检测到冲突，就充实当前操作直到没有冲突为止。基于这样的算法，CAS操作即使没有锁，也可以发现其他线程对当前线程的干扰（临界区值的修改），并进行恰当的处理。

**java.util.concurrent.atomic包**

<https://blog.csdn.net/liubenlong007/article/details/53761730>

1. 什么是乐观锁和悲观锁？

**悲观锁**：悲观锁指对数据被意外修改持保守态度，依赖数据库原生支持的锁机制来保证当前事务处理的安全性，防止其他并发事务对目标数据的破坏或破坏其他并发事务数据，**将在事务开始执行前或执行中申请锁定，执行完后再释放锁定。**这对于长事务来讲，可能会严重影响系统的并发处理能力

**乐观锁**：乐观锁相对悲观锁而言，先假想数据不会被并发操作修改，没有数据冲突，只在**数据进行提交更新的时候，才会正式对数据的冲突与否进行检测**，如果发现**冲突**了，则**宣告失败**，否则更新数据。这就要求**避免使用长事务和锁机制**，以免导致系统并发处理能力降低，保障系统生产效率。

1. **什么是****AQS？\***

java.util.concurrent.locks 里面的锁机制就是基于AQS机制。

高并发框架，AbstractQueuedSynchronizer, **AQS是JDK下提供的一套用于实现基于FIFO等待队列的阻塞锁和相关的同步器的一个同步框架**。它为不同场景提供了实现锁及同步机制的基本框架，为同步状态的**原子性管理、线程的阻塞、线程的解除阻塞及排队管理**提供了一种通用的机制。

1. 什么是原子操作？在Java Concurrency API中有哪些原子类(atomic classes)？

**原子操作**是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

int++并不是一个原子操作，所以当一个线程读取它的值并加1时，另外一个线程有可能会读到之前的值，这就会引发错误。

为了解决这个问题，必须保证增加操作是原子的，在JDK1.5之前我们可以使用**同步技术来做到这一点**。到JDK1.5，java.util.concurrent.atomic包提供了int和long类型的包装类，它们可以自动的保证对于他们的操作是原子的并且不需要使用同步。

**原子更新方式**

原子更新基本类型

原子更新数组

原子更新引用

原子更新属性（字段）

**原子更新基本类型**

AtomicBoolean ：原子更新布尔类型

AtomicInteger： 原子更新整型

AtomicLong: 原子更新长整型

**原子更新数组**

AtomicIntegerArray ：原子更新整型数组里的元素

AtomicLongArray :原子更新长整型数组里的元素

AtomicReferenceArray : 原子更新引用类型数组的元素

AtomicBooleanArray ：原子更新布尔类型数组的元素

**原子更新引用类型**

AtomicReference ：原子更新引用类型

AtomicReferenceFieldUpdater ：原子更新引用类型里的字段

AtomicMarkableReference：原子更新带有标记位的引用类型。可以原子更新一个布尔类型的标记位和应用类型

**原子更新字段类**

AtomicIntegerFieldUpdater:原子更新整型的字段的更新器

AtomicLongFieldUpdater：原子更新长整型字段的更新器

AtomicStampedReference:原子更新带有版本号的引用类型。该类将整型数值与引用关联起来，可用于原子的更新数据和数据的版本号，可以解决使用CAS进行原子更新时可能出现的ABA问题。

1. **什么是Executors框架？\***

Executor框架同java.util.concurrent.Executor 接口在Java 5中被引入。Executor框架是一个根据一组执行**策略调用，调度，执行和控制的异步任务的框架**。

**无限制的创建线程会引起应用程序内存溢出**。所以创建一个线程池是个更好的的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。利用Executors框架可以非常方便的创建一个线程池。

1. **什么是阻塞队列？如何使用阻塞队列来实现生产者-消费者模型？\***

java.util.concurrent.BlockingQueue的特性是：**当队列是空**的时，从队列中**获取或删除**元素的操作将**会被阻塞**，或者当**队列是满时**，往队列里添加元素的操作**会被阻塞**。

阻塞队列不接受空值，当你尝试向队列中添加空值的时候，它会抛出NullPointerException。

阻塞队列的实现都是线程安全的，所有的查询方法都是原子的并且使用了内部锁或者其他形式的并发控制。

BlockingQueue 接口是java collections框架的一部分，它主要用于实现生产者-消费者问题。

1. **什么是Callable和Future?\***

Java 5在concurrency包中引入了java.util.concurrent.Callable 接口，它和Runnable接口很相似，但它可以返回一个对象或者抛出一个异常。

**Callable接口**使用泛型去定义它的**返回类型**。Executors类提供了一些有用的方法去在线程池中执行Callable内的任务。由于**Callable任务**是并行的，我们必须等待它返回的结果。java.util.concurrent.Future对象为我们解决了这个问题。**在线程池提交Callable任务后返回了一个Future对象**，使用它我们可以知道Callable任务的状态和得到Callable返回的执行结果。Future提供了get()方法让我们可以等待Callable结束并获取它的执行结果.

1. **什么是FutureTask?\***

**FutureTask是Future的一个基础实现**，我们可以将它同Executors使用处理异步任务。通常我们不需要使用FutureTask类，但当我们打算重写Future接口的一些方法并保持原来基础的实现是，它就变得非常有用。我们可以仅仅继承于它并重写我们需要的方法。

1. **什么是同步容器和并发容器的实现？\***

数组，对象，集合等等都是容器都叫容器，Vector和HashTable是同步容器，实现线程安全的方式就是将它们的状态封装起来，并在需要同步的方法上加上关键字synchronized。

ConcurrentHashMap是并发容器，它允许完全并发的读取，并且支持给定数量的并发更新。

1. 什么是多线程？优缺点？

多线程就是指一个进程中同时有**多个执行路径（线程）**正在执行，

1.在一个程序中，有很多的操作是非常耗时的，如**数据库读写操作，IO操作**等，如果使用单线程，那么程序就必须等待这些操作执行完成之后才能执行其他操作。使用多线程，可以在将耗时任务放在后台继续执行的同时，同时执行其他操作。

2.可以提高程序的效率。

3.在一些等待的任务上，如用户输入，文件读取等，多线程就非常有用了。

**缺点：**

1.使用太多线程，是**很耗系统资源，**因为线程需要开辟内存。更多线程**需要更多内存**。

2.影响系统性能，因为操作系统需要在线程之间来回切换。

3.需要考虑线程操作对程序的影响，如线程挂起，中止等操作对程序的影响。

4.线程使用不当会发生很多问题。

1. **什么是多线程的上下文切换？\***

cpu通过**时间片分配算法**来循环执行任务，当前任务执行一个时间片后切换到下一个任务。但是，在**切换前**会**保存上一个任务的状态**，以便下次切换回这个任务时，可以再加载这个任务的状态。所以**任务从保存到再加载的过程就是一次上下文切换**。

多线程环境中，当一个线程的状态由Runnable转换为非Runnable（Blocked、Waiting、Timed\_Waiting）时，相应线程的上下文信息（包括**cpu的寄存器和程序计数器在某一时间点的内容**等）需要被保存，以便相应线程稍后再次进入Runnable状态时能够在之前的执行进度的基础上继续前进。而一个线程从非Runnable状态进入Runnable状态可能涉及恢复之前保存的上下文信息。**这个对线程的上下文进行保存和恢复的过程就被称为上下文切换。**

1. **ThreadLocal的设计理念与作用？\*（需要在深入下源代码原理）**

ThreadLocal是一个线程的**内部存储类**，可以在每个线程的内部存储数据，当某个数据的作用域应该对应线程的时候就应该使用它；

每个线程中都会维护一个**ThreadLocalMap**，当在某个线程中访问时，会取出这个线程自己的Map并且用当前**ThreadLocal对象做索引**来取出相对应的Value值，从而达到不同线程不同值的效果。

1. ThreadPool（线程池）用法与优势？

java中的线程池是通过Executor框架实现的，ThreadPoolExecutor：线程池的具体实现类,一般用的各种线程池都是基于这个类实现的。

第一：**降低资源消耗**。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

第二：**提高响应速度**。当任务到达时，任务可以不需要等到线程创建就能立即执行。

第三：**提高线程的可管理性**。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用**线程池可以进行统一的分配，调优和监控**。但是要做到合理的利用线程池，必须对其原理了如指掌。

1. Concurrent包里的其他东西：ArrayBlockingQueue、CountDownLatch等等。

**ArrayBlockingQueue**是规定大小的BlockingQueue，其构造函数必须带一个int参数来指明其大小。其所含的对象是以FIFO（先入先出）顺序排序的。

**CountDownLatch**是减计数方式，计数==0时释放所有等待的线程。

**CyclicBarrier**是加计数方式，计数达到构造方法中参数指定的值时释放所有等待的线程。

1. synchronized和ReentrantLock的区别？

ReentrantLock可重入锁可以指定公平锁还是非公平锁，具有锁等待，锁中断功能

Synchronized是在**JVM**层面上实现的 ，ReentranLock是在**JDK**实现的

在资源竞争不是很激烈的情况下，Synchronized的性能要优于ReetrantLock，但是在资源竞争很激烈的情况下，Synchronized的性能会下降几十倍，但是ReetrantLock的性能能维持常态

如果使用 synchronized ，如果**A不释放，B将一直等下去**，不能被中断

如果使用ReentrantLock，如果**A不释放，可以使B在等待了足够长的时间以后，中断等待**，而干别的事情

1. **Semaphore有什么作用？\***

<https://www.cnblogs.com/liuling/p/2013-8-20-03.html>

Semaphore(信号量)，

Semaphore就是一个信号量，**它的作用是限制某段代码块的并发数**。

Semaphore有一个构造函数，可以**传入一个int型整数n，表示某段代码最多只有n个线程可以访问**，

**如果超出了n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入**。

由此可以看出如果Semaphore构造函数中传入的int型整数n=1，相当于变成了一个synchronized了。

Semaphore类中比较重要的几个方法，首先是acquire()、release()方法：

acquire()用来获取一个许可，若无许可能够获得，则会一直等待，直到获得许可。

release()用来释放许可。注意，在释放许可之前，必须先获获得许可。

1. Java Concurrency API中的Lock接口(Lock interface)是什么？对比同步它有什么优势？

**Lock接口比同步方法和同步块提供了更具扩展性的锁操作**。他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。

它的优势有：

可以使锁更**公平**

可以使线程在等待锁的时候**响应中断**

可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间

可以在不同的范围，以不同的**顺序获取和释放锁**

1. Hashtable的size()方法中明明只有一条语句”return count”，为什么还要做同步？

同一时间只能有一条线程执行固定类的同步方法，但是对于类的非同步方法，可以多条线程同时访问。所以，这样就有问题了，可能线程A在执行Hashtable的put方法添加数据，线程B则可以正常调用size()方法读取Hashtable中当前元素的个数，那读取到的值可能不是最新的，可能线程A添加了完了数据，但是没有对size++，线程B就已经读取size了，那么对于线程B来说读取到的size一定是不准确的。

**而给size()方法加了同步之后，意味着线程B调用size()方法只有在线程A调用put方法完毕之后才可以调用，这样就保证了线程安全性**

**Put和读取多线程导致的问题。**

1. **ConcurrentHashMap的并发度是什么？\***

ConcurrentHashMap的并发度就是**segment**的大小，默认为16，这意味着最多同时可以有16条线程操作ConcurrentHashMap，这也是ConcurrentHashMap对Hashtable的最大优势

1. ReentrantReadWriteLock读写锁的使用？

**读写锁：分为读锁和写锁，多个读锁不互斥，读锁与写锁互斥，这是由jvm自己控制的，你只要上好相应的锁即可。**  
如果你的代码只读数据，可以很多人同时读，但不能同时写，那就上读锁；

如果你的代码修改数据，只能有一个人在写，且不能同时读取，那就上写锁。总之，读的时候上读锁，写的时候上写锁！

1. CyclicBarrier和CountDownLatch的用法及区别？

1.闭锁CountDownLatch做**减计数**，而栅栏CyclicBarrier则是**加计数**。  
2.CountDownLatch是一次性的，CyclicBarrier可以重用。  
3.CountDownLatch强调一个线程**等多个线程完成某件事情**。CyclicBarrier是**多个线程互等**，等大家都完成。  
4.鉴于上面的描述，CyclicBarrier在一些场景中可以替代CountDownLatch实现类似的功能。

1. LockSupport工具？

java.util.concurrent.locks，**LockSupprot是线程的阻塞原语，用来阻塞线程和唤醒线程**。每个使用LockSupport的线程都会与一个许可关联，如果该许可可用，并且可在线程中使用，则调用park()将会立即返回，否则可能阻塞。如果许可尚不可用，则可以调用 unpark 使其可用。但是注意许可**不可重入**，也就是说只能调用一次park()方法，否则会一直阻塞。

1. **Condition接口及其实现原理？\***

<https://blog.csdn.net/bohu83/article/details/51098106>

Condition定义了等待/通知两种类型的方法，当前线程调用这些方法时，需要提前获取到Condition对象关联的锁。Condition对象是由Lock对象（调用Lock对象的newCondition()方法）创建出来的，换句话说，**Condition是依赖Lock对象的**。

一般都会将Condition对象作为成员变量。当调用**await()**方法后，当前线程会释放锁并在此等待，而其他线程调用Condition对象的**signal()**方法，通知当前线程后，当前线程才从await()方法返回，并且在返回前已经获取了锁。

ConditionObject是同步器AbstractQueuedSynchronizer的内部类，因为Condition的操作需要获取相关联的锁，所以作为同步器的内部类也较为合理。每个Condition对象都包含着一个队列，该队列是Condition对象实现等待/通知功能的关键。下面将分析Condition的实现，主要包括：等待队列、等待和通知。

1. **Fork/Join框架的理解?\***

<https://blog.csdn.net/timheath/article/details/71307834>

Fork/Join框架是Java7提供了的一个用于并发执行任务的框架，是一个把**大任务分割成若干个小任务**，最终汇总每个小任务结果后得到大任务结果的框架, RecursiveAction和RecursiveTask.(**工作窃取模式**)

**ForkJoinPool**

**ForkJoinPool**是ForkJoin框架中的**任务调度器**，和ThreadPoolExecutor一样实现了自己的线程池，提供了三种调度子任务的方法：

**execute**：**异步**执行指定任务，**无返回结果**；

**invoke、invokeAll**：**异步**执行指定任务，等待完成才**返回结果**；

submit：**异步**执行指定任务，并立即返回一个**Future对象**；

**ForkJoinTask**

Fork/Join框架中的实际的执行任务类，有以下两种实现，一般继承这两种实现类即可。

**RecursiveAction**：用于**无结果返回的子任务**；

**RecursiveTask**：用于**有结果返回的子任务**

1. **wait()和sleep()的区别?\***

在调用**sleep()**方法的过程中，线程**不会释放对象锁**。

当调用**wait()**方法的时候，线程**会放弃对象锁**，进入等待此对象的**等待锁定池**，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。

1. **线程的五个状态（五种状态，创建、就绪、运行、阻塞和死亡）?\***

新建状态(New)——就绪状态(Runnable)——运行状态(Running)——阻塞状态(Blocked)——死亡状态(Dead)

1. start()方法和run()方法的区别？

一个新创建的线程并不自动开始运行，要执行线程，必须调用线程的**start()**方法。当线程对象调用start()方法即启动了线程，start()方法创建线程运行的系统资源，并调度线程运行run()方法。**当start()方法返回后，线程就处于就绪状态**。

处于就绪状态的线程并不一定立即运行run()方法，线程还必须同其他线程**竞争CPU时间**，只有获得CPU时间才可以运行线程。因为在**单CPU**的计算机系统中，不可能同时运行多个线程，一个时刻仅有一个线程处于运行状态。因此此时可能有多个线程处于就绪状态。对多个处于就绪状态的线程是由Java运行时系统的**线程调度程序**(thread scheduler)来调度的。

1. Runnable接口和Callable接口的区别？

(1)Callable规定的方法是**call(),**Runnable规定的方法是**run()**.  
(2)Callable的任务执行后可返回值，而Runnable的任务是不能返回值得  
(3)call方法可以抛出异常，run方法不可以  
(4)运行Callable任务可以拿到一个Future对象，表示异步计算的结果。它提供了检查计算是否完成的方法，以等待计算的完成，并检索计算的结果。通过Future对象可以了解任务执行情况，可取消任务的执行，还可获取执行结果。

1. Java中如何获取到线程dump文件？

【内存dump】

jmap –dump:live,format=b,file=heap.bin <pid>

【线程dump】

jstack -m <pid> >jvm\_deadlocks.txt

jstack -l <pid> >jvm\_listlocks.txt

1. **线程和进程有什么区别？\***

**进程是资源（CPU、内存等）分配的基本单位，它是程序执行时的一个实例。**程序运行时系统就会创建一个进程，并为它分配资源，然后把该进程放入进程就绪队列，进程调度器选中它的时候就会为它分配CPU时间，程序开始真正运行。

**线程是程序执行时的最小单位，它是进程的一个执行流，是CPU调度和分派的基本单位，**一个进程可以由很多个线程组成，线程间共享进程的所有资源，每个线程有自己的堆栈和局部变量。线程由CPU独立调度执行**，在多CPU环境下就允许多个线程同时运行**。同样多线程也可以实现并发操作，每个请求分配一个线程来处理。

1. **线程实现的方式有几种（四种）？\***

1、继承Thread，

2、实现Runable，

3、线程池框架使用ExecutorService、Callable、Future实现有返回结果的线程，

4、实现Callable接口通过FutureTask包装器来创建Thread线程

1. **高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？\***

（1）**高**并发、任务执行时间**短**的业务，线程池线程数可以设置为**CPU核数+1**，**减少线程上下文的切换**

（2）并发**不高**、任务执行时间**长**的业务要区分开看：

a）假如是业务时间长集中在**IO操作上**，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以**加大线程池中的线程数目**，让CPU处理更多的业务

b）假如是业务时间长集中在**计算操作上**，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的**线程数设置得少一些，减少线程上下文的切换**

（3）并**发高**、业务执行时间**长**，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做**缓存**是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

1. 如果你提交任务时，线程池队列已满，这时会发生什么？

这个问题问得很狡猾，许多程序员会认为该任务会阻塞直到线程池队列有空位。事实上如果一个任务不能被调度执行那么**ThreadPoolExecutor’s submit()方法**将会抛出一个**RejectedExecutionException**异常。

1. 锁的等级：方法锁、对象锁、类锁?

修饰代码块时，需要一个**reference对象**作为锁的对象。

修饰方法时，默认是当前对像作为锁的对象。

修饰类时，默认是当前类的**Class对象**作为锁的对象。

1. 如果同步块内的线程抛出异常会发生什么？

只要退出了synchronized块，无论是正常还是异常，都会释放锁。

1. **并发编程（concurrency）并行编程（parallellism）有什么区别？\***

并发（concurrency）和并行（parallellism）是：  
解释一：并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔发生。  
解释二：并行是在不同实体上的多个事件，并发是在同一实体上的多个事件。  
解释三：并发在一台处理器上“同时”处理多个任务，并行在多台处理器上同时处理多个任务。如hadoop分布式集群

1. 如何保证多线程下 i++ 结果正确？

AtomicInteger，一个提供原子操作的Integer的类。在Java语言中，++i和i++操作并不是线程安全的，在使用的时候，不可避免的会用到synchronized关键字。而**AtomicInteger**则通过一种线程安全的加减操作接口，**Volatile修饰的成员变量**在每次被线程访问时，都强迫从共享内存**重新读取**该成员的值，而且，当成员变量值发生变化时，强迫将变化的值重新写入**共享内存**，这样两个不同的线程在访问同一个共享变量的值时，始终看到的是同一个值。

1. 一个线程如果出现了运行时异常会怎么样?

如果该异常被**捕获或抛出**，则程序**继续运行**。   
如果异常**没有被捕获**该线程将会**停止执行**。   
Thread.UncaughtExceptionHandler是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。当一个未捕获异常将造成线程中断的时候JVM会使用Thread.getUncaughtExceptionHandler()来查询线程的UncaughtExceptionHandler，并将线程和异常作为参数传递给handler的uncaughtException()方法进行处理

1. **如何在两个线程之间共享数据?\***

**使用同一个runnable对象**

如果每个线程执行的代码相同,那么可以使用同一个runnable对象,这个**runnable有那个共享数据**,例如,卖票系统就是这么做的.

**使用不同的runnable对象**

1).实现两个runnable对象,将**共享数据分别传递**给两个不同线程.

2).将这些runnable对象作为一个**内部类**,将**共享数据作为成员变量**.

1. **生产者消费者模型的作用是什么?\***

**生产者消费者模型是多线程当中比较经典的一个模型，该模型模拟线程间公用同一个对象，通过调度不同的线程休眠、等待和唤醒起到预防死锁的作用。**

\* 1 同一时间内只能有一个生产者生产 生产方法加锁sychronized

\* 2 同一时间内只能有一个消费者消费 消费方法加锁sychronized

\* 3 共享空间空时消费者不能继续消费 消费前循环判断是否为空，空的话将该线程wait，释放锁允许其他同步方法执行

\* 4 共享空间满时生产者不能继续生产 生产前循环判断是否为满，满的话将该线程wait，释放锁允

1. **怎么唤醒一个阻塞的线程?\***

如果线程是因为调用了**wait()、sleep()或者join()**方法而导致的阻塞，可以**中断线程**，并且通过抛出InterruptedException来唤醒它；如果线程遇到了IO阻塞，无能为力，因为IO是操作系统实现的，**Java代码**并没有办法直接接触到操作系统。以下是详细的唤醒方法：

**1. sleep() 方法**

sleep（毫秒），指定以毫秒为单位的时间，使线程在该时间内进入线程阻塞状态，期间得不到cpu的时间片，等到时间过去了，线程重新进入可执行状态。（暂停线程，不会释放锁）

**2.suspend0() 和 resume0() 方法**

挂起和唤醒线程，suspende()使线程进入阻塞状态，只有对应的resume()被调用的时候，线程才会进入可执行状态。（不建议用，容易发生死锁）

**3. yield() 方法**

会使的**线程放弃当前分得的cpu时间片**，但此时线程任然处于可执行状态，随时可以再次分得cpu时间片。yield()方法只能使同优先级的线程有执行的机会。调用 yield()的效果等**价于调度程序认为该线程已执行了足够的时间从而转到另一个线程**。（暂停当前正在执行的线程，并执行其他线程，且让出的时间不可知）

**4.wait() 和 notify() 方法\***

两个方法搭配使用，wait()使线程进入阻塞状态，调用notify()时，线程进入可执行状态。wait()内可加或不加参数，加参数时是以毫秒为单位，当到了指定时间或调用notify()方法时，进入可执行状态。（属于Object类，而不属于Thread类，wait()会先释放锁住的对象，然后再执行等待的动作。由于wait()所等待的对象必须先锁住，因此，它只能用在同步化程序段或者同步化方法内，否则，会抛出异常IllegalMonitorStateException.

**5.join()方法**

也叫**线程加入**。是当前线程A调用另一个线程B的join()方法，当前线程转A入阻塞状态，直到线程B运行结束，线程A才由阻塞状态转为可执行状态。

以上是Java线程唤醒和阻塞的五种常用方法，不同的方法有不同的特点，其中wait() 和 notify()是其中功能最强大、使用最灵活的方法，但这也导致了它们效率较低、较容易出错的特性，因此，在实际应用中应灵活运用各种方法，以达到期望的目的与效果！

1. **Java中用到的线程调度算法是什么\***

[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)的核心，它实际就是一个[常驻内存](https://www.baidu.com/s?wd=%E5%B8%B8%E9%A9%BB%E5%86%85%E5%AD%98&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)的程序，不断地对线程队列进行扫描，利用特定的算法（[时间片轮转](https://www.baidu.com/s?wd=%E6%97%B6%E9%97%B4%E7%89%87%E8%BD%AE%E8%BD%AC&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)法、[优先级](https://www.baidu.com/s?wd=%E4%BC%98%E5%85%88%E7%BA%A7&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)调度法、多级反馈队列调度法等）找出比当前占有CPU的线程更有CPU使用权的线程，并从之前的线程中收回处理器，再使待运行的线程占用处理器。

1. 单例模式的线程安全性?

有一些对象只能使用一个，例如：**数据库连接、线程池（threadpool）、缓存（cache）、对话框、处理偏好（preferences）设置和这侧表（registry）**的对象、日志对象、充当打印机、显卡等设备的驱动程序的对象，即用于管理共享的资源。这种对象只能有一个实例，制造多个会导致问题。

线程不安全的话可能会返回两个实例，静态初始化器中创建单例实例，双重检查加锁。

1. 线程类的构造方法、静态块是被哪个线程调用的?

**线程类的构造方法、静态块是被new这个线程类所在的线程所调用的  
而run方法里面的代码才是被线程自身所调用的。**

1. 同步方法和同步块，哪个是更好的选择?

1. 语法不同。  
2. 同步块需要注明**锁定对象**，**同步方法默认锁定this**。  
3. 在静态方法中，都是**默认锁定类对象**。  
4. 在考虑性能方面，最好使用**同步块**来减少锁定范围**提高并发效率**。

1. **如何检测死锁？怎么预防死锁？\***

指**两个或两个以上的进程**在执行过程中，**因争夺资源而造成的一种互相等待的现象**，若无外力作用，它们都将**无法推进下去**。此时称系统处于死锁。

**资源一次性分配**：（破坏请求和保持条件）。

**可剥夺资源**：即当某进程新的资源未满足时，释放已占有的资源（破坏不可剥夺条件）

**资源有序分配法**：系统给每类资源赋予一个编号，每一个进程按编号递增的顺序请求资源，释放则相反（破坏环路等待条件）。

**设计模式**

1. 装饰器模式

一个接口component，一个具体实现类concreteComponent，一个装饰类Decorator，不同的装饰实现类（super构造）ConcreteDecoratorA,ConcreteDecoratorB.

//Component 英雄接口

public interface Hero {

//学习技能

void learnSkills();

}

//ConcreteComponent 具体英雄盲僧

public class BlindMonk implements Hero {

private String name

public BlindMonk(String name) {

this.name = name;

}

@Override

public void learnSkills() {

System.out.println(name + "学习了以上技能！");

}

}

//Decorator 技能栏 **装饰类**

public class Skills implements Hero{

//持有一个英雄对象接口

private Hero hero;

public Skills(Hero hero) {

this.hero = hero;

}

@Override

public void learnSkills() {

if(hero != null)

hero.learnSkills();

}

}

//ConreteDecorator 技能：Q **具体装饰类实现类**

public class Skill\_Q extends Skills{

private String skillName;

public Skill\_Q(Hero hero,String skillName) {

super(hero);

this.skillName = skillName;

}

@Override

public void learnSkills() {

System.out.println("学习了技能Q:" +skillName);

super.learnSkills();

}

}

//ConreteDecorator 技能：W

public class Skill\_W extends Skills{

private String skillName;

public Skill\_W(Hero hero,String skillName) {

super(hero);

this.skillName = skillName;

}

@Override

public void learnSkills() {

System.out.println("学习了技能W:" + skillName);

super.learnSkills();

}

}

//ConreteDecorator 技能：E

public class Skill\_E extends Skills{

private String skillName;

public Skill\_E(Hero hero,String skillName) {

super(hero);

this.skillName = skillName;

}

@Override

public void learnSkills() {

System.out.println("学习了技能E:"+skillName);

super.learnSkills();

}

}

//ConreteDecorator 技能：R

public class Skill\_R extends Skills{

private String skillName;

public Skill\_R(Hero hero,String skillName) {

super(hero);

this.skillName = skillName;

}

@Override

public void learnSkills() {

System.out.println("学习了技能R:" +skillName );

super.learnSkills();

}

}

//客户端：召唤师

public class Player {

public static void main(String[] args) {

//选择英雄

Hero hero = new BlindMonk("李青");

Skills skills = new Skills(hero);

Skills r = new Skill\_R(skills,"猛龙摆尾");

Skills e = new Skill\_E(r,"天雷破/摧筋断骨");

Skills w = new Skill\_W(e,"金钟罩/铁布衫");

Skills q = new Skill\_Q(w,"天音波/回音击");

//学习技能

q.learnSkills();

}

}

1. **工厂模式**

通过使用一个共同的接口来指向新创建的对象

一个共同接口，多个具体实现类，一个工厂类根据信息产生对象

抽象工厂模式

在简单的工厂模式上，2个工厂类封装

多个接口，每个接口对应多个具体实现类，抽象接口工厂用来获取对应接口工厂，接口工厂继承抽象工厂，外加实现类信息获取具体实现类

1. **单例模式\***

一般情况下，不建议使用第 1 种和第 2 种懒汉方式，建议使用第 3 种饿汉方式。只有在要明确实现 lazy loading 效果时，才会使用第 5 种登记方式。如果涉及到反序列化创建对象时，可以尝试使用第 6 种枚举方式。如果有其他特殊的需求，可以考虑使用第 4 种双检锁方式。

**3、饿汉式**

public class Singleton {

private static Singleton instance = new Singleton();

private Singleton (){}

public static Singleton getInstance() { return instance; }

}

**4、双检锁/双重校验锁（DCL，即 double-checked locking**

public class Singleton {

private volatile static Singleton singleton;

private Singleton (){}

public static Singleton getSingleton() {

if (singleton == null)

{ synchronized (Singleton.class)

{

if (singleton == null)

{ singleton = new Singleton(); }

}

}

return singleton;

}

}

**6、枚举**

public enum Singleton {

INSTANCE;

public void whateverMethod() { }

}

1. **观察者模式\***

当对象间存在一对多关系时，则使用观察者模式（Observer Pattern）。比如，当一个对象被修改时，则会自动通知它的依赖对象。观察者模式属于行为型模式。

观察者模式使用三个类 Subject、Observer 和 Client。

public abstract class Observer {

public abstract void update(String msg);

}

第一个观察者：

public class F\_Observer extends Observer {

public void update(String msg) {

System.out.println(F\_Observer.class.getName() + " : " + msg);

}

}

第二个观察者：

public class S\_Observer extends Observer {

public void update(String msg) {

System.out.println(S\_Observer.class.getName() + " : " + msg);

}

}

第三个观察者：

public class T\_Observer extends Observer {

public void update(String msg) {

System.out.println(T\_Observer.class.getName() + " : " + msg);

}

}

被观察者：

public class Subject {

private List<Observer> observers = new ArrayList<>(); //状态改变

public void setMsg(String msg) {

notifyAll(msg);

}

//订阅

public void addAttach(Observer observer) {

observers.add(observer);

}

//通知所有订阅的观察者

private void notifyAll(String msg) {

for (Observer observer : observers) {

observer.update(msg);

}

}

}

使用方法：

public class Main {

public static void main(String[] args) {

F\_Observer fObserver = new F\_Observer();

S\_Observer sObserver = new S\_Observer();

T\_Observer tObserver = new T\_Observer();

Subject subject = new Subject();

subject.addAttach(fObserver);

subject.addAttach(sObserver);

subject.addAttach(tObserver);

subject.setMsg("msg change");

}

}

1. **动态代理模式\***

委托类，中间类，代理类，**中间类要实现InvocationHan**，重写invoke方法。

静态代理类就想供应商和微商的赶脚，微商代理，可以做过滤处理。

而invoke动态代理，就可以在中间类中去过滤，中间类去掉用代理类中的方法。

1. 适配器模式

将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

**类适配器，对象适配器，接口适配器**

1. 模板模式

public abstract class Game {

abstract void initialize();

abstract void startPlay();

abstract void endPlay(); //模板

public final void play(){ //初始化游戏 initialize(); //开始游戏 startPlay(); //结束游戏 endPlay(); } }

1. 策略模式

一个策略接口，多个实现类，通过构造方法选择。

**JVM**

1. **内存模型以及分区，需要详细到每个区放什么？\***

**程序计数器：**

一块较小的内存空间，它的作用可以看做是**当前线程所执行的字节码**的行号**指示器**。

**栈：**  
Java虚拟机栈描述的是java方法执行的内存模型，每个方法被执行的时候都会创建一个**栈帧用于存储局部变量表、操作栈、动态链接、方法出口等信息**。线程私有。生命周期与线程相同。   
**本地方法栈**为虚拟机使用到的Native方法服务。

**堆：**  
**存放对象实例**，内存最大，所有线程共享，GC堆。

**方法区（非堆）：**  
与堆一样，线程共享，存放虚拟机加载**的类信息、常量、静态变量、即时编译器编译后的代码**等数据。

**运行时常量池：**   
方法区的一部分，**存放编译期生成的各种字面量和符号引用**。

**直接内存：**  
不是虚拟机运行时数据区的一部分，也不是虚拟机规范中定义的内存区域，**频繁被使用**，例如NIO，chnnel和buffer，避免java对和Native堆来回复制数据。

1. **对象创建方法，对象的内存分配，对象的访问定位。\***

JVM遇到一个new指令，首先去检查这个指令的参数能否在常量池中找到一个类的符号引用，并且检查这个符号引用所代表的类是否已经被**加载、解析、初始化过**，如果没有，将先进行类加载的过程。在类加载检查通过后，虚拟机就要为**新手对象分配内存**，锁需内存的大小在类加载完成后便可以完全确定，为对象分配空间的任务等同于把一块确定大小的 内存从java堆中划分出来，

**分配内存有两种方式：**

**指针碰撞**

假设**Java堆中的对象是绝对规整**的，所有用过的放一边，没用过的放另一边，中间有一个指针作为一个分界的指示器，那**所分配内存**就仅仅是把**那个指针向空闲空间那边挪动一段与对象大小相等的距离**，这种分配方式称为“指针碰撞”。

**空闲列表**

如果**Java堆中的内存并不是规整**的，已使用的内存和未使用的内存相互相错，这时就没办法用指针碰撞的方式，这时虚拟机就必须维护一个列表，用以记录哪些内存块是可用的，在分配的时候从列表中找出一块足够大的空间划分给对象实例，并更新列表上的记录，这种分配方式称为“空闲列表”。

**考虑线程安全的处理方式:**

分配空间的动作进行同步—cas

分配空间的动作按线程划分在不同的空间之中进行

对象的访问：主流的虚拟机有两种——**使用句柄和直接指针**

**句柄访问方式**，java堆中将会划分出一块内存来作为句柄池，**reference中**存储的就是对象的句柄地址，而句柄中包含了**对象实例数据和类型数据各自的具体地址信息**。

**直接指针访问方式**，Java堆对象的布局中就必须考虑如何放置访问类型数据的相关信息，**reference中直接存储的就是对象地址**。

**1：对象的创建**包括三步骤：①当遇到new命令的时候，会在常量池中**检查**该对象的符号引用**是否存在**，不存在则进行**类的加载**，否则执行下一步②分配内存，将将要分配的**内存都清零**。③虚拟机进行必要的设置，如设置**hashcode，gc的分代年龄**等，此时会执行<init>命令在执行之前所有的字段都为0，执行<init>指令以后，安装程序的意愿进行初始化字段。

**2：对象的内存分配**：包括对象头，实例数据，对齐填充

①**对象头**:包括对象的hascode，gc分代年龄，锁状态标等。

②**实例数据**:也就是初始化以后的对象的字段的内容，包括父类中的字段等

③**对齐填充**：对象的地址是8字节，虚拟机要求对象的大小是对象的整数倍(1倍或者两倍)。因此就会有空白区。

**3：对象的访问**：hotspan中 是**采用对象直接指向对象地址的方式**（这样的方式访问比较快）（还有一种方式就是句柄，也就是建一张表维护各个指向各个地址的指针，然后给指针设置一个句柄 （别名），然后引用直接指向这个别名，就可以获得该对象，这种的优势就是，实例对象地址改变了，只要修改句柄池中的指针就可以了，而不用引用本身不会发生改变）。

1. **GC的两种判定方法：引用计数与引用链。\***

**1：引用计数**：给一个对象设置一个计数器，当被引用一次就加1，当引用失效的时候就减1，如果该对象长时间保持为0值，则该对象将被标记为回收。优点：算法简单，效率高，缺点：很难解决对象之间的相互循环引用问题。

**2：引用链（可达性分析）**：现在主流的gc都采用可达性分析算法来判断对象是否已经死亡。可达性分析：通过一系列成为GC Roots的对象作为起点，从这些起点向下搜索，搜索所走过的路径成为引用链，当一个对象到引用链没有相连时，则判断该对象已经死亡。

**3：可作为gc roots的对象**：虚拟机栈（本地方法表）中引用的对象（因为在栈内，被线程引用），方法区中类静态属性引用的对象，方法区中常量引用的（常量存放在常量池中，常量池是方法区的一部分）对象，native方法引用的对象

**4：引用计数和引用链**是只是用来标记，判断一个对象是否失效，而不是用来清除。

1. **GC的三种收集方法：标记清除、标记整理、复制算法的原理与特点，分别用在什么地方，如果让你优化收集方法，有什么思路？\***

**标记-清除算法：**首先标记处所有需要回收的对象，在标记完成后统一回收掉所欲被标记的对象。（效率不高，易产生大量不连续的内存碎片）

复制算法：两块内存，原有一块内存所有存活对象全部复制到另外一块上，然后把上一块整体清除。（简单，高效，内存减半）

**标记-整理算法**：多有存活对象都向一段移动，然后直接清理掉段边界以外的内存。

**分代收集算法：**把堆分为新生代和老年代，新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存放，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成手机。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标记-整理算法”

1. GC收集器有哪些？CMS收集器与G1收集器的特点。

Serial收集器，ParNew收集器，Parallel Scavernge收集器，Serial Old收集器，Parallel Old收集器，CMS收集器，G1收集器。

CMS收集器以标记-清除，快，并发收集，低停顿。

G1收集器以标记-整理，并行+并发的垃圾收集器，老年代和新生代区域收集

1. **Minor?GC与Full?GC分别在什么时候发生？\***

Minor GC触发条件：当Eden区满时，触发Minor GC

Full GC触发条件：

（1）调用System.gc时，系统建议执行Full GC，但是不必然执行

（2）老年代空间不足

（3）方法区空间不足

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

1. 几种常用的内存调试工具：jmap、jstack、jconsole。

Jmap：Java内存映像工具

Jstack：Java堆栈跟踪工具

Jconsole：Java监视与管理控制台

1. 类加载的五个过程：加载、验证、准备、解析、初始化。

**加载：**

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）：将这个字节流锁代表的静态存储结构转化为方法区的运行时数据结构

3）：在Java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口。

**验证：**

验证是连接阶段的第一步，这一阶段的木得是未了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会畏寒虚拟机自身的安全。

文件格式验证、元数据验证、字节码验证和符号引用验证。

**准备：**

正式位类变量分派内存并设置类变量初始值的阶段，这些内存都将在方法区中进行分配。

Public static int value = 123；准备初始值为0

Public static final int value = 123；

编译时javac将值生成ConstantValue属性；准备初始值为123

**解析：**

1. 类或接口的解析
2. 字段解析
3. 类方法解析
4. 接口方法解析

**初始化：**

对静态变量和静态代码块执行初始化工作。

1)  加载：根据查找路径找到相应的class文件，然后导入。类的加载方式分为

**隐式加载**和**显示加载**两种。隐式加载指的是程序在使用new关键词创建对象时，会隐式的调用**类的加载器**把对应的类**加载到jvm**中。**显示加载**指的是通过直接调用class.forName（）方法来把所需的类加载到jvm中。

2)  检查：检查加载的class文件的正确性。

3)  准备：给类中的静态变量分配内存空间。

4)  解析：虚拟机将常量池中的符号引用替换成直接引用的过程。符号引用就理解为一个标示，而在直接引用直接指向内存中的地址。

5)  初始化：对静态变量和静态代码块执行初始化工作。

1. **双亲委派模型：Bootstrap?ClassLoader、Extension?ClassLoader、ApplicationClassLoader。\***

启动类加载器

扩展类加载器

应用程序类加载器

组合关系，双亲委派模型的工作过程是：如果一个类加载器收到类加载的请求，它首先**不会自己去尝试加载这个类**，而是把这个**请求委派给父类加载器**去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求(它的搜索范围中没有找到所需的类)时，子加载器才会尝试自己去加载。

1. 分派：静态分派与动态分派。

静态分派：所有依赖静态类型来定位方法执行版本的分派动作，都称为静态分派。（重载）**在重载的时候是通过参数的静态类型而不是实际类型作为判断依据的**

动态分派：（重写）

1. JVM过去过来就问了这么些问题，没怎么变，内存模型和GC算法这块问得比较多，可以在网上多找几篇博客来看看。
2. 推荐书籍：《深入理解java虚拟机》

**数据结构与算法**

1. 链表与数组。

数组：ArrayList—**静态分配内存**，内存**连续**。数组元素在**栈区**。

链表：LinkedList—手持下一个人的地址，**动态内存分配**，内存**不连续**。数组元素在**堆区**

1. 队列和栈，出栈与入栈。

**queue队列是先进先出**

入栈，s.push(x)  
出栈，s.pop()  
访问栈顶，s.top()  
判断栈空，s.empty()  
访问栈中的元素个数，s.size()

**stack栈是先入后出**

入队，q.push(x)  
出队，q.pop()  
访问队首元素，q.front()、访问队尾元素，q.back()  
判断队列空，q.empty()  
访问队列中的元素个数，q.size()

1. **链表的删除、插入、反向。\***

<https://blog.csdn.net/guyuealian/article/details/51119499>

数据域，指针域，当前节点

**删除**需要找到上一个节点，然后指向下下节点，size减去1

**插入**，找上一个节点，设置next到新节点，新节点初始化上一节点的current.next

**反向：**head.getnext获取末尾节点，然后以此重新指向反转（递归反转法）

遍历反转法，按顺序依次反转。

1. 字符串操作。

(1)**字符串的连接**  
public String concat(String str)   
该方法的参数为一个String类对象,作用是将参数中的字符串str连接到原来字符串的后面.   
(2)**求字符串的长度**  
public int length()  
返回字串的长度,这里的长度指的是字符串中Unicode字符的数目.  
(3)**求字符串中某一位置的字符**  
public char charAt(int index)  
该方法在一个特定的位置索引一个字符串,以得到字符串中指定位置的字符.值得注意的是,在字符串中第一个字符的索引是0,第二个字符的索引是1,依次类推,最后一个字符的索引是length()-1.   
(4)**字符串的比较**  
比较字符串可以利用String类提供的下列方法:  
1)public int compareTo(String anotherString)  
该方法比较两个字符串,和Character类提供的compareTo方法相似,Character类提供的compareTo方法比较的是两个字符类数据,而这里比较的是字符串数据.  
其比较过程实际上是两个字符串中相同位置上的字符按Unicode中排列顺序逐个比较的结果.如果在整个比较过程中,没有发现任何不同的地方,则表明两个字符串是完全相等的,compareTo方法返回0;如果在比较过程中,发现了不同的地方,则比较过程会停下来,这时一定是两个字符串在某个位置上不相同,如果当前字符串在这个位置上的字符大于参数中的这个位置上的字符,compareTo方法返回一个大于0的整数,否则返回一个小于0的整数.   
2)public boolean equals(Object anObject)  
该方法比较两个字符串,和Character类提供的equals方法相似,因为它们都是重载Object类的方法.该方法比较当前字符串和参数字符串,在两个字符串相等的时候返回true,否则返回false.  
3)public boolean equalsIgnoreCase(String anotherString)  
该方法和equals方法相似,不同的地方在于,equalsIgnoreCase方法将忽略字母大小写的区别.  
(5)**从字符串中提取子串**  
利用String类提供的substring方法可以从一个大的字符串中提取一个子串,该方法有两种常用的形式:  
1)public String substring(int beginIndex)  
该方法从beginIndex位置起,从当前字符串中取出剩余的字符作为一个新的字符串返回.  
2)public String substring(int beginIndex, int endIndex)  
该方法从当前字符串中取出一个子串,该子串从beginIndex位置起至endIndex-1为结束.子串返的长度为endIndex-beginIndex.   
(6)**判断字符串的前缀和后缀**  
判断字符串的前缀是否为指定的字符串利用String类提供的下列方法:  
1)public boolean startsWith(String prefix)  
该方法用于判断当前字符串的前缀是否和参数中指定的字符串prefix一致,如果是,返回true,否则返回false.  
2)public boolean startsWith(String prefix, int toffset)  
该方法用于判断当前字符串从toffset位置开始的子串的前缀是否和参数中指定的字符串prefix一致,如果是,返回true,否则返回false.  
判断字符串的后缀是否为指定的字符串利用String类提供的方法:  
public boolean endsWith(String suffix)  
该方法用于判断当前字符串的后缀是否和参数中指定的字符串suffix一致,如果是,返回true,否则返回false.  
(7)**字符串中单个字符的查找**  
字符串中单个字符的查找可以利用String类提供的下列方法:  
1)public int indexOf(int ch)  
该方法用于查找当前字符串中某一个特定字符ch出现的位置.该方法从头向后查找,如果在字符串中找到字符ch,则返回字符ch在字符串中第一次出现的位置;如果在整个字符串中没有找到字符ch,则返回-1.   
2)public int indexOf(int ch, int fromIndex)  
该方法和第一种方法类似,不同的地方在于,该方法从fromIndex位置向后查找,返回的仍然是字符ch在字符串第一次出现的位置.   
3)public int lastIndexOf(int ch)  
该方法和第一种方法类似,不同的地方在于,该方法从字符串的末尾位置向前查找,返回的仍然是字符ch在字符串第一次出现的位置.  
4)public int lastIndexOf(int ch, int fromIndex)  
该方法和第二种方法类似,不同的地方在于,该方法从fromIndex位置向前查找,返回的仍然是字符ch在字符串第一次出现的位置.  
(8**)字符串中子串的查找**  
字符串中子串的查找与字符串中单个字符的查找十分相似,可以利用String类提供的下列方法:  
1)public int indexOf(String str)  
2)public int indexOf(String str, int fromIndex)  
3)public int lastIndexOf(String str)  
4)public int lastIndexOf(String str, int fromIndex)   
(9)**字符串中字符大小写的转换**  
字符串中字符大小写的转换,可以利用String类提供的下列方法:  
1)public String toLowerCase()  
该方法将字符串中所有字符转换成小写,并返回转换后的新串.  
2)public String toUpperCase()  
该方法将字符串中所有字符转换成大写,并返回转换后的新串.   
(10)**字符串中多余空格的去除**  
public String trim()  
该方法只是去掉开头和结尾的空格,并返回得到的新字符串.值得注意的是,在原来字符串中间的空格并不去掉.   
(11)**字符串中字符的替换**  
1)public String replace(char oldChar,char newChar)  
该方法用字符newChar替换当前字符串中所有的字符oldChar,并返回一个新的字符串.  
2)public String replaceFirst(String regex, String replacement)  
该方法用字符串replacement的内容替换当前字符串中遇到的第一个和字符串regex相一致的子串,并将产生的新字符串返回.   
3)public String replaceAll(String regex, String replacement)  
该方法用字符串replacement的内容替换当前字符串中遇到的所有和字符串regex相一致的子串,并将产生的新字符串返回.   
**字符串变量与StringBuffer类**  
1.创建StringBuffer类对象  
StringBuffer类对象表示的是字符串变量,每一个StringBuffer类对象都是可以扩充和修改的字符串变量.以下是常用的StringBuffer类构造函数:  
(1)public StringBuffer()  
(2)public StringBuffer(int length)   
(3)public StringBuffer(String str)

1. Hash表的hash函数，冲突解决方法有哪些。

开放定址法或者叫再散列法；

1>线性探测再散列：冲突发生时，查看下个位置是否空，然后遍历下去找到个空的地方存放；

    2>二次探测再散列：冲突发生时，在表的左右进行跳跃探测，di=12 -12 22 -22....k2 -k2;

    3>伪随机探测再散列：di=伪随机序列；

再哈希法；

拉链法。

1. 各种排序：冒泡、选择、插入、希尔、归并、快排、堆排、桶排、基数的原理、平均时间复杂度、最坏时间复杂度、空间复杂度、是否稳定。

<https://blog.csdn.net/yushiyi6453/article/details/76407640>

<https://blog.csdn.net/wanglelelihuanhuan/article/details/51340290>



**冒泡排序原理：**

相连元素两两比较，大的往后放，第一次完毕后，最大值就出现在了最大索引处。同理，，继续，即可得到一个排好序的数组。

**选择排序原理：**

每一趟从待排序的记录中选出最小的元素，顺序放在已排好序的序列最后，直到全部记录排序完毕。也就是：每一趟在n-i+1(i=1，2，…n-1)个记录中选取关键字最小的记录作为有序序列中第i个记录。

**插入排序原理：**

它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

**归并排序的原理:**

从小到大排序：首先让数组中的每一个数单独成为长度为1的区间，然后两两一组有序合并，得到长度为2的有序区间，依次进行，直到合成整个区间。

**快速排序的原理:**

从小到大排序：在数组中随机选一个数（默认数组首个元素），数组中小于等于此数的放在左边，大于此数的放在右边，再对数组两边递归调用快速排序，重复这个过程。

**堆排序的原理:**

堆排序从小到大排序：首先将数组元素建成大小为n的大顶堆，堆顶（数组第一个元素）是所有元素中的最大值，将堆顶元素和数组最后一个元素进行交换，再将除了最后一个数的n-1个元素建立成大顶堆，再将最大元素和数组倒数第二个元素进行交换，重复直至堆大小减为1。

**希尔排序的原理:**

希尔排序是插入排序改良的算法，希尔排序**步长**从大到小调整，第一次循环后面元素逐个和前面元素按间隔步长进行比较并交换，直至步长为1，步长选择是关键。

**桶排序的原理:**

桶排序是计数排序的变种，把计数排序中相邻的m个”小桶”放到一个”大桶”中，在分完桶后，对每个桶进行排序（一般用快排），然后合并成最后的结果。

1. 快排的partition函数与归并的Merge函数。

**partition函数**：双向扫描

**Merge函数**:最后再看每一组（一对）子表的归并，其原理是相同的，只是子表表长不同，换句话说，是子表的首记录号与尾记录号不同，把这个归并操作作为核心算法写成函数 merge

1. **对冒泡与快排的改进。\***

8.1 对冒泡的改进

**改进1**：设置一个**标志位**，标志位代表在某一个冒泡遍历时候是否发生位置数据的交换，如果没有交换，则表明序列已经排序完成，否则继续排序。减少不必要的遍历。   
    **改进2**：再设置一个标志位，标志位是序列的某个**下标**，下标之后的代表已经排序完成，下标之前未排序，则遍历大于标志位时，不再遍历。减少一次遍历中已排完序的序列的遍历   
    **改进3**：在一次遍历时，**同时找出最大值和最小值**，从而提高效率。   
参考：[排序算法（一）——冒泡排序及改进](http://blog.csdn.net/u012152619/article/details/47305859)

8.2对快排的改进

基准的选取影响快排的效率，一般基准的选取有三种：   
    1）**固定位置**。选序列**第一位或者最后一位**，算法的导论中提到的就是固定选择最后一位。   
    2）**随机选取**。对于序列中部分有序的情况，如果选择固定位置作为基准，会导致全序列都需要交换位置，这会使得效率低下。因此会采用**随机选取数据作为基准**。   
    3）**三数取中**。最佳划分是将序列划分成等长的两个子序列，因此**提出三数取中**的思想。取序列中，下标第一位，下标中间一位，下标最后一位的三个数进行排序，取排序结果中排中间的数据作为基准。（此外，也可以取5个数作为数据的基准。）   
参考：[三种快速排序以及快速排序的优化](http://blog.csdn.net/insistgogo/article/details/7785038)  
    针对以上三种情况中，**三数取中效果最优**，但是依然无法解决序列中出现重复情况，对此进行再次优化：   
    **优化1**：当待排序序列的长度分割到一定大小后，使用**插入排序**。对于很小和部分有序的数组，快排不如插排好。   
    **优化2**：**与基准值相同的不加入分割**。在每一次分割结束后，可以把与基准相等的元素聚在一起，继续下次分割时，不用再对与基准相等元素分割。减少重复序列的反复分割   
    **优化3**：**优化递归操作**，快排函数在函数尾部有两次递归操作，我们可以对其使用**尾递归**优化。如果待排序的序列划分极端不平衡，递归的深度将趋近于n，而栈的大小是很有限的，每次递归调用都会耗费一定的栈空间，函数的参数越多，每次递归耗费的空间也越多。优化后，可以缩减堆栈深度，由原来的O(n)缩减为O(logn)，将会提高性能。   
    这里提一下尾递归，如果一个函数中所有递归形式的调用都出现**在函数的末尾**，我们称这个递归函数是尾递归。需要说明的是递归调用必须整个函数体中最后执行的语句且它的返回值不属于表达式的一部分。   
**尾递归的优点**：   
    1）尾递归通过迭代的方式，不存在子问题被多次计算的情况   
    2）尾递归的调用发生在方法的末尾，在计算过程中，完全可以把上一次留在堆栈的状态擦掉，保证程序以O(1)的空间复杂度运行。   
    **可惜的是，在jvm中第二点并没有被优化。**

1. 二分查找，与变种二分查找。

二分查找的中间下标：mid=low+0.5∗(high−low)mid=low+0.5∗(high−low)   
    **二分+插值**：   
    如果序列长度为1000，查找的关键字在10位置上，则还是需要从500中间开始二分查找，这样会产生多次无效查询，因此优化的方式就是更改分割的比例，采用三分，四分，分割位置：mid′=low+(high−low)∗(key−a[low])/(a[high]−key)mid′=low+(high−low)∗(key−a[low])/(a[high]−key)   
    插值查找是根据**要查找的关键字的key**与查找**表中最大最小记录的关键字**比较之后的查找算法。   
    **黄金分割比**：用黄金分割比来作为mid值

1. 二叉树、B+树、AVL树、红黑树、哈夫曼树。

<https://blog.csdn.net/xwg778899123/article/details/74157453>

二叉树：

二叉树的数据结构就不多说了，这里列举一些常见题目   
1）求解二叉树的节点   
    递归求解：   
        a) 树为空，节点数为0   
        b) 二叉树节点个数 = 左子树节点个数 + 右子树节点个数 + 1   
2）求二叉树的深度   
    递归解法：   
        a）如果二叉树为空，二叉树的深度为0   
        b）如果二叉树不为空，二叉树的深度 = max(左子树深度， 右子树深度) + 1   
3) 先根遍历，中序遍历，后序遍历   
    依然递归求解   
4）广度优先   
    借助队列。   
5）将二叉查找树变为有序的双向链表   
    要求不能创建新节点，只调整指针。   
    递归解法：   
        a）如果二叉树查找树为空，对应双向链表的第一个节点和最后一个节点是NULL   
        b）如果二叉查找树不为空：   
        设置参数flag，代表父节点与子节点的关系。如果修正的是左子树与父节点的关系，则递归返回的是序列最后的节点。   
6)求二叉树第K层的节点个数   
    递归解法：   
        a）如果二叉树为空或者k<1返回0   
        b）如果二叉树不为空并且k==1，返回1   
        c）如果二叉树不为空且k>1，返回左子树中k-1层的节点个数与右子树k-1层节点个数之和   
7)求二叉树中叶子节点的个数   
    递归解法：   
        a）如果二叉树为空，返回0   
        b）如果二叉树不为空且左右子树为空，返回1   
        c）如果二叉树不为空，且左右子树不同时为空，返回左子树中叶子节点个数加上右子树中叶子节点个数   
8)判断二叉树是不是平衡二叉树(AVL树)   
    递归解法：   
        a）如果二叉树为空，返回真   
        b）如果二叉树不为空，如果左子树和右子树都是AVL树并且左子树和右子树高度相差不大于1，返回真，其他返回假   
9)由前序遍历序列和中序遍历序列重建二叉树   
    二叉树前序遍历序列中，第一个元素总是树的根节点的值。中序遍历序列中，左子树的节点的值位于根节点的值的左边，右子树的节点的值位于根节点的值的右边。   
    递归解法：   
        a）如果前序遍历为空或中序遍历为空或节点个数小于等于0，返回NULL;   
        b）创建根节点。前序遍历的第一个数据就是根节点的数据，在中序遍历中找到根节点的位置，可分别得知左子树和右子树的前序和中序遍历序列，重建左右子树   
10)判断是不是完全二叉树

1. 二叉树的前中后续遍历：递归与非递归写法，层序遍历算法。
2. 图的BFS与DFS算法，最小生成树prim算法与最短路径Dijkstra算法。
3. KMP算法。
4. 排列组合问题。
5. 动态规划、贪心算法、分治算法。（一般不会问到）
6. 大数据处理：类似10亿条数据找出最大的1000个数.........等等
7. 算法的话其实是个重点，因为最后都是要你写代码，所以算法还是需要花不少时间准备，这里有太多算法题，写不全，我的建议是没事多在OJ上刷刷题（牛客网、leetcode等），剑指offer上的算法要能理解并自己写出来，编程之美也推荐看一看

**数据库**

<https://my.oschina.net/yanpenglei/blog/1650277>

1. **事务四大特性（ACID）原子性、一致性、隔离性、持久性\***

**原子性**：是指事务是一个不可再分割的工作单位，**事务**中的操作**要么都发生，要么都不发生**。

**一致性:** 是指在**事务开始之前和事务结束以后**，数据库的完整性约束没有被破坏。这是说数据库事务不能破坏**关系数据的完整性以及业务逻辑**上的一致性。

**隔离性**：多个事务并发访问时，事务之间是**隔离的**，一个事务不应该影响**其它事务运行效果**。

事务之间的相互影响：脏读，不可重复读，幻读，丢失更新。

**脏读** 意味着一个事务读取了另一个事务未提交的数据，而这个数据是有可能回滚的

**不可重复读** 意味着，在数据库访问中，一个事务范围内两个相同的查询却返回了不同数据。这是由于查询时系统中其他事务修改的提交而引起的。

**幻读** 是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象发生了幻觉一样.

丢失更新 两个事务同时读取同一条记录，A先修改记录，B也修改记录（B是不知道A修改过），B提交数据后B的修改结果覆盖了A的修改结果。

**持久性：**

意味着在事务完成以后，该事务所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。

1. **数据库隔离级别，每个级别会引发什么问题，mysql默认是哪个级别\***



默认已提交读。

1. **innodb和myisam存储引擎的区别\***

<https://www.cnblogs.com/kevingrace/p/5685355.html>

1）**InnoDB支持事务，MyISAM不支持**，这一点是非常之重要。事务是一种高级的处理方式，如在一些列增删改中只要哪个出错还可以回滚还原，而MyISAM就不可以了。  
2）**MyISAM适合查询**以及**插入为**主的应用，**InnoDB适合频繁修改**以及涉及到安全性较高的应用  
3）InnoDB支持**外键**，MyISAM不支持  
4）从MySQL5.5.5以后，**InnoDB是默认引擎**  
5）InnoDB不支持FULLTEXT类型的索引  
6）InnoDB中不保存表的行数，如select count(\*) from table时，InnoDB需要扫描一遍整个表来计算有多少行，但是MyISAM只要简单的读出保存好的行数即可。注意的是，当count(\*)语句包含where条件时MyISAM也需要扫描整个表  
7）对于自增长的字段，InnoDB中必须包含只有该字段的索引，但是在MyISAM表中可以和其他字段一起建立联合索引  
8）清空整个表时，InnoDB是一行一行的删除，效率非常慢。MyISAM则会重建表  
9）InnoDB支持行锁（某些情况下还是锁整表，如 update table set a=1 where user like '%lee%'

1. MYSQL的两种存储引擎区别（事务、锁级别等等），各自的适用场景

现在一般都是选用innodb了，主要是myisam的全表锁，读写串行问题，并发效率锁表，效率低myisam对于读写密集型应用一般是不会去选用的。

**MyISAM是表锁，InnoDB是行锁。**

1. 查询语句不同元素（where、jion、limit、group by、having等等）执行先后顺序

**Join where groupby having limit**

1. **数据库的优化（从sql语句优化和索引两个部分回答）\***

<https://www.jb51.net/article/107054.htm>

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

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**

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。

应尽量避免在 **where 子句中使用 or 来连接条件**，如果一个字段**有索引**，一个字段**没有索引**，将导致引擎放弃使用索引而进行全表扫描

**in 和 not in** 也要慎用，否则会导致全表扫描

下面的查询也将导致全表扫描：

|  |  |
| --- | --- |
| 1 | select id from t where name like ‘%abc%' |

如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描

1. 索引有B+索引和hash索引，各自的区别

**B+树是一个平衡的多叉树，从根节点到每个叶子节点的高度差值不超过1，而且同层级的节点间有指针相互链接。**

在B+树上的常规检索，从根节点到叶子节点的搜索效率基本相当，不会出现大幅波动，而且基于索引的顺序扫描时，也可以利用双向指针快速左右移动，效率非常高。

**哈希索引就是采用一定的哈希算法**，把键值换算成新的哈希值，检索时不需要类似B+树那样从根节点到叶子节点逐级查找，只需一**次哈希算法即可立刻定位到相应的位置**，速度非常快。

1. B+索引数据结构，和B树的区别

**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+**树是对B树的一种变形树，它与B树的差异在于：

* 有k个子结点的结点必然有k个关键码；
* 非叶结点仅具有索引作用，跟记录有关的信息均存放在叶结点中。
* 树的所有叶结点构成一个有序链表，可以按照关键码排序的次序遍历全部记录。

1. **索引的分类（主键索引、唯一索引），最左前缀原则，哪些情况索引会失效\***

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。索引不会包含有null值的列，索引项可以为null（唯一索引、组合索引等），但是只要列中有null值就不会被包含在索引中。

    （1）普通索引：create index index\_name on table(column)；

    或者创建表时指定，create table(..., index index\_name column);

    （2）唯一索引：类似普通索引，索引列的值必须唯一（可以为空，这点和主键索引不同）

    create unique index index\_name on table(column)；或者创建表时指定unique index\_name column

    （3）主键索引：特殊的唯一索引，不允许为空，只能有一个，一般是在建表时指定primary key(column)

   （4）组合索引：在多个字段上创建索引，遵循最左前缀原则。alter table t add index index\_name(a,b,c);

   （5）全文索引：主要用来查找文本中的关键字，不是直接与索引中的值相比较，像是一个搜索引擎，配合match against使用，现在只有char，varchar，text上可以创建全文索引。在数据量较大时，**先将数据放在一张没有全文索引的表里，然后再利用create index创建全文索引，比先生成全文索引再插入数据快很多**。

    （1）主键，unique字段；

    （2）和其他表做连接的字段需要加索引；

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

    （4）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；

3、何时不使用索引

    （1）表记录太少；

    （2）数据重复且分布平均的字段（只有很少数据值的列）；

    （3）经常插入、删除、修改的表要减少索引；

    （4）text，image等类型不应该建立索引，这些列的数据量大（假如text前10个字符唯一，也可以对text前10个字符建立索引）；

    （5）MySQL能估计出全表扫描比使用索引更快时，不使用索引；

4、索引何时失效

    （1）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；

    （2）like未使用最左前缀，where A like '%China'；

    （3）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 ；

    （4）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；

1. 聚集索引和非聚集索引区别。

其中**聚集索引**表示表中存储的数据**按照索引的顺序存储**，检索效率比非聚集索引高，但**对数据更新影响较大**。**非聚集索引**表示数据存储在一个地方，索引存储在另一个地方，索引带有**指针指向数据的存储位置**，非聚集索引检索效率比聚集索引低，但对**数据更新影响较小**。

1. **有哪些锁（乐观锁悲观锁），select时怎么加排它锁\***

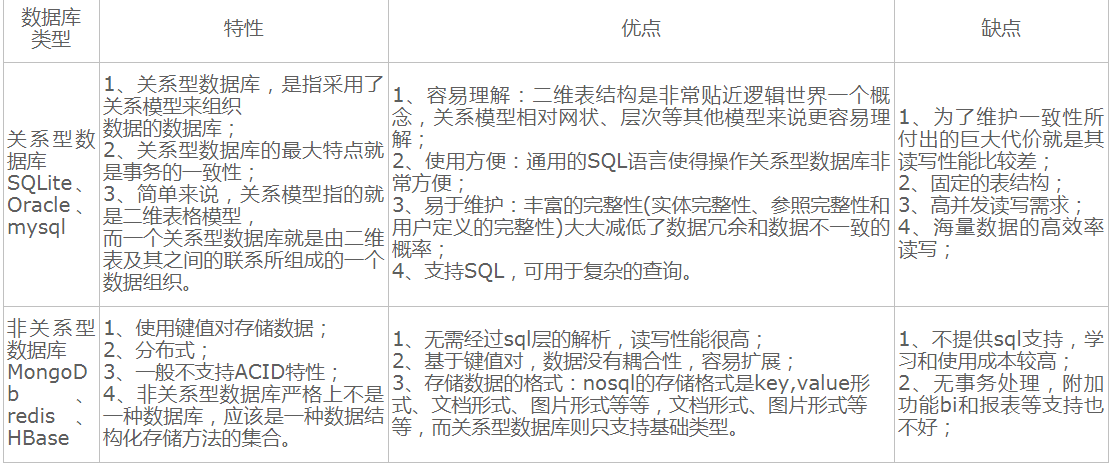
**乐观锁不是数据库自带的，需要我们自己去实现**。乐观锁是指操作数据库时(更新操作)，想法很乐观，认为这次的操作不会导致冲突，在**操作数据时，并不进行任何其他的特殊处理**（也就是不加锁），**而在进行更新后，再去判断是否有冲突**了。

与乐观锁相对应的就是悲观锁了。悲观锁就是在操作数据时，认为此操作会出现数据冲突，所以在进行**每次操作时都要通过获取锁才能进行对相同数据的操作**，这点跟java中的synchronized很相似，所以悲观锁需要耗费较多的时间。另外与乐观锁相对应的，**悲观锁是由数据库自己实现了的**，要用的时候，我们直接调用数据库的相关语句就可以了。

用法： select … for update;

1. 关系型数据库和非关系型数据库区别

1.**关系型数据库**通过**外键**关联来建立**表与表**之间的关系，  
2.**非关系型数据库**通常指数据以**对象的形式存储在数据库**中，而对象之间的关系通过每个对象自身的属性来决定



1. 数据库三范式，根据某个场景设计数据表（可以通过手绘ER图）

目前关系数据库有六种范式：**第一范式（1NF）、第二范式（2NF）、第三范式（3NF）、巴斯-科德范式（BCNF）、第四范式(4NF）和第五范式（5NF，又称完美范式）**。   
**第一范式（1NF）**：是指在关系模型中，对域添加的一个规范要求，**所有的域都应该是原子性的**，即数据库表的每一列都是不可分割的原子数据项，而不能是集合，数组，记录等非原子数据项。即实体中的某个属性有多个值时，必须拆分为不同的属性。在符合第一范式（1NF）表中的每个域值只能是实体的一个属性或一个属性的一部分。简而言之，**第一范式就是无重复的域**。   
**第二范式（2NF）**：在1NF的基础上，非码属性必须完全依赖于候选码（在1NF基础上消除非主属性对主码的部分函数依赖），要求实体的属性完全依赖于主关键字。所谓完全依赖是指不能存在仅依赖主关键字一部分的属性，如果存在，那么这个属性和主关键字的这一部分应该分离出来形成一个新的实体，新实体与原实体之间是一对多的关系。为实现区分通常需要为表加上一个列，以存储各个实例的唯一标识。简而言之，第二范式就是在**第一范式的基础上属性完全依赖于主键。**第三范式（3NF）：在2NF基础上，任何非主属性不依赖于其它非主属性（在2NF基础上消除传递依赖）。第三范式（3NF）是第二范式（2NF）的一个子集，即满足第三范式（3NF）必须满足第二范式（2NF）。简而言之，**第三范式（3NF）要求一个关系中不包含已在其它关系已包含的非主关键字信息。**

1. **数据库的读写分离、主从复制\***

<https://blog.csdn.net/kisscatforever/article/details/78776133>

可以通过**logbin文件**进行主从复制，修改配置文件

通过设置**主从数据库**实现读写分离，**主数据库负责“写操作”，从数据库负责“读操作”**，根据压力情况，从数据库可以部署多个提高“读”的速度，借此来提高系统总体的性能。

1. 使用explain优化sql和索引

表的读取顺序

数据读取操作的操作类型

哪些索引可以使用

哪些索引被实际使用

表之间的引用

每张表有多少行被优化器查询  
说了这么多使用explain的好处,那么实际上到底该怎么玩? 答案： **explain + 待执行的sql**

对于复杂、效率低的sql语句，我们通常是使用explain sql 来分析sql语句，这个语句可以打印出，语句的执行。这样方便我们分析，进行优化

**table**：显示这一行的数据是关于哪张表的

**type**：这是重要的列，显示连接使用了何种类型。从最好到最差的连接类型为const、eq\_reg、ref、range、index和ALL

**all**: full table scan ;MySQL将遍历全表以找到匹配的行；

**index** ： index scan; index 和 all的区别在于index类型只遍历索引；

**range**：索引范围扫描，对索引的扫描开始于某一点，返回匹配值的行，常见与between ，< ,>等查询；

**ref**：非唯一性索引扫描，返回匹配某个单独值的所有行，常见于使用非唯一索引即唯一索引的非唯一前缀进行查找；

**eq\_ref**：唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配，常用于主键或者唯一索引扫描；

**const**，system：当MySQL对某查询某部分进行优化，并转为一个常量时，使用这些访问类型。如果将主键置于where列表中，MySQL就能将该查询转化为一个常量。

**possible\_keys**：显示可能应用在这张表中的索引。如果为空，没有可能的索引。可以为相关的域从WHERE语句中选择一个合适的语句

**key**： **实际使用的索引。如果为NULL，则没有使用索引**。很少的情况下，MySQL会选择优化不足的索引。这种情况下，可以在SELECT语句中使用USE INDEX（indexname）来强制使用一个索引或者用IGNORE INDEX（indexname）来强制MySQL忽略索引

**key\_len**：**使用的索引的长度。在不损失精确性的情况下，长度越短越好**

**ref**：显示索引的哪一列被使用了，如果可能的话，是一个常数

**rows**：MySQL认为必须检查的用来返回请求数据的行数

**Extra**：关于MySQL如何解析查询的额外信息。将在表4.3中讨论，但这里可以看到的坏的例子是Using temporary和Using filesort，意思MySQL根本不能使用索引，结果是检索会很慢。

1. **long\_query怎么解决\***

**慢查询日志：**默认情况下，MySQL数据库是不开启慢查询日志的，long\_query\_time的默认值为10（即10秒，通常设置为1秒），即运行10秒以上的语句是慢查询语句。

log-slow-queries  ：旧版（5.6以下版本）MySQL数据库慢查询日志存储路径。可以不设置该参数，系统则会默认给一个缺省的文件host\_name-slow.log

slow-query-log-file：新版（5.6及以上版本）MySQL数据库慢查询日志存储路径。可以不设置该参数，系统则会默认给一个缺省的文件host\_name-slow.log

log\_queries\_not\_using\_indexes：未使用索引的查询也被记录到慢查询日志中（可选项）。

**slow\_query\_log**慢查询开启状态。

**slow\_query\_log\_file**慢查询日志存放的位置（这个目录需要MySQL的运行帐号的可写权限，一般设置为MySQL的数据存放目录）。

**long\_query\_time** 查询超过多少秒才记录。

1. 内连接、外连接、交叉连接、笛卡儿积等

**内连接(INNER JOIN)：**  
    分为三种：等值连接、自然连接、不等连接

**外连接(OUTER JOIN)：**    
    分为三种：    
    左外连接(LEFT OUTER JOIN或LEFT JOIN)    
    右外连接(RIGHT OUTER JOIN或RIGHT JOIN)    
    全外连接(FULL OUTER JOIN或FULL JOIN)

**交叉连接(CROSS JOIN)：**  
    没有WHERE 子句，它返回连接表中所有数据行的笛卡尔积

笛卡尔积是两个表每一个字段相互匹配，去掉where 或者inner join的等值 得出的结果就是笛卡尔积。**笛卡尔积也等同于交叉连接**。

**内连接:** 只连接匹配的行。

**左外连接:** 包含左边表的全部行（不管右边的表中是否存在与它们匹配的行），以及右边表中全部匹配的行。

**右外连接:** 包含右边表的全部行（不管左边的表中是否存在与它们匹配的行），以及左边表中全部匹配的行。

**全外连接:** 包含左、右两个表的全部行，不管另外一边的表中是否存在与它们匹配的行。

交叉连接 生成笛卡尔积－它不使用任何匹配或者选取条件，而是直接将一个数据源中的每个行与另一个数据源的每个行都一一匹配

1. **死锁判定原理和具体场景，死锁怎么解决\***

<https://blog.csdn.net/XiaHeShun/article/details/81393796>

数据库是一个多用户使用的共享资源，当多个用户并发地存取数据的时候，在数据库中就会发生多个事务同时存取同一个数据的情况，**加锁是进行数据库并发控制**的一种非常重要的技术。在实际应用中，如果两个事务需要一组有冲突的锁，而不能继续进行下去，这时便发生了死锁。

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

**什么情况下会造成死锁**

所谓死锁<DeadLock>: 是指两个或两个以上的进程在执行过程中。

**因争夺资源而造成的一种互相等待的现象,若无外力作用,它们都将无法推进下去。**

此时称系统处于死锁状态或系统产生了死锁,这些永远在互相等的进程称为死锁进程。

表级锁不会产生死锁.所以解决死锁主要还是针对于最常用的**InnoDB**。

死锁的关键在于：**两个(或以上)的Session加锁**的顺序不一致。

那么对应的**解决死锁问题的关键就是**：让不同的session**加锁有次序**。

**死锁的解决办法**

**查出的线程杀死 kill**

**设置锁的超时时间**

1. **varchar和char的使用场景。\***

**char的长度是不可变的，而varchar的长度是可变的**。

varchar是**以空间效率为首位**。

**char的存储方式是**：对英文字符（ASCII）占用1个字节，对一个汉字占用两个字节。

**varchar的存储方式是**：对每个英文字符占用2个字节，汉字也占用2个字节。

**两者的存储数据都非unicode的字符数据**。

1. mysql并发情况下怎么解决（通过事务、隔离级别、锁）

MySQL 高并发环境解决方案 分库 分表 分布式 增加二级缓存。。。。。

**需求分析**：互联网单位 每天大量数据读取，写入，并发性高。

**现有解决方式**：水平分库分表，由单点分布到多点数据库中，从而降低单点数据库压力。

**集群方案**：解决DB宕机带来的单点DB不能访问问题。

**读写分离策略**：极大限度提高了应用中Read数据的速度和并发量。无法解决高写入压力。

1. 数据库崩溃时事务的恢复机制（REDO日志和UNDO日志）

**Undo Log**

**Undo Log是为了实现事务的原子性**，在MySQL数据库InnoDB存储引擎中，还用了UndoLog来实现多版本并发控制(简称：MVCC)。

**事务的原子性**(Atomicity)事务中的所有操作，要么全部完成，要么不做任何操作，不能只做部分操作。如果在执行的过程中发生了错误，要回滚(Rollback)到事务开始前的状态，就像这个事务从来没有执行过。

**原理**Undo Log的原理很简单，**为了满足事务的原子性，在操作任何数据之前，首先将数据备份到一个地方（这个存储数据备份的地方称为UndoLog）**。然后进行数据的修改。如果出现了错误或者用户执行了ROLLBACK语句，**系统可以利用Undo Log中的备份将数据恢复到事务开始之前的状态**。

**之所以能同时保证原子性和持久化，是因为以下特点**：

更新数据前记录Undo log。

为了保证持久性，必须将数据在事务提交前写到磁盘。只要事务成功提交，数据必然已经持久化。

**Undo log必须先于数据持久化到磁盘**。如果在G,H之间系统崩溃，**undo log是完整的， 可以用来回滚事务**。

如果在A-F之间系统崩溃,因为数据没有持久化到磁盘。所以磁盘上的数据还是保持在事务开始前的状态。

**缺陷**：**每个事务提交前将数据和Undo Log写入磁盘，这样会导致大量的磁盘IO，因此性能很低**。

如果能够将数据**缓存一段时间**，就能减少IO提高性能。但是这样就会丧失事务的持久性。因此引入了另外一种机制来实现持久化，即Redo Log。

**Redo Log**

**原理和Undo Log相反**，**Redo Log记录的是新数据的备份**。**在事务提交前，只要将Redo Log持久化即可，不需要将数据持久化。当系统崩溃时，虽然数据没有持久化，但是Redo Log已经持久化。系统可以根据Redo Log的内容，将所有数据恢复到最新的状态**。

1. 查询语句不同元素（where、jion、limit、group by、having等等）执行先后顺序

**Join where limit group by having**

**Spring**

1. IOC和DI是什么？

IOC---Inversion of Control(控制反转)，IOC意味着将你设计好的对象交给容器控制，而不是传统的在你对象内部直接控制。

DI---Dependency Injection(依赖注入)：是组件之间依赖关系由容器在运行期决定。

1. **Spring IOC 的理解，其初始化过程？\***

Spring IOC的核心是BeanFactory

**定位并获取资源文件：通过配置文件获取资源对象**

ClassPathResource res = new ClassPathResource("my/applicationContext.xml");

**解析资源文件 Bean的载入和解析：对资源进行解析，获取bean对象**

**向IoC容器注册BeanDefinition：利用解析好的BeanDefinition对象完成最终的注册，将beanName和beanDefinition作为键值 放到了beanFactorty的map中**

1. **BeanFactory 和 FactoryBean的区别？\***

**BeanFactory**是接口，提供了OC容器最基本的形式，给具体的IOC容器的实现提供了规范，

**FactoryBean**也是接口，为IOC容器中Bean的实现提供了更加灵活的方式，FactoryBean在IOC容器的基础上给Bean的实现加上了一个**简单工厂模式和装饰模式**(如果想了解装饰模式参考：[修饰者模式(装饰者模式，Decoration)](https://www.cnblogs.com/aspirant/p/9083082.html)我们可以在getObject()方法中灵活配置。其实在Spring源码中有很多FactoryBean的实现类.

**BeanFactory**

以Factory结尾，表示它是一个工厂类(接口)，它负责生产和管理bean的一个工厂。在Spring中，BeanFactory是IOC容器的核心接口，它的职责包括：实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。BeanFactory只是个接口，并不是IOC容器的具体实现，但是Spring容器给出了很多种实现，如 DefaultListableBeanFactory、XmlBeanFactory、ApplicationContext等，其中XmlBeanFactory就是常用的一个，该实现将以XML方式描述组成应用的对象及对象间的依赖关系。

**FactoryBean**

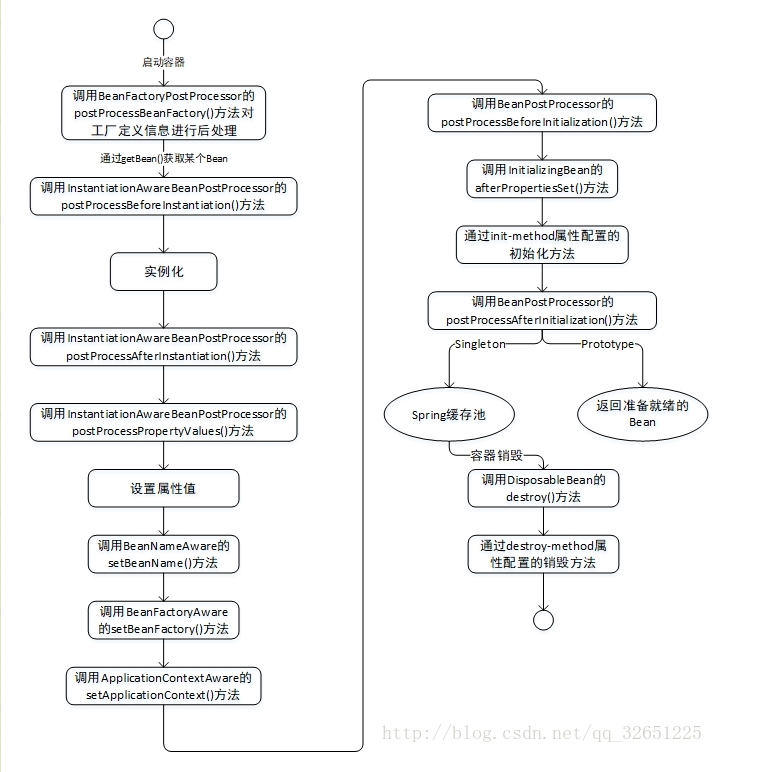
一般情况下，**Spring通过反射机制利用<bean>的class属性指定实现类实例化Bean**，在某些情况下，实例化Bean过程比较复杂，如果按照传统的方式，则需要在<bean>中提供大量的配置信息。配置方式的灵活性是受限的，这时采用编码的方式可能会得到一个简单的方案。Spring为此提供了一个org.springframework.bean.factory.FactoryBean的工厂类接口，用户可以通过实现该接口定制实例化Bean的逻辑。FactoryBean接口对于Spring框架来说占用重要的地位，Spring自身就提供了70多个FactoryBean的实现。

1. **BeanFactory和ApplicationContext的区别？\***

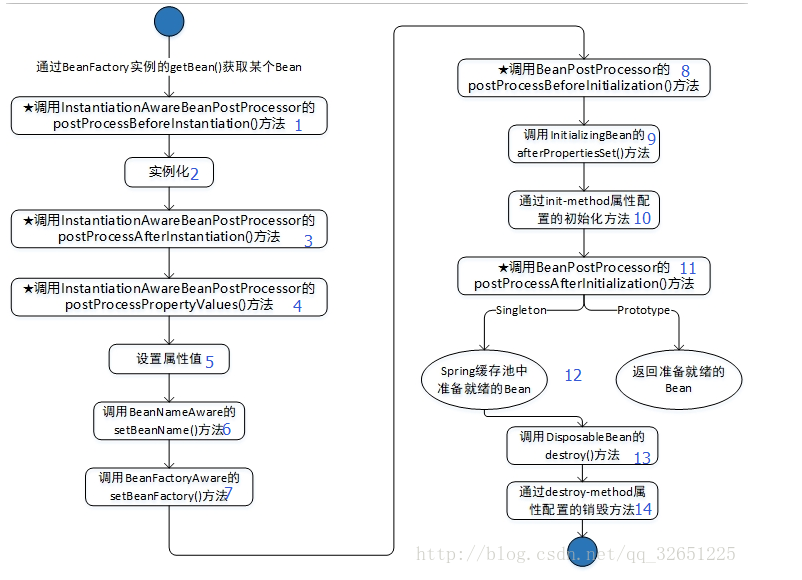
BeanFacotry是spring中比较原始的Factory。如XMLBeanFactory就是一种典型的BeanFactory。原始的BeanFactory无法支持spring的许多插件，如AOP功能、Web应用等。   
ApplicationContext接口,它由BeanFactory接口派生而来，因而提供BeanFactory所有的功能。ApplicationContext以一种更向面向框架的方式工作以及对上下文进行分层和实现继承，ApplicationContext包还提供了以下的功能：   
  • MessageSource, 提供国际化的消息访问    
  • 资源访问，如URL和文件  ResourceLoader  
  • 事件传播ApplicationEvent和ApplicationListener  
  • 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层

1. **ApplicationContext 上下文的生命周期？\***

<https://blog.csdn.net/qq_32651225/article/details/78323527>



1. **Spring Bean 的生命周期？\***



1. Spring AOP的实现原理？

**AOP，面向切面。**AOP技术利用一种称为“横切”的技术，解剖封装的对象内部，并将那些影响了多个类的公共行为封装到一个**可重用模块**，这样就能减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。AOP把软件系统分为两个部分：**核心关注点和横切关注点**。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。横切关注点的一个特点是，他们经常发生在核心关注点的多处，而各处都基本相似。比如**权限认证、日志、事务处理。**

1. **Spring 是如何管理事务的，事务管理机制？\***

Spring的事务机制包括**声明式事务**和**编程式事务**。

**编程式事务管理：**Spring推荐使用TransactionTemplate，实际开发中使用声明式事务较多。

**声明式事务管理：**将我们从复杂的事务处理中解脱出来，**获取连接，关闭连接、事务提交、回滚、异常处理**等这些操作都不用我们处理了，Spring都会帮我们处理。

声明式事务管理使用了AOP面向切面编程实现的，本质就是在目标方法**执行前后进行拦截**。在目标方法执行前加入或创建一个事务，在执行方法执行后，根据实际情况选择提交或是回滚事务。

**如何管理的：**

Spring事务管理主要包括3个接口，Spring的事务主要是由他们三个共同完成的。

1）**PlatformTransactionManager**：事务管理器--主要用于平台相关事务的管理

2）**TransactionDefinition**：事务定义信息--用来定义事务相关的属性，给事务管理器PlatformTransactionManager使用

3）**TransactionStatus**：事务具体运行状态--事务管理过程中，每个时间点事务的状态信息。

1. **Spring 的不同事务传播行为有哪些，干什么用的？\***

1、**PROPAGATION\_REQUIRED**：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

2、**PROPAGATION\_SUPPORTS**：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。‘

3、**PROPAGATION\_MANDATORY**：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

4、**PROPAGATION\_REQUIRES\_NEW**：创建新事务，无论当前存不存在事务，都创建新事务。

5、**PROPAGATION\_NOT\_SUPPORTED**：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

6、**PROPAGATION\_NEVER**：以非事务方式执行，如果当前存在事务，则抛出异常。

7、**PROPAGATION\_NESTED**：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与PROPAGATION\_REQUIRED类似的操作。

1. **Spring 中用到了那些设计模式？\***

**工厂模式：**在各种BeanFactory以及ApplicationContext创建中都用到了；

**单例模式:** 这个比如在创建bean的时候。

**适配器（Adapter）:** 在Spring的Aop中，使用的Advice（通知）来增强被代理类的功能。

**包装器:** 动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator模式相比生成子类更为灵活。

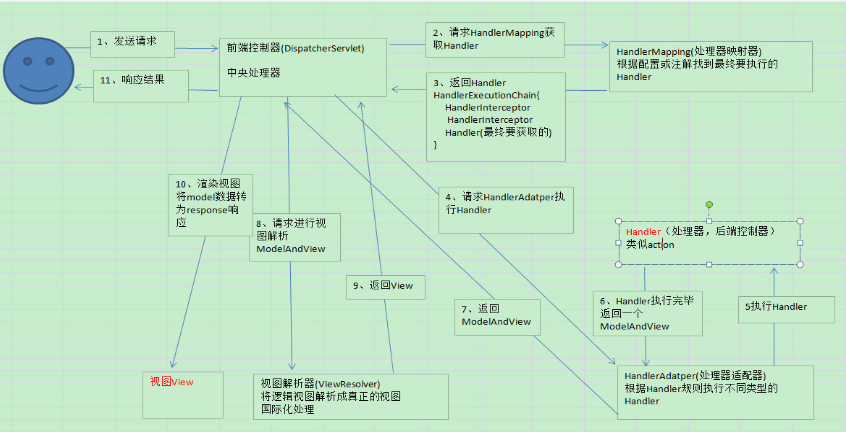
**代理（Proxy）:** Spring的Proxy模式在aop中有体现，比如JdkDynamicAopProxy和Cglib2AopProxy。

**观察者（Observer）:** Spring中Observer模式常用的地方是listener的实现。如ApplicationListener。

**策略（Strategy）:** 加载资源文件的方式，使用了不同的方法

**模板方法:** Spring中的JdbcTemplate

1. **Spring MVC 的工作原理？\***



1. Spring如何解决循环依赖？

**Spring的循环依赖**的理论依据其实是基于Java的引用传递，当我们获取到**对象的引用**时，**对象的field**或则**属性**是可以延后设置的(但是构造器必须是在获取引用之前)。  
Spring的单例对象的初始化主要分为三步：  
（1）**createBeanInstance**：实例化，其实也就是调用对象的构造方法实例化对象  
（2）**populateBean**：填充属性，这一步主要是多bean的依赖属性进行填充  
（3）**initializeBean**：调用spring xml中的init 方法。  
从上面讲述的单例bean初始化步骤我们可以知道，循环依赖主要发生在第一、第二步。也就是构造器循环依赖和field循环依赖。  
那么我们要解决循环引用也应该从初始化过程着手，对于单例来说，在Spring容器整个生命周期内，有且只有一个对象，所以很容易想到这个对象应该存在Cache中，Spring为了解决单例的循环依赖问题，使用了**三级缓存**。

1. **Spring 如何保证 Controller 并发的安全？\***

在Controller中使用ThreadLocal变量  
在spring配置文件Controller中声明 scope="prototype"，每次都创建新的controller

在控制器中不使用实例变量

**Netty**

1. **BIO、NIO和AIO\***

**BIO：** **同步并阻塞**，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理（**一客户端一线程**）。该模型最大的问题就是缺乏弹性伸缩能力，当客户端并发访问量增加后，服务端的线程数与客户端并发访问数呈1:1的关系，系统性能将急剧下降，随着**并发访问量的继续增加**，系统会发生**线程堆栈溢出、创建新线程失败**等问题，并最终导致宕机或僵死。

**NIO：异步非阻塞**，服务器实现模式为**一个请求一个线程**，客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有**I/O请求时才启动一个线程**进行处理。

**AIO：**JDK1.7升级了NIO库，升级后的NIO库被称为NIO2.0，正式引入了**异步通道**的概念。NIO2.0的**异步套接字**通道是真正的异步非阻塞I/O，此即AIO。其服务器实现模式为**一个有效请求一个线程**，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理。

1. **Netty 的各大组件\***

**Bootstrap or ServerBootstrap**

Bootstrap，一个Netty应用通常由一个Bootstrap开始，它主要作用是配置整个Netty程序，串联起各个组件。

**EventLoop**

一个 EventLoop 在它的生命周期内只能与一个Thread绑定。

一个 EventLoop 可被分配至一个或多个 Channel

**EventLoopGroup**

一个EventLoopGroup 包含多个EventLoop 可以理解为一个线程池

**ChannelPipeline**

channelHandler的容器，每个Channel会绑定一个ChannelPipeline，用于处理该Channel上的事件。

**Channel**

代表了一个socket连接，简化了socket进行操作的复杂性

**Future or ChannelFuture**

Netty 为异步非阻塞，即**所有的 I/O 操作都为异步的**，因此，我们不能立刻得知消息是否已经被处理了。Netty 提供了 **ChannelFuture 接口**，通过该接口的 **addListener()** 方法注册一个 ChannelFutureListener，当操作执行成功或者失败时，监听就会自动触发返回结果。**ChannelInitializer**

当一个链接建立时，我们需要知道怎么来接收或者发送数据，当然，我们有各种各样的Handler实现来处理它，那么ChannelInitializer便是用来**配置这些Handler**，它会提供一个ChannelPipeline，并把Handler加入到ChannelPipeline。

**ChannelHandler**

ChannelHandler 为 Netty 中最核心的组件，它充当了所有处理入站和出站数据的应用程序逻辑的容器。**ChannelHandler 主要用来处理各种事件**，这里的事件很广泛，比如可以是连接、数据接收、异常、数据转换等。

ChannelHandler 有两个核心子类 **ChannelInboundHandler** 和 **ChannelOutboundHandler**，其中 ChannelInboundHandler 用于接收、处理入站数据和事件，而 ChannelOutboundHandler 则相反。

1. **Netty的线程模型\***

<https://www.cnblogs.com/duanxz/p/3696849.html>

基于Reactor的**单线程模型、多线程模型、主从模型**

netty支持Reactor的单线程模型、多线程、主从Reactor多线程模型。

从原理图中可以看到，服务端启动时创建了两个NioEventLoopGroup，这是两个相互独立的Reactor线程池，一个是boss线程池，一个是work线程池。两个分工明确，一**个负责接收客户端连接请求，一个负责处理IO相关的读写操作，或者执行Task任务，或执行定时Task任务。**其中NioEventLoopGroup中的NioEventLoop个数默认为**处理器核数\*2**。

通过配置boss线程池和worker线程池的线程个数以及是否共享线程池，来配置单线程、多线程、主从Reactor多线程。

**Boss线程池的任务：**

a.接收客户端的连接请求，初始化channel参数

b.将链路状态变化时间通知给ChannelPipeline

**Worker线程池的作用：**

a.异步读取通信对端的消息，发送读事件到ChannelPipeline

b.异步发送消息到通信对端，调用ChannelPipeline的发送消息接口

c.执行系统Task任务

d.执行系统定时Task任务

**单线程模型：**

作为服务端，接收客户端的TCP连接；

作为客户端，向服务端发起TCP连接；

读取通信对端的请求或者应答消息；

向通信对端发送消息请求或者应答消息。

**多线程模型：**

专门由一个Reactor线程-Acceptor线程用于监听服务端，接收客户端连接请求；

网络I/O操作读、写等由Reactor线程池负责处理；

一个Reactor线程可同时处理多条链路，但一条链路只能对应一个Reactor线程，这样可避免并发操作问题。

**主从线程模型：**

服务端使用一个独立的主Reactor线程池来处理客户端连接，当服务端收到连接请求时，从主线程池中随机选择一个Reactor线程作为Acceptor线程处理连接；

链路建立成功后，将新创建的SocketChannel注册到sub reactor线程池的某个Reactor线程上，由它处理后续的I/O操作。

1. **TCP 粘包/拆包的原因及解决方法\***

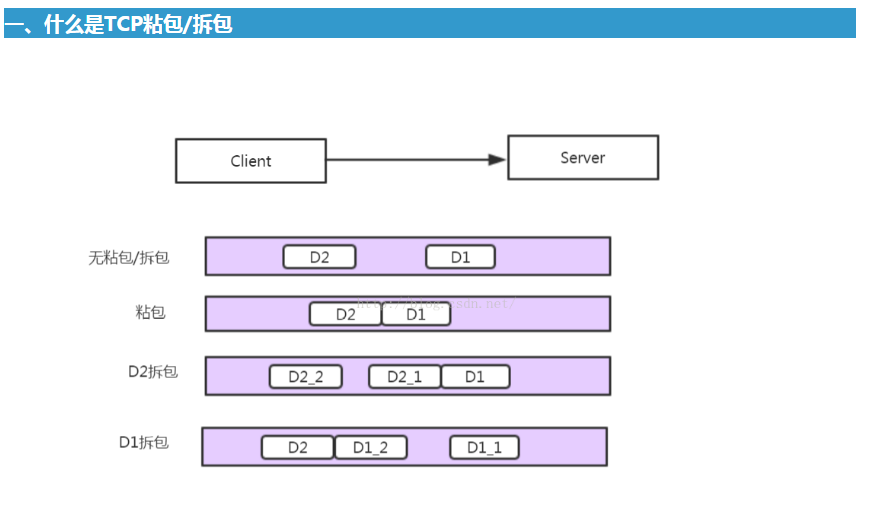
1**.消息定长**，例如每个报文的大小为固定长度200字节，如果不够，空位补空格。

2.在**包尾增加回车换行符进行分割**，例如FTP协议。

3.将消息分为**消息头和消息体**，消息头中包含消息长度的字段，通常设计思路为消息头的第一个字段使用int32来表示消息的总长度

**Uhost通过自定义消息协议来编码和解码，通过定义消息的消息msg，消息类型+消息数据**

**通过编码—8个字节，4个字节存放消息类型，4个字节存放消息长度，后面存放byte[]消息体。**



1. **了解哪几种序列化协议？包括使用场景和如何去选择\***

**序列化（serialization）**就是将对象序列化为二进制形式（字节数组），一般也将序列化称为**编码（Encode）**，主要用于网络传输、数据持久化等；

**反序列化（deserialization）**则是将从网络、磁盘等读取的字节数组还原成原始对象，以便后续业务的进行，一般也将反序列化称为**解码（Decode）**，主要用于网络传输对象的解码，以便完成远程调用。

**XML**: 当做配置文件存储数据，实时数据转换

**JSON**: 轻量级的数据交换格式，简洁和清晰，跨防火墙访问；可调式性要求高的情况；基于Web browser的Ajax请求；传输数据量相对小，实时性要求相对低（例如秒级别）的服务

**Fastjson**: Fastjson是一个Java语言编写的高性能功能完善的JSON库协议交互

Web输出

Android客户端

**Thrift:** 并不仅仅是序列化协议，而是一个RPC框架。它可以让你选择客户端与服务端之间传输通信协议的类别，即文本(text)和二进制(binary)传输协议, 为节约带宽，提供传输效率，一般情况下使用二进制类型的传输协议。

分布式系统的RPC解决方案

**Avro:** Avro属于Apache Hadoop的一个子项目。 Avro提供两种序列化格式：JSON格式或者Binary格式。Binary格式在空间开销和解析性能方面可以和Protobuf媲美，Avro的产生解决了JSON的冗长和没有IDL的问题

在Hadoop中做Hive、Pig和MapReduce的持久化数据格式

**Protobuf:**

**protocol buffers** 由谷歌开源而来，在谷歌内部久经考验。它将数据结构以.proto文件进行描述，通过代码生成工具可以生成对应数据结构的POJO对象和Protobuf相关的方法和属性。

对性能要求高的RPC调用

具有良好的跨防火墙的访问属性

适合应用层对象的持久化

1. **Netty的零拷贝实现\***

“零拷贝”是指计算机操作的过程中，CPU不需要**为数据在内存之间的拷贝消耗资源**。而它通常是指计算机在网络上发送文件时，**不需要将文件内容拷贝到用户空间（User Space）而直接在内核空间（Kernel Space）中传输到网络的方式**。

**传统意义的零拷贝：**

这种方式需要四次数据拷贝和四次上下文切换：  
1. 数据从磁盘读取到内核的read buffer  
2. 数据从内核缓冲区拷贝到用户缓冲区  
3. 数据从用户缓冲区拷贝到内核的socket buffer  
4. 数据从内核的socket buffer拷贝到网卡接口的缓冲区

**通过java的FileChannel.transferTo方法，可以避免上面两次多余的拷贝（当然这需要底层操作系统支持）**  
1. 调用transferTo,数据从文件由DMA引擎拷贝到内核read buffer  
2. 接着DMA从内核read buffer将数据拷贝到网卡接口buffer

上面的两次操作都不需要CPU参与，所以就达到了零拷贝。

**对于ByteBuf，Netty提供了多种实现：**  
1. Heap ByteBuf:直接在堆内存分配  
2. Direct ByteBuf：直接在内存区域分配而不是堆内存  
3. CompositeByteBuf：组合Buffer

**Direct Buffers**

**直接在内存区域分配空间**，而不是在堆内存中分配。如果使用传统的堆内存分配，当我们需要将数据通过socket发送的时候，就需要从堆内存拷贝到直接内存，然后再由**直接内存拷贝到网卡接口层**。  
Netty提供的直接Buffer，直接将数据分配到内存空间，从而避免了数据的拷贝，实现了零拷贝。

**Composite Buffers**

传统的ByteBuffer，如果需要将两个ByteBuffer中的数据组合到一起，我们需要首先创建一个size=size1+size2大小的新的数组，然后将两个数组中的数据拷贝到新的数组中。但是使用Netty提供的组合ByteBuf，就可以避免这样的操作，因为CompositeByteBuf并没有真正将多个Buffer组合起来，而是**保存了它们的引用**，从而避免了数据的拷贝，实现了零拷贝。

**对于FileChannel.transferTo的使用**

Netty中使用了FileChannel的transferTo方法，该方法依赖于操作系统实现零拷贝。

1. Netty的接收和发送**ByteBuffer采用DIRECT BUFFERS**，使用**堆外直接内存进行Socket读写**，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存（HEAP BUFFERS）进行Socket读写，JVM会将堆内存Buffer拷贝一份到直接内存中，然后才写入Socket中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。

2. Netty提供了**组合Buffer对象**，可以聚合多个ByteBuffer对象，用户可以像操作一个Buffer那样方便的对组合Buffer进行操作，避免了传统通过内存拷贝的方式将几个小Buffer合并成一个大的Buffer。

3. Netty的文件传输采用了transferTo方法，它可以直接将**文件缓冲区的数据发送到目标Channel**，避免了传统通过循环write方式导致的内存拷贝问题。

1. **Netty的高性能表现在哪些方面\***

**异步非阻塞通信**

 Netty的IO线程NioEventLoop由于聚合了多路复用器Selector，可以同时并发处理成百上千个客户端Channel，由于读写操作都 是非阻塞的，这就可以充分提升IO线程的运行效率，避免由于频繁IO阻塞导致的线程挂起。另外，由于Netty采用了异步通信模式，一个IO线程可以并发处理N个客户端连接和读写操作，这从根本上解决了传统同步阻塞IO一连接一线程模型，架构的性能、弹性伸缩能力和可靠性都得到了极大的提升。

**零拷贝**

  1) Netty的接收和发送ByteBuffer采用DIRECT BUFFERS，使用堆外直接内存进行Socket读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存（HEAP BUFFERS）进行Socket读写，JVM会将堆内存Buffer拷贝一份到直接内存中，然后才写入Socket中。相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。  
       2) Netty提供了组合Buffer对象，可以聚合多个ByteBuffer对象，用户可以像操作一个Buffer那样方便的对组合Buffer进行操作，避免了传统通过内存拷贝的方式将几个小Buffer合并成一个大的Buffer。  
       3) Netty的文件传输采用了transferTo方法，它可以直接将文件缓冲区的数据发送到目标Channel，避免了传统通过循环write方式导致的内存拷贝问题。

**内存池**

   随着JVM虚拟机和JIT即时编译技术的发展，对象的分配和回收是个非常轻量级的工作。但是对于缓冲区Buffer，情况却稍有不同，特别是对于堆外直接内存的分配和回收，是一件耗时的操作。为了尽量重用缓冲区，**Netty提供了基于内存池的缓冲区重用机制（PooledByteBuf）**。

**高效的Reactor线程模型**

1) Reactor单线程模型；  
       2) Reactor多线程模型；

       3) 主从Reactor多线程模型

**无锁化的串行设计理念**

 在大多数场景下，并行多线程处理可以提升系统的并发性能。但是，如果对于共享资源的并发访问处理不当，会带来严重的锁竞争，这最终会导致性能的下降。为了尽可能的避免锁竞争带来的性能损耗，可以通过串行化设计，即消息的处理尽可能在同一个线程内完成，期间不进行线程切换，这样就避免了多线程竞争和同步锁。  
       为了尽可能提升性能，Netty采用了**串行无锁化设计**，在IO线程内部进行串行操作，避免多线程竞争导致的性能下降。表面上看，串行化设计似乎 CPU利用率不高，并发程度不够。但是，通过**调整NIO线程池的线程参数，可以同时启动多个串行化的线程并行运行**，这种局部无锁化的串行线程设计相比一个队列-多个工作线程模型性能更优。  
       Netty的NioEventLoop读取到消息之后，直接调用ChannelPipeline的fireChannelRead(Object msg)，只要用户不主动切换线程，一直会由NioEventLoop调用到用户的Handler，期间不进行线程切换，这种串行化处理方式避免了多线程 操作导致的锁的竞争，从性能角度看是最优的。

**高效的并发编程**

Netty的高效并发编程主要体现在如下几点：  
       1) volatile的大量、正确使用;  
       2) CAS和原子类的广泛使用；  
       3) 线程安全容器的使用；  
       4) 通过读写锁提升并发性能。

**高性能的序列化框架**

影响序列化性能的关键因素总结如下：  
       1) 序列化后的码流大小（网络带宽的占用）；  
       2) 序列化&反序列化的性能（CPU资源占用）；  
       3) 是否支持跨语言（异构系统的对接和开发语言切换）。  
Netty默认提供了对**Google Protobuf**的支持，通过扩展Netty的编解码接口，用户可以实现其它的高性能序列化框架，例如Thrift的压缩二进制编解码框架。

**灵活的TCP参数配置能力**

 合理设置TCP参数在某些场景下对于性能的提升可以起到显著的效果，例如SO\_RCVBUF和SO\_SNDBUF。如果设置不当，对性能的影响是非常大的。下面总结下对性能影响比较大的几个配置项：  
       1) SO\_RCVBUF和SO\_SNDBUF：通常建议值为128K或者256K；  
       2) SO\_TCPNODELAY：NAGLE算法通过将缓冲区内的小封包自动相连，组成较大的封包，阻止大量小封包的发送阻塞网络，从而提高网络应用效率。但是对于时延敏感的应用场景需要关闭该优化算法；  
       3) 软中断：如果Linux内核版本支持RPS（2.6.35以上版本），开启RPS后可以实现软中断，提升网络吞吐量。RPS根据数据包的源地址，目的地址以及目的和源端口，计算出一个hash值，然后根据这个hash值来选择软中断运行的cpu，从上层来看，也就是说将每个连接和cpu绑定，并通过这个 hash值，来均衡软中断在多个cpu上，提升网络并行处理性能。  
Netty在启动辅助类中可以灵活的配置TCP参数，满足不同的用户场景。

**缓存**

1. **Redis用过哪些数据数据，以及Redis底层怎么实现\***

List，Map，String，ZSet，Set，

Redis的底层是C++实现的



1. **Redis缓存穿透，缓存雪崩\***

**缓存穿透**是指查询一个一定不存在的数据，由于缓存是不命中时被动写的，并且出于容错考虑，如果从存储层查不到数据则不写入缓存，这将导致这个**不存在的数据每次请求都要到存储层去查询**，失去了缓存的意义。在流量大时，可能DB就挂掉了，要是有人利用不存在的key频繁攻击我们的应用，这就是漏洞。

有很多种方法可以有效地解决缓存穿透问题，最常见的则是**采用布隆过滤器**，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被 这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。另外也有一个更为简单粗暴的方法（我们采用的就是这种），如果一个查询返回的数据为空（**不管是数据不存在，还是系统故障**），我们仍然把这个**空结果进行缓存，但它的过期时间会很短，最长不超过五分钟**。

**缓存雪崩**是指在我们设置**缓存时采用了相同的过期时间**，导致缓存在某一时刻同时失效，请求**全部转发到DB**，DB瞬时压力过重雪崩。

缓存失效时的雪崩效应对底层系统的冲击非常可怕。大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线 程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。这里分享一个简单方案就时讲缓存失效时间分散开，比如我们可以在原有的**失效时间基础上增加一个随机值**，比如1-5分钟随机，这样每一个缓存的过期时间的重复率就会降低，就很难引发集体失效的事件。

1. **如何使用Redis来实现分布式锁\***

分布式锁一般有三种实现方式：

1. 数据库乐观锁；

2. 基于Redis的分布式锁；

3. 基于ZooKeeper的分布式锁。

**互斥性**。在任意时刻，只有一个客户端能持有锁。

**不会发生死锁**。即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。

**具有容错性**。只要大部分的Redis节点正常运行，客户端就可以加锁和解锁。

解铃还须系铃人。加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。

**加锁操作**：jedis.set(key,value,"NX","EX",timeOut)【保证加锁的原子操作】

key就是redis的key值作为锁的标识，value在这里作为客户端的标识，只有key-value都比配才有删除锁的权利【保证安全性】

通过timeOut设置过期时间保证不会出现死锁【避免死锁】

NX，EX什么意思？

NX：只有这个key不存才的时候才会进行操作，if not exists；

EX：设置key的过期时间为秒，具体时间由第5个参数决定

luaScript 这个字符串是个lua脚本，代表的意思是如果根据key拿到的value跟传入的value相同就执行del，否则就返回0【保证安全性】

jedis.eval(String,list,list);这个命令就是去执行lua脚本，KEYS的集合就是第二个参数，ARGV的集合就是第三参数【保证解锁的原子操作】

1. Redis的并发竞争问题如何解决

1.客户端角度，为保证每个客户端间正常有序与Redis进行通信，对连接进行池化，同时对客户端读写Redis操作采用内部锁synchronized。

2.服务器角度，利用setnx实现锁。

<https://www.cnblogs.com/liuyang0/p/6744076.html>

获取锁的时候，使用setnx加锁，并使用expire命令为锁添加一个超时时间，超过该时间则自动释放锁，锁的value值为一个随机生成的UUID，通过此在释放锁的时候进行判断。

获取锁的时候还设置一个获取的超时时间，若超过这个时间则放弃获取锁。

释放锁的时候，通过UUID判断是不是该锁，若是该锁，则执行delete进行锁释放。

1. **Redis持久化的几种方式，优缺点是什么，怎么实现的\***

一种是**RDB持久化**（原理是将Reids在内存中的数据库**记录定时dump到磁盘上的RDB持久化**），另外一种是**AOF（append only file）持久化**（原理是**将Reids的操作日志以追加的方式写入文件**）。

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，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。

AOF持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。

#### RDB存在哪些优势呢？

1). 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，这对于**文件备份而言是非常完美的**。比如，你可能打算每个小时归档一次最近24小时的数据，同时还要每天归档一次最近30天的数据。通过这样的备份策略，一旦系统出现灾难性故障，我们可以非常**容易的进行恢复**。

2). **对于灾难恢复而言，RDB是非常不错的选择**。因为我们可以非常轻松的将一个单独的文件压缩后再转移到其它存储介质上。

3). **性能最大化**。对于Redis的服务进程而言，在开始持久化时，它唯一需要做的只是fork出子进程，之后再由子进程完成这些持久化的工作，这样就可以极大的避免服务进程执行IO操作了。

4). 相比于AOF机制，如果数据集很大，RDB的启动效率会更高。

RDB又存在哪些劣势呢？

1). **如果你想保证数据的高可用性，即最大限度的避免数据丢失**，那么RDB将不是一个很好的选择。因为系统一旦在**定时持久化之前出现宕机现象**，此前没有来得及写入磁盘的数据都将丢失。

2). 由于RDB是通过**fork子进程来协助完成数据持久化工作的**，因此，如果当数据集较大时，可能会导致整个**服务器停止服务几百毫秒**，甚至是1秒钟。

#### AOF的优势有哪些呢？

1). **该机制可以带来更高的数据安全性**，即数据持久性。Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。事实上，**每秒同步也是异步完成的**，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。可以预见，这种方式在**效率上是最低的**。至于无同步，无需多言，我想大家都能正确的理解它。

2). 由于该机制对日志文件的写入操作采用的是**append模式**，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。然而如果我们本次操作只是写入了一半数据就出现了系统崩溃问题，不用担心，在Redis下一次启动之前，我们可以通过**redis-check-aof工具**来帮助我们解决数据一致性的问题。

3). 如果**日志过大，Redis可以自动启用rewrite机制**。即Redis以append模式不断的将修改数据写入到老的磁盘文件中，同时Redis还会创建一个新的文件用于记录此期间有哪些修改命令被执行。因此在进行rewrite切换时可以更好的保证数据安全性。

4). AOF包含一个格式清晰、易于理解的日志文件用于记录所有的修改操作。事实上，我们也可以通过该文件完成数据的重建。

AOF的劣势有哪些呢？

1). 对于相同数量的数据集而言，**AOF文件通常要大于RDB文件**。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

2). 根据同步策略的不同，**AOF在运行效率上往往会慢于RDB**。总之，每秒同步策略的效率是比较高的，同步禁用策略的效率和RDB一样高效。

1. **Redis的缓存失效策略\***

###### 当缓存需要被清理时（比如空间占用已经接近临界值了），需要使用某种淘汰算法来决定清理掉哪些数据。常用的淘汰算法有下面几种：

FIFO：First In First Out，先进先出。判断被存储的时间，离目前最远的数据优先被淘汰。

LRU：Least Recently Used，最近最少使用。判断最近被使用的时间，目前最远的数据优先被淘汰。

LFU：Least Frequently Used，最不经常使用。在一段时间内，数据被使用次数最少的，优先被淘汰。

1. Redis集群，高可用，原理

高可用性：在主机挂掉后，自动故障转移，使前端服务对用户无影响。

读写分离：将主机读压力分流到从机上。

<https://www.cnblogs.com/leeSmall/p/8414687.html>

RedisCluster是redis的分布式解决方案，在3.0版本后推出的方案，有效地解决了Redis分布式的需求，当一个服务挂了可以快速的切换到另外一个服务，当遇到单机内存、并发等瓶颈时，可使用此方案来解决这些问题

Redis集群采用了**哈希分区的 虚拟槽分区 方式slot 0－16383，共16384槽位**

进行分区

主从复制

持久化

故障切换

1. Redis缓存分片

如果只使用一个redis实例时，其中保存了服务器中全部的缓存数据，这样会有很大风险，如果单台redis服务宕机了将会影响到整个服务。解决的方法就是我们可以采用分片/分区的技术，将原来一台服务器维护的整个缓存，现在换为由多台服务器共同维护内存空间。

1. Redis的数据淘汰策略

<https://blog.csdn.net/suibo0912hf/article/details/51684625>

**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

**技术框架**

1. 看过哪些开源框架的源码

一般是有需要的话，根据相关博客来查看和寻找相关的源码，比对查看分析。

1. 为什么要用Redis，Redis有哪些优缺点？Redis如何实现扩容？

1 读写性能优异

2 支持数据持久化，支持AOF和RDB两种持久化方式

3 支持主从复制，主机会自动将数据同步到从机，可以进行读写分离。

4 [数据结构](http://lib.csdn.net/base/datastructure)丰富：除了支持string类型的value外还支持string、hash、set、zset、list等数据结构。

缺点：

1 [Redis](http://lib.csdn.net/base/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](http://lib.csdn.net/base/redis)的**主从复制采用全量复制**，复制过程中主机会fork出一个子进程对内存做一份快照，并将子进程的内存快照保存为文件发送给从机，这一过程需要确保主机有足够多的空余内存。若快照文件较大，对集群的服务能力会产生较大的影响，而且复制过程是在从机新加入集群或者从机和主机网络断开重连时都会进行，也就是网络波动都会造成主机和从机间的一次全量的数据复制，这对实际的系统运营造成了不小的麻烦。

4 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。为避免这一问题，运维人员在系统上线时必须确保有足够的空间，这对资源造成了很大的浪费。

1.hash算法：分多个实例存储：增加Redis服务器的数量，在客户端对存储的key进行hash运算，存入不同的Redis服务器中，读取时，也进行相同的hash运算，找到对应的Redis服务器  
  2.集群  
  3.对Redis的访问分为写和读

1. **Netty是如何使用线程池的，为什么这么使用\***

EventExecutorGroup 自己实现了Future和submit

AbstractEventExecutorGroup，最上层实现的还是Executor接口

只不过通过 配置数量，配置线程模型

1. **为什么要使用Spring，Spring的优缺点有哪些\***

Spring是一个轻量级的DI和AOP容器框架。

1.使用Spring的IOC容器，将对象之间的依赖关系交给Spring，降低组件之间的耦合性，让我们更专注于应用逻辑

2.可以提供众多服务，事务管理，WS等。

3.AOP的很好支持，方便面向切面编程。

4.对主流的框架提供了很好的集成支持，如[**hibernate**](http://lib.csdn.net/base/javaee),Struts2,JPA等

5.Spring DI机制降低了业务对象替换的复杂性。

6.Spring属于低侵入，代码污染极低。

7.Spring的高度可开放性，并不强制依赖于Spring，开发者可以自由选择Spring部分或全部

缺点：

1.jsp中要写很多代码、控制器过于灵活，缺少一个公用控制器

 2.Spring不支持分布式，这也是EJB仍然在用的原因之一。

1. **Spring的IOC容器初始化流程\***

1.BeanDifinition的Resource定位

2.BeanDifinition的载入与解析

3.BeanDifinition在Ioc容器中的注册，在IOC容器内部将BeanDifinition注入到一个HashMap中去，Ioc容器就是通过这个HashMap来持有这些BeanDifinition数据的。

1. **Spring的IOC容器实现原理，****为什么可以通过byName和ByType找到Bean**

一、**准备配置文件**：就像前边Hello World配置文件一样，在配置文件中声明Bean定义也就是为Bean配置元数据。  
二、**由IOC容器进行解析元数据**： IOC容器的Bean Reader读取并解析配置文件，根据定义生成BeanDefinition配置元数据对象，IOC容器根据BeanDefinition进行实例化、配置及组装Bean。  
三、实例化IOC容器：由客户端实例化容器，获取需要的Bean。

@Resource的作用相当于@Autowired，只不过@Autowired按byType自动注入，而@Resource默认按 byName自动注入罢了。@Resource有两个属性是比较重要的，分是name和type，Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。所以如果使用name属性，则使用byName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不指定name也不指定type属性，这时将通过反射机制使用byName自动注入策略。   
　　@Resource装配顺序   
　　1. 如果同时指定了name和type，则从Spring上下文中找到唯一匹配的bean进行装配，找不到则抛出异常   
　　2. 如果指定了name，则从上下文中查找名称（id）匹配的bean进行装配，找不到则抛出异常   
　　3. 如果指定了type，则从上下文中找到类型匹配的唯一bean进行装配，**找不到或者找到多个**，都会抛出异常   
　　4. 如果既没有指定name，又没有指定type，则自动按照byName方式进行装配；如果没有匹配，则回退为一个原始类型进行匹配，如果匹配则自动装配；

1. Spring AOP实现原理

(1). AOP面向方面编程基于IoC，是对OOP的有益补充；

(2). AOP利用一种称为“横切”的技术，剖解开封装的对象内部，并将那些影响了多个类的公共行为封装到一个**可重用模块**，并将其名为“Aspect”，即方面。所谓“方面”，简单地说，就是将那些与业务无关，却为业务模块所共同调用的 逻辑或责任封装起来，比如**日志记录**，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可操作性和可维护性。

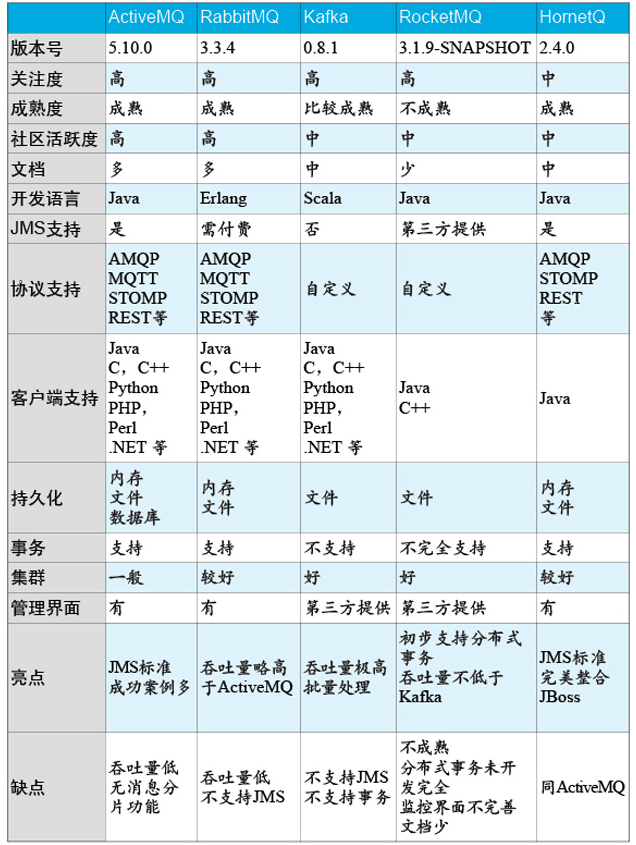
(3). AOP代表的是一个横向的关系，将“对象”比作一个空心的圆柱体，其中封装的是对象的属性和行为；则**面向方面编程**的方法，就是将这个圆柱体以切面形式剖开，选择性的提供业务逻辑。而剖开的切面，也就是所谓的“方面”了。然后它又以**巧夺天功的妙手将这些剖开的切面复原**，不留痕迹，但完成了效果。

(4). 实现AOP的技术，主要分为两大类：一是采用**动态代理技术**，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；二是采用**静态织入的方式，**引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

(5). **Spring实现AOP**：JDK动态代理和CGLIB代理 JDK动态代理：其代理对象必须是某个接口的实现，它是通过在运行期间创建一个接口的实现类来完成对目标对象的代理；其核心的两个类是InvocationHandler和Proxy。 CGLIB代理：实现原理类似于JDK动态代理，只是它在运行期间生成的代理对象是针对目标类扩展的子类。CGLIB是高效的代码生成包，底层是依靠ASM（开源的java字节码编辑类库）操作字节码实现的，性能比JDK强；需要引入包asm.jar和cglib.jar。使用AspectJ注入式切面和@AspectJ注解驱动的切面实际上底层也是通过动态代理实现的。

1. 消息中间件是如何实现的，技术难点有哪些

消息+队列通道，结合生产-消费者模式



**技术深度**

1. 事务的实现原理

 事务必须服从ISO/IEC所制定的ACID原则。ACID是原子性（atomicity）、一致性（consistency）、隔离性 （isolation）和持久性（durability）的缩写。事务的原子性表示事务执行过程中的任何失败都将导致事务所做的任何修改失效。一致性表示 当事务执行失败时，所有被该事务影响的数据都应该恢复到事务执行前的状态。隔离性表示在事务执行过程中对数据的修改，在事务提交之前对其他事务不可见。持久性表示已提交的数据在事务执行失败时，数据的状态都应该正确。

1. 有没有看过JDK源码，看过的类实现原理是什么。
2. HTTP协议

http：+地址+端口+url

基于TCP/IP通信协议来传递数据（HTML 文件, 图片文件, 查询结果等），B/C架构

HTTP request请求：

第一部分：**请求行**，用来说明请求类型,要访问的资源以及所使用的HTTP版本.

GET说明请求类型为GET,[/562f25980001b1b106000338.jpg]为要访问的资源，该行的最后一部分说明使用的是HTTP1.1版本。

第二部分：**请求头部**，紧接着请求行（即第一行）之后的部分，用来说明服务器要使用的附加信息

从第二行起为请求头部，HOST将指出请求的目的地.User-Agent,服务器端和客户端脚本都能访问它,它是浏览器类型检测逻辑的重要基础.该信息由你的浏览器来定义,并且在每个请求中自动发送等等

第三部分：空行，请求头部后面的空行是必须的

即使第四部分的请求数据为空，也必须有空行。

第四部分：请求数据也叫主体，可以添加任意的其他数据。

这个例子的请求数据为空。

HTTP之响应消息Response

HTTP响应也由四个部分组成，分别是：状态行、消息报头、空行和响应正文

1. TCP协议：面向连接的、可靠的、基于字节流的[传输层](https://baike.baidu.com/item/%E4%BC%A0%E8%BE%93%E5%B1%82/4329536)通信协议

IP 协议只是一个地址协议，并不保证数据包的完整。如果路由器丢包（比如缓存满了，新进来的数据包就会丢失），就需要发现丢了哪一个包，以及如何重新发送这个包。这就要依靠 TCP 协议。

简单说，**TCP 协议的作用是，保证数据通信的完整性和可靠性，防止丢包**

TCP是一种**面向连接（连接导向）**的、可靠的基于字节流的传输层通信协议。TCP将用户数据打包成**报文段**，它发送后启动一个定时器，另一端收到的数据进行**确认、对失序的数据重新排序、丢弃重复数据。**  
TCP的特点有：

TCP是面向连接的运输层协议

每一条TCP连接只能有两个端点，每一条TCP连接只能是点对点的

TCP提供可靠交付的服务

TCP提供全双工通信。数据在两个方向上独立的进行传输。因此，连接的每一端必须保持每个方向上的传输数据序号。

面向字节流。面向字节流的含义：虽然应用程序和TCP交互是一次一个数据块，但TCP把应用程序交下来的数据仅仅是一连串的无结构的字节流。

**TCP报文抓取工具：Wireshark**

三次握手：第一次，C向S发送连接请求

第二次，S收到C发过来的报文，ack给客户端

第三次，C收到S验证ack和其他标志位验证。OK后，就可以传输数据

四次挥手：第一次，C请求S中断

第二次，S回复ack，请等待我准备好

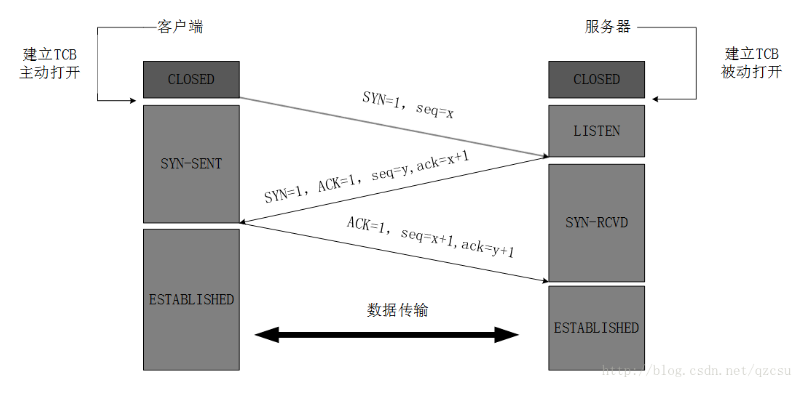
第三次，S确认OK，请准备好关闭，S关闭

第四次，C收到消息，发ack确认然后等待，2ms没有响应回传，自己关闭

**建立连接需要三次握手**

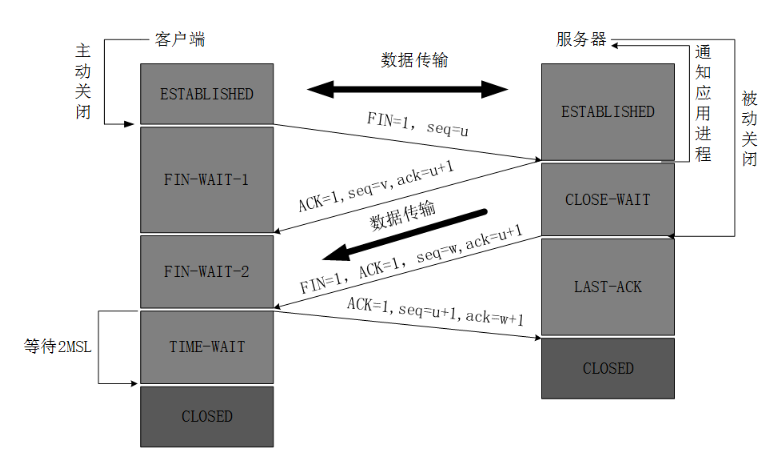
[**https://blog.csdn.net/qzcsu/article/details/72861891**](https://blog.csdn.net/qzcsu/article/details/72861891)

1. TCP服务器进程先创建传输控制块TCB，时刻准备接受客户进程的连接请求，此时服务器就进入了LISTEN（监听）状态；
2. TCP客户进程也是先创建传输控制块TCB，然后向服务器发出连接请求报文，这是报文首部中的同部位SYN=1，同时选择一个初始序列号 seq=x ，此时，TCP客户端进程进入了 SYN-SENT（同步已发送状态）状态。TCP规定，SYN报文段（SYN=1的报文段）不能携带数据，但需要消耗掉一个序号。
3. TCP服务器收到请求报文后，如果同意连接，则发出确认报文。确认报文中应该 ACK=1，SYN=1，确认号是ack=x+1，同时也要为自己初始化一个序列号 seq=y，此时，TCP服务器进程进入了SYN-RCVD（同步收到）状态。这个报文也不能携带数据，但是同样要消耗一个序号。
4. TCP客户进程收到确认后，还要向服务器给出确认。确认报文的ACK=1，ack=y+1，自己的序列号seq=x+1，此时，TCP连接建立，客户端进入ESTABLISHED（已建立连接）状态。TCP规定，ACK报文段可以携带数据，但是如果不携带数据则不消耗序号。
5. 当服务器收到客户端的确认后也进入ESTABLISHED状态，此后双方就可以开始通信了。

****

**断开连接需要四次挥手**

1. 提醒：中断连接端可以是Client端，也可以是Server端。只要将下面两角色互换即可。客户端进程发出连接释放报文，并且停止发送数据。释放数据报文首部，FIN=1，其序列号为seq=u（等于前面已经传送过来的数据的最后一个字节的序号加1），此时，客户端进入FIN-WAIT-1（终止等待1）状态。 TCP规定，FIN报文段即使不携带数据，也要消耗一个序号。
2. 服务器收到连接释放报文，发出确认报文，ACK=1，ack=u+1，并且带上自己的序列号seq=v，此时，服务端就进入了CLOSE-WAIT（关闭等待）状态。TCP服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个CLOSE-WAIT状态持续的时间。
3. 客户端收到服务器的确认请求后，此时，客户端就进入FIN-WAIT-2（终止等待2）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的数据）。
4. 服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，FIN=1，ack=u+1，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为seq=w，此时，服务器就进入了LAST-ACK（最后确认）状态，等待客户端的确认。
5. 客户端收到服务器的连接释放报文后，必须发出确认，ACK=1，ack=w+1，而自己的序列号是seq=u+1，此时，客户端就进入了TIME-WAIT（时间等待）状态。注意此时TCP连接还没有释放，必须经过2∗∗MSL（最长报文段寿命）的时间后，当客户端撤销相应的TCB后，才进入CLOSED状态。
6. 服务器只要收到了客户端发出的确认，立即进入CLOSED状态。同样，撤销TCB后，就结束了这次的TCP连接。可以看到，服务器结束TCP连接的时间要比客户端早一些。



1. **一致性Hash算法\***

一致性hash作为一个**负载均衡算法**，可以用**在分布式缓存、数据库的分库分表**等场景中，还可以应用在负载均衡器中作为作为负载均衡算法。在有多台服务器时，对于某个请求资源通过hash算法，映射到某一个台服务器，当增加或减少一台服务器时，可能会改变这些资源对应的hash值，这样可能导致一部分缓存或数据失效了。一致性hash就是尽可能在将同一个资源请求路由到同一台服务器中。

一致性哈希采用的做法如下：**引入一个环的概念**，如上面的第一个图。先将机器映射到这个环上，再将数据也通过相同的哈希函数映射到这个环上，**数据存储在它顺时针走向的那台机器上**。以环为中介，实现了数据与机器数目之间的解耦。这样，当机器的数目变化时，只会影响到增加或删除的那台机器所在的环的邻接机器的数据存储，而其他机器上的数据不受影响。

1. **JVM如何加载字节码文件\***

类从被加载到虚拟机内存到卸载出内存的生命周期包括：**加载->连接(验证->准备->解析)->初始化-**>使用->卸载

加载：1通过一个类的权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的运行时数据结构

3 在java堆中生成一个代表这个类的java.lang.Class对象，作为方法区这些数据的访问入口。

进行 文件格式、元数据、字节码。符号引用验证

准备 正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区中进行分配。

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程，类接口，字段，类方法，接口方法。

初始化加载程序字节码

1. 类加载器如何卸载字节码
2. IO和NIO的区别，NIO优点

|  |  |
| --- | --- |
| **IO** | **NIO** |
| 面向流 | 面向缓冲 |
| 阻塞IO | 非阻塞IO |
| 无 | 选择器 |

**IO是面向流的，NIO是面向缓冲区的**

Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方；

NIO则能前后移动流中的数据，因为是面向缓冲区的

**IO流是阻塞的，NIO流是不阻塞的**

Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了

Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取。NIO可让您只使用一个（或几个）单线程管理多个通道（网络连接或文件），但付出的代价是解析数据可能会比从一个阻塞流中读取数据更复杂。

非阻塞写也是如此。一个线程请求写入一些数据到某通道，但不需要等待它完全写入，这个线程同时可以去做别的事情。

**选择器**

Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

1. Java线程池的实现原理，keepAliveTime等参数的作用。

**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

**handler**：超出线程范围和队列容量的任务的处理程序

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://www.jianshu.com/p/87bff5cc8d8c](https://www.jianshu.com/p/87bff5cc8d8c%20)

**keepAliveTime：**线程空闲时的存活时间，即当线程没有任务执行时，继续存活的时间；默认情况下，该参数只在线程数大于corePoolSize时才有用；

1. HTTP连接池实现原理

1、**降低延迟：**如果不采用连接池，每次连接发起Http请求的时候都会重新建立TCP连接(经历3次握手)，用完就会关闭连接(4次挥手)，如果采用连接池则减少了这部分时间损耗，别小看这几次握手，本人经过测试发现，基本上3倍的时间延迟

2、**支持更大的并发：**如果不采用连接池，每次连接都会打开一个端口，在大并发的情况下系统的端口资源很快就会被用完，导致无法建立新的连接

PoolingHttpClientConnectionManager

配置请求超时设置—RequestConfig

CloseableHttpClient 获取httpClient对象，post，get封装

1. 数据库连接池实现原理

装载数据库驱动程序；

通过jdbc建立数据库连接；  
访问数据库，执行sql语句；

断开数据库连接。

创建连接池，获取连接，用完后返回给连接池。

1. 数据库的实现原理

**分布式**

1. 什么是CAP定理

一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本），换句话就是说，**任何时刻，所用的应用程序都能访问得到相同的数据。**

可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性），换句话就是说，**任何时候，任何应用程序都可以读写数据**。

分区容错性（P）：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择，换句话说，**系统可以跨网络分区线性的伸缩和扩展**。

1. CAP 理论和 BASE 理论

eBay的架构师Dan Pritchett源于对大规模分布式系统的实践总结，在ACM上发表文章提出BASE理论，BASE理论是对CAP理论的延伸，核心思想是即使无法做到**强一致性**（Strong Consistency，CAP的一致性就是强一致性），但应用可以采用适合的方式达到**最终一致性**（Eventual Consitency）。

**基本可用（Basically Available）**

基本可用是指分布式系统在出现故障的时候，允许损失部分可用性，即保证核心可用。

电商大促时，为了应对访问量激增，部分用户可能会被引导到降级页面，服务层也可能只**提供降级服务。这就是损失部分可用性的体现**。

**软状态（ Soft State）**

软状态是指允许系统存在**中间状态，而该中间状态不会影响系统整体可用性**。分布式存储中一般一份数据至少会有三个副本，允许不同节点间副本同步的延时就是软状态的体现。mysql replication的异步复制也是一种体现。

**最终一致性（ Eventual Consistency）**

最终一致性是指系统中的所有数据副本经过一定时间后，**最终能够达到一致的状态**。弱一致性和强一致性相反，最终一致性是弱一致性的一种特殊情况。

1. CAP 理论

**1. CAP理论**

2000年7月，加州大学伯克利分校的Eric Brewer教授在ACM PODC会议上提出CAP猜想。2年后，麻省理工学院的Seth Gilbert和Nancy Lynch从理论上证明了CAP。之后，CAP理论正式成为分布式计算领域的公认定理。

CAP理论为：一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项。

**1.1 一致性（Consistency）**

一致性指“all nodes see the same data at the same time”，即更新操作成功并返回客户端完成后，所有节点在同一时间的数据完全一致。

**1.2 可用性（Availability）**

可用性指“Reads and writes always succeed”，即服务一直可用，而且是正常响应时间。

**1.3 分区容错性（Partition tolerance）**

分区容错性指“the system continues to operate despite arbitrary message loss or failure of part of the system”，即分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务。

1. CAP 理论和最终一致性

一言以蔽之：**过程松，结果紧**，最终结果必须保持一致性

最终一致性是弱一致性的一种特例。假如A首先write了一个值到存储系统，存储系统保证如果在A,B,C后续读取之前没有其它写操作更新同样的值的话，最终所有的读取操作都会读取到最A写入的最新值。此种情况下，如果没有失败发生的话，“不一致性窗口”的大小依赖于以下的几个因素：交互延迟，系统的负载，以及复制技术中replica的个数（这个可以理解为master/salve模式中，salve的个数），最终一致性方面最出名的系统可以说是DNS系统，当更新一个域名的IP以后，根据配置策略以及缓存控制策略的不同，最终所有的客户都会看到最新的值

1. 最终一致性实现方式

<https://blog.csdn.net/zxl315/article/details/53433707>

1. 一致性 Hash

<https://blog.csdn.net/zxl315/article/details/53433707>

1. 分布式事务，两阶段提交。

两阶段提交涉及到多个节点的网络通信,通信时间如果过长,事务的相对时间也就会过长,那么锁定资源的时间也就长了.在高并发的服务中,就会存在严重的性能瓶颈

1. **如何实现分布式锁\***

<https://blog.csdn.net/xlgen157387/article/details/79036337>

基于数据库实现分布式锁；   
基于缓存（Redis等）实现分布式锁；   
基于Zookeeper实现分布式锁；

1. **如何实现分布式Session\***

<https://www.cnblogs.com/cxrz/p/8529587.html>

1. 基于数据库的Session共享

2. 基于NFS共享文件系统

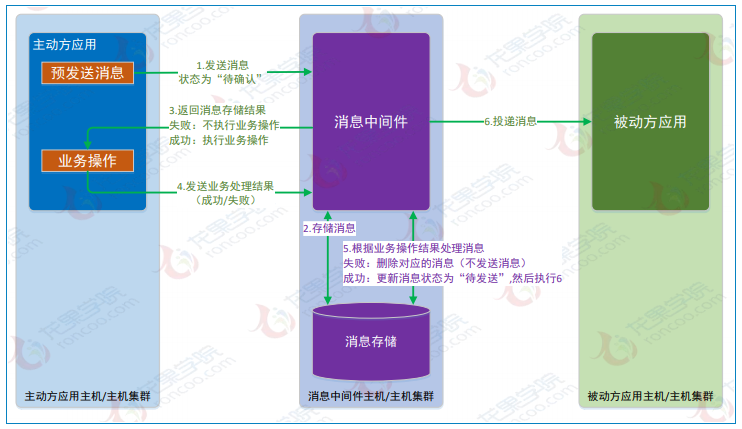
3. 基于memcached 的session，如何保证 memcached 本身的高可用性？

4. 基于resin/tomcat web容器本身的session复制机制

5. 基于TT/Redis 或 jbosscache 进行 session 共享。

6. 基于cookie 进行session共享

1. **如何保证消息的一致性\***



1. 负载均衡

负载均衡是高可用网络基础架构的的一个关键组成部分，有了负载均衡，我们通常可以将我们的应用服务器部署多台，然后通过负载均衡将用户的请求分发到不同的服务器用来提高网站、应用、数据库或其他服务的性能以及可靠性

<https://baijiahao.baidu.com/s?id=1595722616086971162&wfr=spider&for=pc>

1. **正向代理（客户端代理）和反向代理（服务器端代理）\***

**正向代理**是一个位于**客户端和原始服务器(origin server)**之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标(原始服务器)，然后代理向原始服务器转交请求并将获得的内容返回给客户端。客户端必须要进行一些特别的设置才能使用正向代理。

（1）访问**原来无法访问的资源**，如google

    （2）可以做**缓存**，加速访问资源

　　（3）对**客户端访问授权**，上网进行认证

　　（4）代理可以记录用户**访问记录**（上网行为管理），对外隐藏用户信息

**反向代理**（Reverse Proxy）实际运行方式是指以**代理服务器来接受internet上的连接请求**，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给internet上请求连接的客户端，此时代理服务器对外就表现为一个服务器。

（1）保证**内网**的安全，可以使用反向代理提供WAF功能，阻止**web攻击**

（2）负载均衡，通过**反向代理服务器来优化网站的负载**

nginx支持配置反向代理，通过反向代理实现网站的负载均衡。这部分先写一个nginx的配置，后续需要深入研究nginx的**代理模块和负载均衡模块**。

<https://www.cnblogs.com/Anker/p/6056540.html>

1. CDN实现原理

（1）CDN节点解决了跨运营商和跨地域访问的问题，访问延时大大降低；

（2）大部分请求在CDN边缘节点完成，CDN起到了分流作用，减轻了源站的负载。

CDN即内容分发网络，加速的意思，那么**网站CND服务**是网站加速服务。

**CDN加速**将网站的内容缓存在网络边缘（离用户接入网络最近的地方），然后在用户访问网站内容的时候，通过调度系统将用户的请求路由或者引导到离用户**接入网络最近或者访问效果的缓存服务器**上，有该缓存服务器为用户提供内容服务；相对于直接访问源站，这种方式缩短了**用户和内容之间的网络距离**，从而达到加速的效果

1. **怎么提升系统的QPS和吞吐量\***

**QPS（TPS）：**每秒钟request/事务 数量

**并发数**：系统同时处理的request/事务数

**响应时间**：一般取平均响应时间

简单而言通过**增加集群**来提升qps和吞吐量

实际上要比这个要复杂

首先我们需要知道系统的瓶颈

我们所知道的系统拓扑架构

对于rest接口而言

**系统**设施依次是：

dns

　　nginx

　　　　tomcat

　　　　　　db/soa

首先我们可以通过增加集群来增加qps和吞吐量

其次考虑到负载均衡的问题，我们可以通过其他设施来保证集群节点的负载均衡，进一步提高系统qps

于是就有**nginx集群+负载均衡**

**tomcat集群+负载均衡**

到db/soa这一层的时候，同样也可以通过增加集群+负载均衡的方式来解决

我们还可以在每一层**增加缓存来应对热点数据**

然而另外一个方面，可以系统拆分，服务拆分，分别针对瓶颈的系统单独增加集群和负载均衡来解决

同样db也可以分库分表，

因为单表超过1000万条数据时就很慢了，所以这个时候就需要库拆分，于是就有垂直拆分，水平拆分。

异步化，可以不同调用的异步化，使用mq，比如发送短信，发送邮件等

综上所述：

**集群+负载均衡**

**增加缓存**

**系统拆分**

**分库分表**

**垂直拆分+水平拆分**

**异步化+MQ**

1. **Dubbo的底层实现原理和机制\***

采用**分层的方式**来架构，采用服务**提供方和服务消费方**简单模型。

Dubbo框架设计一共划分了10个层，而最上面的**Service层**是留给实际想要使用Dubbo开发分布式服务的开发者实现业务逻辑的**接口层**。图中左边淡蓝背景的为服务消费方使用的接口，右边淡绿色背景的为服务提供方使用的接口，位于中轴线上的为双方都用到的接口。

**1、服务接口层（Service）**：该层是与实际业务逻辑相关的，根据服务提供方和服务消费方的业务设计对应的接口和实现。

**2、配置层（Config）**：对外配置接口，以ServiceConfig和ReferenceConfig为中心，可以直接new配置类，也可以通过spring解析配置生成配置类。

**3、服务代理层（Proxy）**：服务接口透明代理，生成服务的客户端Stub和服务器端Skeleton，以ServiceProxy为中心，扩展接口为ProxyFactory。

**4、服务注册层（Registry）**：封装服务地址的注册与发现，以服务URL为中心，扩展接口为RegistryFactory、Registry和RegistryService。可能没有服务注册中心，此时服务提供方直接暴露服务。

**5、集群层（Cluster）**：封装多个提供者的路由及负载均衡，并桥接注册中心，以Invoker为中心，扩展接口为Cluster、Directory、Router和LoadBalance。将多个服务提供方组合为一个服务提供方，实现对服务消费方来透明，只需要与一个服务提供方进行交互。

**6、监控层（Monitor）**：RPC调用次数和调用时间监控，以Statistics为中心，扩展接口为MonitorFactory、Monitor和MonitorService。

**7、远程调用层（Protocol）**：封将RPC调用，以Invocation和Result为中心，扩展接口为Protocol、Invoker和Exporter。Protocol是服务域，它是Invoker暴露和引用的主功能入口，它负责Invoker的生命周期管理。Invoker是实体域，它是Dubbo的核心模型，其它模型都向它靠扰，或转换成它，它代表一个可执行体，可向它发起invoke调用，它有可能是一个本地的实现，也可能是一个远程的实现，也可能一个集群实现。

**8、信息交换层（Exchange）**：封装请求响应模式，同步转异步，以Request和Response为中心，扩展接口为Exchanger、ExchangeChannel、ExchangeClient和ExchangeServer。

**9、网络传输层（Transport）**：抽象mina和netty为统一接口，以Message为中心，扩展接口为Channel、Transporter、Client、Server和Codec。

**10、数据序列化层（Serialize）**：可复用的一些工具，扩展接口为Serialization、 ObjectInput、ObjectOutput和ThreadPool。

Dubbo提供了三个关键功能：**基于接口的远程调用**，**容错与负载均衡**，**服务自动注册与发现**。

Dubbo作为一个分布式服务框架，主要具有如下几个核心的要点：

**服务定义**  
    服务是围绕服务提供方和服务消费方的，服务提供方实现服务，而服务消费方调用服务。

**服务注册**  
    服务提供方，它需要发布服务，而且由于应用系统的复杂性，服务的数量、类型也不断膨胀；对于服务消费方，它最关心如何获取到它所需要的服务，而面对复杂的应用系统，需要管理大量的服务调用。而且，对于服务提供方和服务消费方来说，他们还有可能兼具这两种角色，即既需要提供服务，有需要消费服务。  
通过将服务统一管理起来，可以有效地优化内部应用对服务发布/使用的流程和管理。服务注册中心可以通过**特定协议来完成服务对外的统一**。Dubbo提供的注册中心有如下几种类型可供选择：

Multicast注册中心

Zookeeper注册中心

Redis注册中心

Simple注册中心

**服务监控**

     服务提供方，还是服务消费方，他们都需要对服务调用的实际状态进行有效的监控，从而改进服务质量。

**远程通信与信息交换**

远程通信需要指定通信双方所约定的协议，在保证通信双方理解协议语义的基础上，还要保证高效、稳定的消息传输。Dubbo继承了当前主流的网络通信框架，主要包括如下几个：

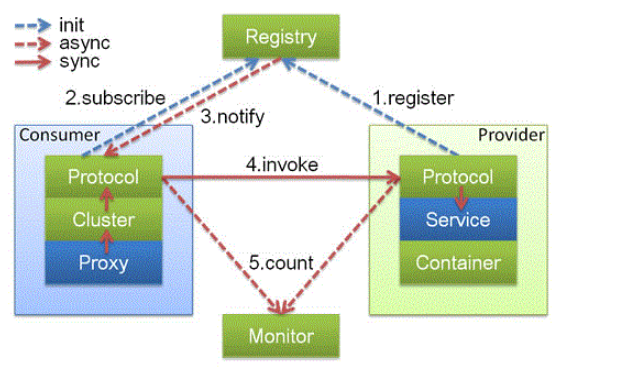
Mina

Netty

Grizzly

**服务调用**

    下面从Dubbo官网直接拿来，看一下基于RPC层，服务提供方和服务消费方之间的调用关系，



上图中，蓝色的表示与业务有交互，绿色的表示只对Dubbo内部交互。上述图所描述的调用流程如下：

服务提供方发布服务到服务注册中心；

服务消费方从服务注册中心订阅服务；

服务消费方调用已经注册的可用服务；

**节点角色说明：**

Provider: **暴露服务**的服务提供方。

Consumer: **调用远程服务**的服务消费方。

Registry: 服务注册与发现的注册中心。

Monitor: 统计服务的调用次数和调用时间的监控中心。

Container: 服务运行容器。

调用关系说明：

0. **服务容器**负责启动，加载，运行服务提供者。

1. **服务提供者**在启动时，向注册中心注册自己提供的服务。

2. **服务消费者**在启动时，向注册中心订阅自己所需的服务。

3. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将**基于长连接推送变更数据给消费者**。

4. **服务消费者**，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

5. **服务消费者和提供者**，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

1. **描述一个服务从发布到被消费的详细过程\***

**创建接口服务**

配置文件配置服务注册中心，**配置接口服务**

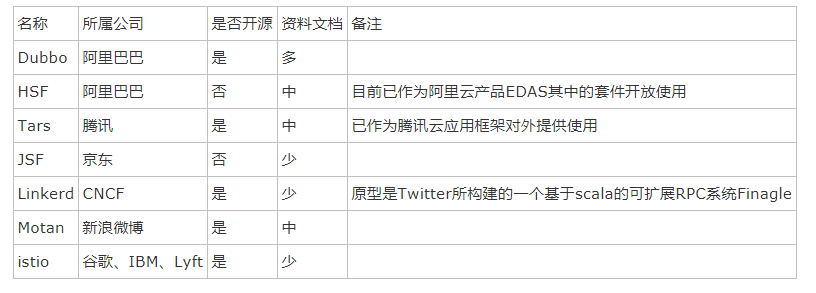
启动后，服务提供者启动时，向注册中心注册自己提供的服务，服务消费者在启动时，向注册中心订阅自己所需的服务

注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

1. 分布式系统怎么做服务治理



服务自动注册

客户端自动发现

变更下发

1. 接口的幂等性的概念

**在数学里，幂等有两种主要的定义：**

在某二元运算下，幂等元素是指被自己重复运算(或对于函数是为复合)的结果等于它自己的元素。例如，乘法下唯一两个幂等实数为0和1。 即 s \*s = s

某一元运算为幂等的时，其作用在任一元素两次后会和其作用一次的结果相同。例如，高斯符号便是幂等的，即f(f(x)) = f(x)。

HTTP的幂等性指的是一次和多次请求某一个资源应该具有相同的副作用。如通过PUT接口将数据的Status置为1，无论是第一次执行还是多次执行，获取到的结果应该是相同的，即执行完成之后Status =1。

orderStatus由0->1 是需要幂等性的

1. **消息中间件如何解决消息丢失问题\***

消息持久化

ACK确认机制

设置集群镜像模式

1）单节点模式：最简单的情况，非集群模式，节点挂了，消息就不能用了。业务可能瘫痪，只能等待。  
2）普通模式：默认的集群模式，某个节点挂了，该节点上的消息不能用，有影响的业务瘫痪，只能等待节点恢复重启可用（必须持久化消息情况下）。  
3）镜像模式：把需要的队列做成镜像队列，存在于多个节点，属于RabbitMQ的HA方案

消息补偿机制：息补偿机制需要建立在消息要写入DB日志，发送日志，接受日志，两者的状态必须记录。然后根据DB日志记录check 消息发送消费是否成功，不成功，进行消息补偿措施，重新发送消息处理。

1. Dubbo的服务请求失败怎么处理

因此，将**应用拆分**，并抽取出**核心服务**来解决上述问题，还要考虑**负载均衡、服务监控、高可用性、服务隔离与降级、路由策略、完善的容错机制、序列化方案的选择、通信框架的选择、开发人员对底层细节无感知、服务升级兼容性**等问题。Dubbo满足了以上所有需求。

1. 重连机制会不会造成错误

dubbo在调用服务不成功时，**默认会重试2次**。  
**Dubbo的路由机制**，会把超时的请求路由到其他机器上，而不是本机尝试，所以 **dubbo的重试机制也能一定程度的保证服务的质量。**但是如果不合理的配置重试次数，当失败时会进行重试多次，这样在某个时间点出现性能问题，调用方再连续重复调用。  
系统请求变为正常值的retries倍，系统压力会大增，容易引起服务雪崩，需要根据业务情况规划好如何进行异常处理，何时进行重试。

1. 对分布式事务的理解

**分布式事务**就是指**事务的参与者、支持事务的服务器、资源服务器以及事务管理器**分别位**于不同的分布式系统的不同节点之上**。以上是百度百科的解释，简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

**分布式事务**，本质上是对多个数据库的事务进行统一控制，按照控制力度可以分为：**不控制、部分控制和完全控制**。不控制就是不引入分布式事务，部分控制就是各种变种的两阶段提交，包括上面提到的消息事务+最终一致性、TCC模式，而完全控制就是完全实现两阶段提交。**部分控制的好处是并发量和性能很好，缺点是数据一致性减弱了，完全控制则是牺牲了性能，保障了一致性，具体用哪种方式，最终还是取决于业务场景。**

**下单—涉及扣库存和更新订单状态。**

1. 如何实现负载均衡，有哪些算法可以实现？

既然要解决后端系统的承载能力：**nginx的配置**

**均衡算法**主要解决将请求如何发送给后端服务

随机（random）、轮训（round-robin）、一致哈希（consistent-hash）和主备（master-slave）。

1. Zookeeper的用途，选举的原理是什么？

分布式系统基本上都是主从结构，所以需要zookeeper进行协调服务，他做很多事情的，比如命名服务，配置管理，集群管理，分布式协调通知等等

当leader崩溃或者leader失去大多数的follower，这时候zk进入恢复模式，恢复模式需要重新选举出一个新的leader，让所有的Server都恢复到一个正确的状态。Zk的选举算法有两种：一种是基于basic paxos实现的，另外一种是基于fast paxos算法实现的。系统默认的选举算法为fast paxos。

**1.服务器初始化时Leader选举**

zookeeper由于其自身的性质，一般建议选取奇数个节点进行搭建分布式服务器集群。以3个节点组成的服务器集群为例，说明服务器初始化时的选举过程。启动第一台安装zookeeper的节点时，无法单独进行选举，启动第二台时，两节点之间进行通信，开始选举Leader。

1）每个Server投出一票。他们两都选自己为Leader，投票的内容为（SID，ZXID）。SID即Server的id，安装zookeeper时配置文件中所配置的myid；ZXID，事务id，为节点的更新程度，ZXID越大，代表Server对Znode的操作越新。由于服务器初始化，每个Sever上的Znode为0，所以Server1投的票为（1,0），Server2为（2,0）。两Server将各自投票发给集群中其他机器。

2）每个Server接收来自其他Server的投票。集群中的每个Server先判断投票有效性，如检查是不是本轮的投票，是不是来Looking状态的服务器投的票。

3）对投票结果进行处理。先了解下处理规则

- 首先对比ZXID。ZXID大的服务器优先作为Leader

- 若ZXID相同，比如初始化的时候，每个Server的ZXID都为0，就会比较myid，myid大的选出来做Leader。

对于Server而言，他接受到的投票为（2,0），因为自身的票为（1,0），所以此时它会选举Server2为Leader，将自己的更新为（2,0）。而Server2收到的投票为Server1的（1,0）由于比他自己小，Server2的投票不变。Server1和Server2再次将票投出，投出的票都为（2,0）。

4） 统计投票。每次投票之后，服务器都会统计投票信息，如果判定某个Server有过半的票数投它，那么该Server将会作为Leader。对于Server1和Server2而言,统计出已经有两台机器接收了（2,0）的投票信息，此时认为选出了Leader。

5）改变服务器状态。当确定了Leader之后，每个Server更新自己的状态，Leader将状态更新为Leading，Follower将状态更新为Following。

**2.服务器运行期间的Leader选举**

zookeeper运行期间，如果有新的Server加入，或者非Leader的Server宕机，那么Leader将会同步数据到新Server或者寻找其他备用Server替代宕机的Server。若Leader宕机，此时集群暂停对外服务，开始在内部选举新的Leader。假设当前集群中有Server1、Server2、Server3三台服务器，Server2为当前集群的Leader，由于意外情况，Server2宕机了，便开始进入选举状态。过程如下

1） 变更状态。其他的非Observer服务器将自己的状态改变为Looking，开始进入Leader选举。

2） 每个Server发出一个投票（myid，ZXID），由于此集群已经运行过，所以每个Server上的ZXID可能不同。假设Server1的ZXID为145，Server3的为122，第一轮投票中，Server1和Server3都投自己，票分别为（1，145）、（3,122）,将自己的票发送给集群中所有机器。

3） 每个Server接收接收来自其他Server的投票，降下来的步骤与启动时步骤相同。

1. 数据的垂直拆分水平拆分。

垂直拆分，对于表来说，可以按业务模型进行拆分

水平拆分，对于表来说，是分多个票取模存放到不同数据库

1. **zookeeper原理和适用场景\***

Zookeeper 作为一个分布式的服务框架，主要用来解决分布式集群中应用系统的一致性问题，它能提供基于类似于文件系统的目录节点树方式的数据存储， Zookeeper 作用主要是用来维护和监控存储的数据的状态变化，通过监控这些数据状态的变化，从而达到基于数据的集群管理

简单的说，zookeeper=文件系统+通知机制。

ZooKeeper以Fast Paxos（帕克索斯）算法为基础，让集群中的每个zk实例数据保持一致。一般部署集群，机器数设置为奇数个，更容易满足>N/2的投票条件。

**Zookeeper应用场景**

**统一命名服务**

       分布式环境下，经常需要对应用/服务进行统一命名，便于识别不同服务。类似于域名与ip之间对应关系，域名容易记住。通过名称来获取资源或服务的地址，提供者等信息按照层次结构组织服务/应用名称可将服务名称以及地址信息写到Zookeeper上，客户端通过Zookeeper获取可用服务列表类。

**配置管理**

       分布式环境下，配置文件管理和同步是一个常见问题。一个集群中，所有节点的配置信息是一致的，比如Hadoop。对配置文件修改后，希望能够快速同步到各个节点上配置管理可交由Zookeeper实现。可将配置信息写入Zookeeper的一个znode上。各个节点监听这个znode。一旦znode中的数据被修改，zookeeper将通知各个节点。

集群管理

       分布式环境中，实时掌握每个节点的状态是必要的。可根据节点实时状态作出一些调整。Zookeeper可将节点信息写入Zookeeper的一个znode上。监听这个znode可获取它的实时状态变化。典型应用比如Hbase中Master状态监控与选举。

**分布式通知/协调**

       分布式环境中，经常存在一个服务需要知道它所管理的子服务的状态。例如，NameNode须知道各DataNode的状态，JobTracker须知道各TaskTracker的状态。心跳检测机制和信息推送也是可通过Zookeeper实现。

**分布式锁**

       Zookeeper是强一致的。多个客户端同时在Zookeeper上创建相同znode，只有一个创建成功。Zookeeper实现锁的独占性。多个客户端同时在Zookeeper上创建相同znode ，创建成功的那个客户端得到锁，其他客户端等待。Zookeeper 控制锁的时序。各个客户端在某个znode下创建临时znode （类型为CreateMode. EPHEMERAL \_SEQUENTIAL），这样，该znode可掌握全局访问时序。

**分布式队列**

       两种队列。当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达，这种是同步队列。队列按照 FIFO 方式进行入队和出队操作，例如实现生产者和消费者模型。（可通过分布式锁实现）  
       同步队列。一个job由多个task组成，只有所有任务完成后，job才运行完成。可为job创建一个/job目录，然后在该目录下，为每个完成的task创建一个临时znode，一旦临时节点数目达到task总数，则job运行完成。

<https://blog.csdn.net/king866/article/details/53992653/>

1. zookeeper watch机制

监视是一种简单的机制，使客户端收到关于ZooKeeper集合中的更改的通知。客户端可以在读取特定znode时设置Watches。Watches会向注册的客户端发送任何znode（客户端注册表）更改的通知。

Znode更改是与znode相关的数据的修改或znode的子项中的更改。只触发一次watches。如果客户端想要再次通知，则必须通过另一个读取操作来完成。当连接会话过期时，客户端将与服务器断开连接，相关的watches也将被删除。

1. **redis/zk节点宕机如何处理\***

解决方法是**连接从服务器，做save操作**。将会在从服务器的data目录保存一份从服务器最新的dump.rdb文件。将这份dump.rdb文件拷贝到主服务器的data目录下。再重启主服务器。

1. **分布式集群下如何做到唯一序列号\***

<http://stor.51cto.com/art/201711/558600.htm>

**1、利用数据库递增，全数据库唯一。**

优点：明显，可控。

缺点：单库单表，数据库压力大。

**2、UUID， 生成的是length=32的16进制格式的字符串，如果回退为byte数组共16个byte元素，即UUID是一个128bit长的数字，一般用16进制表示。**

优点：对数据库压力减轻了。

缺点：但是排序怎么办？

此外还有UUID的变种，增加一个时间拼接，但是会造成id非常长。

**3、twitter在把存储系统从MySQL迁移到Cassandra的过程中由于Cassandra没有顺序ID生成机制，于是自己开发了一套全局唯一ID生成服务：Snowflake。**

41位的时间序列（精确到毫秒，41位的长度可以使用69年）

10位的机器标识（10位的长度最多支持部署1024个节点）

12位的计数顺序号（12位的计数顺序号支持每个节点每毫秒产生4096个ID序号） 最高位是符号位，始终为0。

优点：高性能，低延迟；独立的应用；按时间有序。

缺点：需要独立的开发和部署。

**4、Redis生成ID**

当使用数据库来生成ID性能不够要求的时候，我们可以尝试使用Redis来生成ID。这主要依赖于Redis是单线程的，所以也可以用生成全局唯一的ID。可以用Redis的原子操作INCR和INCRBY来实现。

可以使用Redis集群来获取更高的吞吐量。假如一个集群中有5台Redis。可以初始化每台Redis的值分别是1,2,3,4,5，然后步长都是5。

1. **用过哪些MQ，怎么用的，和其他mq比较有什么优缺点，MQ的连接是线程安全的吗\***

**可靠消费**

Redis：没有相应的机制保证消息的消费，当消费者消费失败的时候，消息体丢失，需要手动处理  
RabbitMQ：具有消息消费确认，即使消费者消费失败，也会自动使消息体返回原队列，同时可全程持久化，保证消息体被正确消费

**可靠发布**

Reids：不提供，需自行实现  
RabbitMQ：具有发布确认功能，保证消息被发布到服务器

**高可用**

Redis：采用主从模式，读写分离，但是故障转移还没有非常完善的官方解决方案  
RabbitMQ：集群采用磁盘、内存节点，任意单点故障都不会影响整个队列的操作

**持久化**

Redis：将整个Redis实例持久化到磁盘  
RabbitMQ：队列，消息，都可以选择是否持久化

**消费者负载均衡**

Redis：不提供，需自行实现  
RabbitMQ：根据消费者情况，进行消息的均衡分发

**队列监控**

Redis：不提供，需自行实现  
RabbitMQ：后台可以监控某个队列的所有信息，（内存，磁盘，消费者，生产者，速率等）

**流量控制**

Redis：不提供，需自行实现  
RabbitMQ：服务器过载的情况，对生产者速率会进行限制，保证服务可靠性

kafka，

activemq，

RocketMQ

1. MQ系统的数据如何保证不丢失
2. 列举出你能想到的数据库分库分表策略；分库分表后，如何解决全表查询

１、中间变量　＝ user\_id%（库数量\*每个库的表数量）;

２、库序号　＝　取整（中间变量／每个库的表数量）;

３、表序号　＝　中间变量％每个库的表数量;

Join连表查询，或者多sql查询

**系统架构**

<https://www.cnblogs.com/vindia/category/1082005.html>

1. 如何搭建一个高可用系统

容灾

集群

主备

熔断

限流

监控

降级

日志

分库分表

读写分离

Nginx反向代理

CDN加速

SSD硬盘持久化

自动备份

1. 哪些设计模式可以增加系统的可扩展性

工厂模式

抽象工厂模式

观察者模式：很方便增加观察者，方便系统扩展

模板方法模式：很方便的实现不稳定的扩展点，完成功能的重用

适配器模式：可以很方便地对适配其他接口

代理模式：可以很方便在原来功能的基础上增加功能或者逻辑

责任链模式：可以很方便得增加拦截器/过滤器实现对数据的处理，比如struts2的责任链

策略模式：通过新增策略从而改变原来的执行策略

1. 介绍设计模式，如模板模式，命令模式，策略模式，适配器模式、桥接模式、装饰模式，观察者模式，状态模式，访问者模式。
2. 抽象能力，怎么提高研发效率。

[人生](http://www.jiyifa.com/rensheng/)无处不记忆，  
随时随地可记忆。  
不要抱怨没时间，  
是你自己在偷懒。  
不要抱怨路太难，  
天下没有免费餐。  
勤学巧练加恒心，  
快忆好似活神仙。

我们需要解决的问题，我们需要通过程序来解决这些问题

如何将问题抽象成计算机可以识别逻辑，通过抽象能力，把现实生活中的问题域转化成计算机中可以识别的抽象问题，然后就可以通过计算机中的处理方式来解决现实生活中的问题

1. **什么是高内聚低耦合，请举例子如何实现\***

同一类功能放一块，如utils包，但是各个模块功能不依赖关联，这就是低耦合

接口、继承、多态也是低耦合的实现

1. 什么情况用接口，什么情况用消息

接口的特点是同步调用，接口实时响应，阻塞等待

消息的特点是异步处理，非实时响应，消息发送后则返回，消息队列可以削峰

一般对实时性要求比较高的功能采用接口

对实时性要求不高的功能可以采用消息，削峰时可以采用消息

1. **如果AB两个系统互相依赖，如何解除依赖\***

A--->B，同时B--->A

解除这种双向依赖的话，需要在AB之外增加一个C，用C封装A依赖的B的那部分功能，让A改为依赖C，C依赖B

然后就是这样

A--->C，C---->B，B--->A

不过这样依然存在环路依赖

1. 如何写一篇设计文档，目录是什么

基于此我认为主要会分为8个部分。分别为 **项目背景，项目目标，需求分析，方案对比，概要设计，详细设计（存储模型设计，接口设计），开发以及上线计划，方案排期**。

1. 什么场景应该拆分系统，什么场景应该合并系统

**拆分系统：**

当系统通过集群的方式已经无法解决性能问题的时候，或者业务扩展到很大的时候，需要把拆分系统

**按照业务的方式垂直拆分：**将业务功能结合比较紧密的部分拆分成独立的系统，独立维护

**按照性能瓶颈点拆分：**将系统性能瓶颈点拆分出一个独立的系统，可以针对这个独立的系统集群部署，增加可伸缩性，提高系统整体的性能

**合并系统：**

**或者系统间通过跨进程访问的性能损耗过高，可以将系统合并成一个系统，减少跨进程访问的消耗**

1. 系统和模块的区别，分别在什么场景下使用

系统和模块

**系统**是一个完整功能的系统，拥有独立的访问方式，和部署方式，拥有完整的生命周期，系统由模块组成

**模块**是系统的组成部分，不能单独工作，需要依附于系统才能发挥作用，通常是解决一定场景下的问题

系统用于**系统性解决问题**的方案

模块是针对**单个问题方面**的解决方案

1. 实战能力
2. 有没有处理过线上问题？出现内存泄露，CPU利用率标高，应用无响应时如何处理的。
3. 开发中有没有遇到什么技术问题？如何解决的
4. 如果有几十亿的白名单，每天白天需要高并发查询，晚上需要更新一次，如何设计这个功能。
5. 新浪微博是如何实现把微博推给订阅者
6. Google是如何在一秒内把搜索结果返回给用户的。
7. 12306网站的订票系统如何实现，如何保证不会票不被超卖。
8. 如何实现一个秒杀系统，保证只有几位用户能买到某件商品。

设计这个系统是一个考虑全面的问题，可以发散出很多问题，考察很多方面，不是仅仅回答通过redis的自减操作完成

比如简单的方案：

1，页面开启倒计时，要保证不能把**下单接口**暴露过早暴露出来，防止机器刷下单接口

2，前端限流，比如nginx对下单接口限流，命中限流则**返回302到秒杀页**

3，后端单独部署，**独立域名和nginx**，与线上正常运行的系统隔离开来，避免影响到线上环境

4，由于生成订单操作比较耗时，采用队列的方式来解耦下单成功和生成订单，针对进入后端的请求，采用redis自减，针对自减结果>0的请求则认为下单成功，触发一个生成订单的消息，然后立即返回给用户结果

5，用户方面，针对秒杀成功有两种处理方式

　　a，用户端收到秒杀成功的结果，则开启提示页面，并进入**倒计时**，倒计时时间为订单生成的预估时间

　　b，秒杀成功后，给当前用户在redis中生成一个**订单生成状态的标识**，用户端开启提示页面，loading，并轮询后端订单生成状态，生成成功之后让前端跳转到订单页面

6，订单服务订阅下单系统发送的消息，并开始生成订单，生成订单成功之后更新redis中用户秒杀订单的状态为已生成订单

**系统应该有页面和接口**

页面用于展示用户界面，接口用于获取数据

界面：秒杀页面，秒杀成功页面，秒杀失败页面，命中限流页面（查看订单页面不算秒杀系统的功能）

接口：秒杀下单接口，秒杀成功获取订单生成状态接口

**LINUX**

1. 硬链接和软连接区别

原理上，**硬链接和源文件的inode节点号相同**，两者互为硬链接。软连接和源文件的inode节点号不同，进而指向的block也不同，软连接block中存放了源文件的路径名。

实际上，**硬链接和源文件是同一份文件**，而**软连接是独立的文件，类似于快捷方式**，存储着源文件的位置信息便于指向。

使用限制上，不能对目录创建硬链接，不能对不同文件系统创建硬链接，不能对不存在的文件创建硬链接；可以对目录创建软连接，可以跨文件系统创建软连接，可以对不存在的文件创建软连接。

1. kill用法，某个进程杀不掉的原因（进入内核态，忽略kill信号）

做过Linux开发的人通常遇到过一个进程不能kill掉的情况，即使使用的是kill -9方式，而一般的教课书都只说kill -9能杀死任何进程，遇到这种情况时就会感觉到很矛盾，其它这也是正常的，通常有两种情况是不能kill掉的：

一是进程已经成为**僵死进程**，当它的父进程将它回收或将它的父进程kill掉即可在ps输出看不到了；

二是进程正处在**内核状态**中，Linux进程运行时分内核和用户两种状态，当进程进入内核状态后，会屏蔽所有信号，包括SIGKIL，所以这个时候kill -9也变得无效了

1. linux用过的命令

tar –zxvf

rm

ls

grep

du –sh

1. 系统管理命令（如查看内存使用、网络情况）

[cat /proc/meminfo](http://www.adminmimi.com/)

netstat -anp

1. 管道的使用 |

ps –ef |grep python

1. grep的使用，一定要掌握，每次都会问在文件中查找

当前目录递归查找 grep 文件 –nr .

1. shell脚本

.sh比较简单，百度写

1. find命令

在/home目录下查找以.txt结尾的文件名

find /home -name "\*.txt"

1. awk使用

awk是一种报表生成器，就是对文件进行格式化处理的，这里的格式化不是文件系统的格式化，而是对文件内容进行各种“排版”，进而格式化显示。

**TCP/IP**

1. **OSI与TCP/IP各层的结构与功能，都有哪些协议。\***

<https://blog.csdn.net/qq_35503221/article/details/80333899>

**OSI模型—应用层，表示层，回话层，传输层，网络层，数据链路层，物理层**

1.**物理层**：主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流（就是由1、0转化为电流强弱来进行传输,到达目的地后在转化为1、0，也就是我们常说的数模转换与模数转换）。这一层的数据叫做比特。

2.**数据链路层**：定义了如何让格式化数据以进行传输，以及如何让控制对物理介质的访问。这一层通常还提供错误检测和纠正，以确保数据的可靠传输。

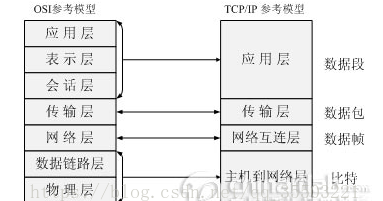
3.**网络层**：在位于不同地理位置的网络中的两个主机系统之间提供连接和路径选择。Internet的发展使得从世界各站点访问信息的用户数大大增加，而网络层正是管理这种连接的层。

4.**传输层**：定义了一些传输数据的协议和端口号（WWW端口80等），如：TCP（传输控制协议，传输效率低，可靠性强，用于传输可靠性要求高，数据量大的数据），UDP（[用户数据报协议](http://baike.baidu.com/view/468464.htm)，与TCP特性恰恰相反，用于传输可靠性要求不高，数据量小的数据，如QQ聊天数据就是通过这种方式传输的）。 主要是将从下层接收的数据进行分段和传输，到达目的地址后再进行重组。常常把这一层数据叫做段。

5.**会话层**：通过传输层（端口号：传输端口与接收端口）建立数据传输的通路。主要在你的系统之间发起会话或者接受会话请求（设备之间需要互相认识可以是IP也可以是MAC或者是主机名）

6.**表示层**：可确保一个系统的应用层所发送的信息可以被另一个系统的应用层读取。例如，PC程序与另一台计算机进行通信，其中一台计算机使用扩展二一十进制交换码（EBCDIC），而另一台则使用美国信息交换标准码（ASCII）来表示相同的字符。如有必要，表示层会通过使用一种通格式来实现多种数据格式之间的转换。

7.**应用层**： 是最靠近用户的OSI层。这一层为用户的应用程序（例如电子邮件、文件传输和终端仿真）提供网络服务。



1. **TCP与UDP的区别。\***

TCP(Transmission Control Protocol)：传输控制协议

UDP(User Datagram Protocol)：用户数据报协议

TCP是面向连接的、可靠的、有序的、速度慢的协议；

UDP是无连接的、不可靠的、无序的、速度快的协议。

TCP开销比UDP大，TCP头部需要20字节，UDP头部只要8个字节。

TCP无界有拥塞控制，UDP有界无拥塞控制。



1. TCP报文结构。

<https://blog.csdn.net/mary19920410/article/details/58030147>

**1、端口号**：用来标识同一台计算机的不同的应用进程。

**1）源端口**：源端口和IP地址的作用是标识报文的返回地址。

**2）目的端口**：端口指明接收方计算机上的应用程序接口。

TCP报头中的源端口号和目的端口号同IP数据报中的源IP与目的IP唯一确定一条TCP连接。

**2、序号和确认号**：是TCP可靠传输的关键部分。**序号**是本报文段发送的数据组的第一个字节的序号。在TCP传送的流中，每一个字节一个序号。e.g.一个报文段的序号为300，此报文段数据部分共有100字节，则下一个报文段的序号为400。所以序号确保了TCP传输的有序性。确认号，即ACK，指明下一个期待收到的字节序号，表明该序号之前的所有数据已经正确无误的收到。确认号只有当ACK标志为1时才有效。比如建立连接时，SYN报文的ACK标志位为0。

**3、数据偏移／首部长度**：4bits。由于首部可能含有可选项内容，因此TCP报头的长度是不确定的，报头不包含任何任选字段则长度为20字节，4位首部长度字段所能表示的最大值为1111，转化为10进制为15，15\*32/8 = 60，故报头最大长度为60字节。首部长度也叫数据偏移，是因为首部长度实际上指示了数据区在报文段中的起始偏移值。

**4、保留**：为将来定义新的用途保留，现在一般置0。

**5、控制位**：URG  ACK  PSH  RST  SYN  FIN，共6个，每一个标志位表示一个控制功能。

**1）URG**：紧急指针标志，为1时表示紧急指针有效，为0则忽略紧急指针。

**2）ACK**：确认序号标志，为1时表示确认号有效，为0表示报文中不含确认信息，忽略确认号字段。

**3）PSH**：push标志，为1表示是带有push标志的数据，指示接收方在接收到该报文段以后，应尽快将这个报文段交给应用程序，而不是在缓冲区排队。

**4）RST**：重置连接标志，用于重置由于主机崩溃或其他原因而出现错误的连接。或者用于拒绝非法的报文段和拒绝连接请求。

**5）SYN**：同步序号，用于建立连接过程，在连接请求中，SYN=1和ACK=0表示该数据段没有使用捎带的确认域，而连接应答捎带一个确认，即SYN=1和ACK=1。

**6）FIN**：finish标志，用于释放连接，为1时表示发送方已经没有数据发送了，即关闭本方数据流。

**6、窗口**：滑动窗口大小，用来告知发送端接受端的缓存大小，以此控制发送端发送数据的速率，从而达到流量控制。窗口大小时一个16bit字段，因而窗口大小最大为65535。

**7、校验和**：奇偶校验，此校验和是对整个的*TCP*报文段，包括*TCP*头部和*TCP*数据，以*16*位字进行计算所得。由发送端计算和存储，并由接收端进行验证。

**8、紧急指针**：只有当*URG*标志置*1*时紧急指针才有效。紧急指针是一个正的偏移量，和顺序号字段中的值相加表示紧急数据最后一个字节的序号。*TCP*的紧急方式是发送端向另一端发送紧急数据的一种方式。

**9、选项和填充**：最常见的可选字段是最长报文大小，又称为MSS（Maximum Segment Size），每个连接方通常都在通信的第一个报文段（为建立连接而设置SYN标志为1的那个段）中指明这个选项，它表示本端所能接受的最大报文段的长度。选项长度不一定是32位的整数倍，所以要加填充位，即在这个字段中加入额外的零，以保证TCP头是32的整数倍。

**10、数据部分**：*TCP*报文段中的数据部分是可选的。在一个连接建立和一个连接终止时，双方交换的报文段仅有*TCP*首部。如果一方没有数据要发送，也使用没有任何数据的首部来确认收到的数据。在处理超时的许多情况中，也会发送不带任何数据的报文段。

1. TCP的三次握手与四次挥手过程，各个状态名称与含义，TIMEWAIT的作用。
2. TCP拥塞控制。

<https://blog.csdn.net/lishanmin11/article/details/77090316>

拥塞控制就是防止过多的数据注入网络中，这样可以使网络中的路由器或者链路不会导致过载。拥塞控制是一个全局性的过程，和流量控制不同，流量控制指点对点通信量的控制

**拥塞控制主要是四个算法：1）慢启动，2）拥塞避免，3）拥塞发生，4）快速恢复。**

**（1）在通信子网出现过多数据包的情况**，使得网络的性能下降，甚至不能正常工作，这种现象就称为拥塞。

**（2）网络拥塞的成因主要有三**：1、处理器的速度太慢。2、线路容量的限制。3、节点输出包的能力小于输入包的能力。

**（3）拥塞控制与流量控制是相关的**，流量控制在数据链路层对一条通信路径上的流量进行控制，其的是保证发送者的发送速度不超过接收者的接收速度，它只涉及一全发送者和一个接收者，是局部控制。拥塞控制是对整个通信子网的流量进行控制，其目的是保证通信子网中的流量与其资源相匹配，使子网不会出现性能下降和恶化、甚至崩溃，是全局控制。

**（4）拥塞控制的最终目标是**：1、防止由于过载而使吞吐量下降，损失效率；2、合理分配网络资源；3、避免死锁；4、匹配传输速度。

**（5）对拥塞控制**，可用的方法有两类：开环控制和闭环控制。  
1、开环控制的思想是通过良好的设计避免拥塞问题的出现，确保拥塞问题在开始时就不可能发生。开环控制方法包括何时接受新的通信何时丢弃包、丢弃哪些包。其特点是在作出决定时不考虑网络当前的状态。  
2、闭环控制的思想是反馈控制。即通过将网络工作的动态信息反馈给网络中节点的有关进程，节点根据网络当前的动态信息，调整转发数据包的策略。闭环控制过程包括三部分： ①监视系统  检测网络发生或将要发生拥塞的时间和地点。②报告  将监视中检测到的信息传送到可以进行拥塞控制的节点。③决策  调整系统的操作行为，以解决问题。

**（6）对应于开环控制的方法有**：（基于拥塞预防）

1、预定缓冲区 2、合理分配缓冲区 3、通信量整形法（A、许可证算法，B、漏桶算法，C、令牌漏桶算法。）

对应于闭环控制的方法有：（基于拥塞抑制，即拥塞出现或即将出现时采取适当的措施进行控制，直到消除拥塞）

1、阻塞包法。 2、负载丢弃法

1. TCP滑动窗口与回退N针协议。
2. Http的报文结构。

（1）HTTP请求报文  
一个HTTP请求报文由请求行（request line）、请求头部（header）、空行和请求数据4个部分组成，下图给出了请求报文的一般格式。



（2）HTTP响应也由三个部分组成，分别是：状态行、消息报头、响应正文。

1. Http的状态码含义。

1xx: 信息

|  |  |
| --- | --- |
| **消息：** | **描述：** |
| 100 Continue | 服务器仅接收到部分请求，但是一旦服务器并没有拒绝该请求，客户端应该继续发送其余的请求。 |
| 101 Switching Protocols | 服务器转换协议：服务器将遵从客户的请求转换到另外一种协议。 |
| 103 Checkpoint | 用于 PUT 或者 POST 请求恢复失败时的恢复请求建议。 |

2xx: 成功

|  |  |
| --- | --- |
| **消息：** | **描述：** |
| 200 OK | 请求成功（这是对HTTP请求成功的标准应答。） |
| 201 Created | 请求被创建完成，同时新的资源被创建。 |
| 202 Accepted | 供处理的请求已被接受，但是处理未完成。 |
| 203 Non-Authoritative Information | 请求已经被成功处理，但是一些应答头可能不正确，因为使用的是其他文档的拷贝。 |
| 204 No Content | 请求已经被成功处理，但是没有返回新文档。浏览器应该继续显示原来的文档。如果用户定期地刷新页面，而Servlet可以确定用户文档足够新，这个状态代码是很有用的。 |
| 205 Reset Content | 请求已经被成功处理，但是没有返回新文档。但浏览器应该重置它所显示的内容。用来强制浏览器清除表单输入内容。 |
| 206 Partial Content | 客户发送了一个带有Range头的GET请求，服务器完成了它。 |

3xx: 重定向

|  |  |
| --- | --- |
| **消息：** | **描述：** |
| 300 Multiple Choices | 多重选择。链接列表。用户可以选择某链接到达目的地。最多允许五个地址。 |
| 301 Moved Permanently | 所请求的页面已经转移至新的 URL 。 |
| 302 Found | 所请求的页面已经临时转移至新的 URL 。 |
| 303 See Other | 所请求的页面可在别的 URL 下被找到。 |
| 304 Not Modified | 未按预期修改文档。客户端有缓冲的文档并发出了一个条件性的请求（一般是提供If-Modified-Since头表示客户只想比指定日期更新的文档）。服务器告诉客户，原来缓冲的文档还可以继续使用。 |
| 305 Use Proxy | 客户请求的文档应该通过Location头所指明的代理服务器提取。 |
| 306 Switch Proxy | 目前已不再使用，但是代码依然被保留。 |
| 307 Temporary Redirect | 被请求的页面已经临时移至新的 URL 。 |
| 308 Resume Incomplete | 用于 PUT 或者 POST 请求恢复失败时的恢复请求建议。 |

4xx: 客户端错误

|  |  |
| --- | --- |
| **消息：** | **描述：** |
| 400 Bad Request | 因为语法错误，服务器未能理解请求。 |
| 401 Unauthorized | 合法请求，但对被请求页面的访问被禁止。因为被请求的页面需要身份验证，客户端没有提供或者身份验证失败。 |
| 402 Payment Required | 此代码尚无法使用。 |
| 403 Forbidden | 合法请求，但对被请求页面的访问被禁止。 |
| 404 Not Found | 服务器无法找到被请求的页面。 |
| 405 Method Not Allowed | 请求中指定的方法不被允许。 |
| 406 Not Acceptable | 服务器生成的响应无法被客户端所接受。 |
| 407 Proxy Authentication Required | 用户必须首先使用代理服务器进行验证，这样请求才会被处理。 |
| 408 Request Timeout | 请求超出了服务器的等待时间。 |
| 409 Conflict | 由于冲突，请求无法被完成。 |
| 410 Gone | 被请求的页面不可用。 |
| 411 Length Required | "Content-Length" 未被定义。如果无此内容，服务器不会接受请求。 |
| 412 Precondition Failed | 请求中的前提条件被服务器评估为失败。 |
| 413 Request Entity Too Large | 由于所请求的实体太大，服务器不会接受请求。 |
| 414 Request-URI Too Long | 由于 URL 太长，服务器不会接受请求。当 POST 请求被转换为带有很长的查询信息的 GET 请求时，就会发生这种情况。 |
| 415 Unsupported Media Type | 由于媒介类型不被支持，服务器不会接受请求。 |
| 416 Requested Range Not Satisfiable | 客户端请求部分文档，但是服务器不能提供被请求的部分。 |
| 417 Expectation Failed | 服务器不能满足客户在请求中指定的请求头。 |

5xx: 服务器错误

|  |  |
| --- | --- |
| **消息：** | **描述：** |
| 500 Internal Server Error | 请求未完成。服务器遇到不可预知的情况。 |
| 501 Not Implemented | 请求未完成。服务器不支持所请求的功能，或者服务器无法完成请求。 |
| 502 Bad Gateway | 请求未完成。服务器充当网关或者代理的角色时，从上游服务器收到一个无效的响应。 |
| 503 Service Unavailable | 服务器当前不可用（过载或者当机）。 |
| 504 Gateway Timeout | 网关超时。服务器充当网关或者代理的角色时，未能从上游服务器收到一个及时的响应。 |
| 505 HTTP Version Not Supported | 服务器不支持请求中指明的HTTP协议版本。 |
| 511 Network Authentication Required | 用户需要提供身份验证来获取网络访问入口。 |

1. **Http?request的几种类型。\***

1.  OPTIONS：返回服务器针对特定资源所支持的HTTP请求方法。也可以利用向Web服务器发送'\*'的请求来测试服务器的功能性。

2.  HEAD：向服务器索要与GET请求相一致的响应，只不过响应体将不会被返回。这一方法可以在不必传输整个响应内容的情况下，就可以获取包含在响应消息头中的元信息。

3.  GET：向特定的资源发出请求。

4.  POST：向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的创建和/或已有资源的修改。

5.  PUT：向指定资源位置上传其最新内容。

6.  DELETE：请求服务器删除Request-URI所标识的资源。

7.  TRACE：回显服务器收到的请求，主要用于测试或诊断。

8.  CONNECT：HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。

1. **Http1.1和Http1.0的区别\***

**长连接**

HTTP 1.0需要使用keep-alive参数来告知服务器端要建立一个长连接，而HTTP1.1默认**支持长连接。**

HTTP是基于TCP/IP协议的，创建一个TCP连接是需要经过三次握手的,有一定的开销，如果每次通讯都要重新建立连接的话，对性能有影响。因此最好能维持一个长连接，可以用个长连接来发多个请求。

**节约带宽**

HTTP 1.1支持只发送header信息(不带任何body信息)，如果服务器认为客户端有权限请求服务器，则返回100，否则返回401。客户端如果接受到100，才开始把请求body发送到服务器。

这样当服务器返回401的时候，客户端就可以不用发送请求body了，节约了带宽。

另外HTTP还支持传送内容的一部分。这样当客户端已经有一部分的资源后，只需要跟服务器请求另外的部分资源即可。这是支持文件断点续传的基础。

**HOST域**

现在可以web server例如tomat，设置虚拟站点是非常常见的，也即是说，web server上的多个虚拟站点可以共享同一个ip和端口。

HTTP1.0是没有host域的，HTTP1.1才支持这个参数。

1. **Http怎么处理长连接。\***

在HTTP1.0和HTTP1.1协议中都有对长连接的支持。其中HTTP1.0需要在request中增加Connection： keep-alive header才能够支持，而HTTP1.1默认支持。

**http1.0请求与服务端的交互过程:**

（1）客户端发出带有包含一个header：”Connection： keep-alive“的请求

（2）服务端接收到这个请求后,根据http1.0和”Connection： keep-alive“判断出这是一个长连接,就会在response的header中也增加”Connection： keep-alive“，同时不会关闭已建立的tcp连接.

（3）客户端收到服务端的response后,发现其中包含”Connection： keep-alive“，就认为是一个长连接，不关闭这个连接。并用该连接再发送request.转到（1）

**http1.1请求与服务端的交互过程:**

（1）客户端发出http1.1的请求

（2）服务端收到http1.1后就认为这是一个长连接,会在返回的response设置Connection： keep-alive,同时不会关闭已建立的连接.

（3）客户端收到服务端的response后,发现其中包含”Connection： keep-alive“，就认为是一个长连接，不关闭这个连接。并用该连接再发送request.转到（1）

基于http协议的长连接减少了请求,减少了建立连接的时间,但是每次交互都是由客户端发起的,客户端发送消息,服务端才能返回客户端消息。

1. Cookie与Session的作用于原理。

**1.1 Cookie机制**

在程序中，会话跟踪是很重要的事情。理论上，一个用户的所有请求操作都应该属于同一个会话，而另一个用户的所有请求操作则应该属于另一个会话，二者不能混淆。例如，用户A在超市购买的任何商品都应该放在A的购物车内，不论是用户A什么时间购买的，这都是属于同一个会话的，不能放入用户B或用户C的购物车内，这不属于同一个会话。

而Web应用程序是使用HTTP协议传输数据的。HTTP协议是无状态的协议。一旦数据交换完毕，客户端与服务器端的连接就会关闭，再次交换数据需要建立新的连接。这就意味着服务器无法从连接上跟踪会话。即用户A购买了一件商品放入购物车内，当再次购买商品时服务器已经无法判断该购买行为是属于用户A的会话还是用户B的会话了。要跟踪该会话，必须引入一种机制。

Cookie就是这样的一种机制。它可以弥补HTTP协议无状态的不足。在Session出现之前，基本上所有的网站都采用Cookie来跟踪会话。

**1.2 Session机制**

Session是另一种记录客户状态的机制，不同的是Cookie保存在客户端浏览器中，而Session保存在服务器上。客户端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上。这就是Session。客户端浏览器再次访问时只需要从该Session中查找该客户的状态就可以了。

如果说Cookie机制是通过检查客户身上的“通行证”来确定客户身份的话，那么Session机制就是通过检查服务器上的“客户明细表”来确认客户身份。Session相当于程序在服务器上建立的一份客户档案，客户来访的时候只需要查询客户档案表就可以了。

Java Web规范支持通过配置的方式禁用Cookie。下面举例说一下怎样通过配置禁止使用Cookie，可以使用重定向url。   
cookie数据保存在客户端，session数据保存在服务器端   
第一次请求服务器，生成session和sessionID，sessionID用cookie保存   
第二次请求服务器，携带seesionID，服务器从请求中取出sessionID

1. 电脑上访问一个网页，整个过程是怎么样的：DNS、HTTP、TCP、OSPF、IP、ARP。

**应用层：**

连接：当我们输入一个url请求时，首先要建立socket连接，因为socket是通过ip和端口建立的，所有有一个DNS解析的过程，首先我们知道我们本地的机器上在配置网络时都会填写DNS，这样本机就会把这个url发给这个配置的DNS服务器，如果能够找到相应的url则返回其ip，否则该DNS将继续将该解析请求发送给上级DNS，整个DNS可以看做是一个树状结构，该请求将一直发送到根直到得到结果。现在已经拥有了目标ip和端口号，这样我们就可以打开socket连接了。

请求：连接成功建立后，开始向web服务器发送请求，这个请求一般是GET或POST命令（POST用于FORM参数的传递）。GET命令的格式为：　　GET 路径/文件名 HTTP/1.0

文件名指出所访问的文件，HTTP/1.0指出Web浏览器使用的HTTP版本。现在可以发送GET命令：

GET /mytest/index.html HTTP/1.0，

应用层：

**1. DNS（53）：**

我们输入的是一个URL需要转化成IP地址。首先我们知道我们本地的机器上在配置网络时都会填写DNS，这样本机就会把这个url发给这个配置的DNS服务器，如果能够找到相应的url则返回其ip，否则该DNS将继续将该解析请求发送给上级DNS，整个DNS可以看做是一个树状结构，该请求将一直发送到根直到得到结果。

**2. HTTP（80）**

HTTP协议的主要职责是生成针对目标web服务器的http请求报文（请求行、请求头部）

传输层

**3. TCP**

将http请求报文分割成报文段，按序号分为多个报文段。（三次握手）

网络层

**4. IP**

搜索目标的地址，一边中转一边传送。（路由）

**5. ARP**

因为最终都要在数据链路层上进行传输，而数据链路层并不认识IP地址，所以ARP的职责就是把IP地址转换成数据链路层认识的MAC地址。

通过数据链路层到达目标机器之后

网络层

**6. RARP**

这其实是ARP的逆过程，将MAC地址转换成Ip地址

传输层

**7. TCP**

将接收到的报文段按序号进行重组。

应用层

**8. HTTP**

1. Ping的整个过程。ICMP报文是什么。

ICMP（网际控制报文协议）：用来测试网络层是不是有故障，若有故障，该协议还能报告故障。Ping命令来使用这个协议

<https://blog.csdn.net/guoweimelon/article/details/50859658>

1. **C/S模式下使用socket通信，几个关键函数。\***

<https://blog.csdn.net/suky520/article/details/39137507>

client ： socket（ip，端口）

socket.close();

server：serversocket（端口）

socket = server.accept()

1. IP地址分类。

**IP地址分类（A类 B类 C类 D类 E类）**

     IP地址由四段组成，每个字段是一个字节，8位，最大值是255，,

     IP地址由两部分组成，即[网络地址](http://baike.baidu.com/view/547479.htm)和[主机地址](http://baike.baidu.com/view/547482.htm)。网络地址表示其属于互联网的哪一个网络，主机地址表示其属于该网络中的哪一台主机。二者是主从关系。

  IP地址的四大类型标识的是网络中的某台主机。IPv4的地址长度为32位，共4个字节，但实际中我们用[点分十进制](http://baike.baidu.com/view/828066.htm)记法。

     IP地址根据网络号和主机号来分，分为A、B、C三类及特殊地址D、E。    全0和全1的都保留不用。

**A类：**（**1.**0.0.0-**126.**0.0.0）（默认子网掩码：255.0.0.0或 0xFF000000）

     第一个字节为网络号，后三个字节为主机号。该类IP地址的最前面为“0”，所以地址的网络号取值于1~126之间。

     一般用于大型网络。

**B类：**（**128.1.**0.0-**191.255.**0.0）（默认子网掩码：255.255.0.0或0xFFFF0000）

     前两个字节为网络号，后两个字节为主机号。该类IP地址的最前面为“10”，所以地址的网络号取值于128~191之间。

     一般用于中等规模网络。

**C类：**（192.0.1.0-223.255.255.0）（子网掩码：255.255.255.0或 0xFFFFFF00）

     前三个字节为网络号，最后一个字节为主机号。该类IP地址的最前面为“110”，所以地址的网络号取值于192~223之间。

     一般用于小型网络。

**D类：**是多播地址。该类IP地址的最前面为“1110”，所以地址的网络号取值于224~239之间。一般用于多路广播用户[1]  。

**E类：**是保留地址。该类IP地址的最前面为“1111”，所以地址的网络号取值于240~255之间。

**回送地址：**127.0.0.1。 也是本机地址，等效于**localhost**或**本机IP**。

       一般用于测试使用。例如：ping 127.0.0.1来测试本机TCP/IP是否正常。

1. 路由器与交换机区别。

一、工作所在的OSI层次不一样（根本区别，导致接下来的区别）   
交换机工作在 OSI模型的数据链路层，所以工作原理比较简单；   
路由器工作在OSI模型的网络层，具备更多的网络协议信息，所以可以做出更好的数据转发策略。

二、数据转发所依据的对象也不一样。   
交换机工作在数据链路层，所以交换机转发数据依靠的是每个物理地址（MAC地址），MAC地址一般是设备生产商在设备出厂时固定在设备中的，不能进行更改。   
路由器工作在网络层，所以其交换数据依靠网络地址（IP地址），而IP地址是由网络管理员自己分配或者系统自动获取的。

三、是否可以分割广播域   
由交换机连接的所有端口仍然属于同一个广播域，所以极有可能会产生数据拥堵；   
连接到路由器上的所有端口不在属于同一个广播域，所以不会产生类似的数据拥堵问题。

1. 网络其实大体分为两块，一个TCP协议，一个HTTP协议，只要把这两块以及相关协议搞清楚，一般问题不大。
2. 推荐书籍：《TCP/IP协议族》

**软能力**

1. 如何学习一项新技术，比如如何学习Java的，重点学习什么
2. 有关注哪些新的技术
3. 工作任务非常多非常杂时如何处理
4. 项目出现延迟如何处理
5. 和同事的设计思路不一样怎么处理
6. 如何保证开发质量
7. 职业规划是什么？短期，长期目标是什么
8. 团队的规划是什么
9. 能介绍下从工作到现在自己的成长在那里
10. 说说你认为的服务端开发工程师应该具备哪些能力
11. 网络必备，高并发，JVM必会，各种分布式技术，看源码的能力。
12. 说说你认为的架构师是什么样的，架构师主要做什么。

**自我问答总结**

[https://blog.csdn.net/u010697681/article/details/79414112#%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1%E7%89%B9%E5%BE%81](https://blog.csdn.net/u010697681/article/details/79414112%23%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1%E7%89%B9%E5%BE%81)

**1．JDK,JRE,JVM(掌握)**

(1)JVM

保证Java语言跨平台。针对不同的操心系统提供不同的JVM。

问题：java语言是跨平台的吗?JVM是跨平台的吗?

(2)JRE

java程序的运行环境。包括JVM和核心类库

(3)JDK

java开发环境。包括JRE和开发工具(javac,java)

(4)一个Java程序的开发流程

A:编写Java源程序

B:通过javac命令编译java程序，生成字节码文件

C:通过java命令运行字节码文件

**2．主从数据库切换**

<https://blog.csdn.net/u012881904/article/details/77449710>

一般使用多个dataSource，然后创建多个SessionFactory，入侵明显，修改多，session处理比较麻烦。

合适的方案使用AbstractRoutingDataSource实现类通过AOP或者手动处理实现动态使用我们的数据源，入侵低。determineTargetDataSource --

determineCurrentLookupKey

Why：如果主库故障，可以切换从库

Why：针对与mysql，怎么保证主从同步，怎么通知代码切换

**3. Bean获取它所在的Spring容器，可以让该Bean实现ApplicationContextAware接口，Bean获取它所在的Spring容器，可以通过这个上下文环境对象得到Spring容器中的Bean。**

Why：针对于非web项目的spring

**4. ehcache:**

[**https://blog.csdn.net/vbirdbest/article/details/72763048**](https://blog.csdn.net/vbirdbest/article/details/72763048)

**5. StringBuffer是线程安全的，每次操作字符串，String会生成一个新的对象，而StringBuffer不会；StringBuilder是非线程安全的**

String：字符串常量 每当用String操作字符串时，实际上是在不断的创建新的对象，而原来的对象就会变为垃圾被ＧＣ回收掉，

　　　StringBuffer：字符串变量 是可改变的对象，每当我们用它们对字符串做操作时，实际上是在一个对象上操作的

　　　StringBuilder：字符串变量 是可改变的对象，每当我们用它们对字符串做操作时，实际上是在一个对象上操作的

String 字符串常量 线程安全 操作少量数据

StringBuffer 字符串变量 线程安全 操作大量数据 速度慢 多线程适合用

StringBuilder 字符串变量 线程不安全 操作大量数据 速度快 单线程适合用

<https://www.cnblogs.com/java1024/p/7685400.html>

String str = new String("xyz");创建了几个对象。

如果String常量池中，已经创建了"xyz",则不会继续创建，此时只创建了一个对象new String("xyz");

如果String常量池中没有创建"xyz",则会创建两个对象，一个对象的值是"xyz",一个对象是new String("xyz");

**6．关系型数据库和非关系型数据库种类和关系**

|  |  |  |  |
| --- | --- | --- | --- |
| 数据库 类型 | 特性 | 优点 | 缺点 |
| 关系型数据库 SQLite、Oracle、mysql | 1、关系型数据库，是指采用了关系模型来组织 数据的数据库； 2、关系型数据库的最大特点就是事务的一致性； 3、简单来说，关系模型指的就是二维表格模型， 而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。 | 1、容易理解：二维表结构是非常贴近逻辑世界一个概念，关系模型相对网状、层次等其他模型来说更容易理解； 2、使用方便：通用的SQL语言使得操作关系型数据库非常方便； 3、易于维护：丰富的完整性(实体完整性、参照完整性和用户定义的完整性)大大减低了数据冗余和数据不一致的概率； 4、支持SQL，可用于复杂的查询。 | 1、为了维护一致性所付出的巨大代价就是其读写性能比较差； 2、固定的表结构； 3、高并发读写需求； 4、海量数据的高效率读写； |
| 非关系型数据库 MongoDb、redis、HBase | 1、使用键值对存储数据； 2、分布式； 3、一般不支持ACID特性； 4、非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。 | 1、无需经过sql层的解析，读写性能很高； 2、基于键值对，数据没有耦合性，容易扩展； 3、存储数据的格式：nosql的存储格式是key,value形式、文档形式、图片形式等等，文档形式、图片形式等等，而关系型数据库则只支持基础类型。 | 1、不提供sql支持，学习和使用成本较高； 2、无事务处理，附加功能bi和报表等支持也不好； |

7. Vector ,ArrayList 和LinkedList的区别？

 1、Vector、ArrayList都是以类似数组的形式存储在内存中，LinkedList则以链表的形式进行存储。

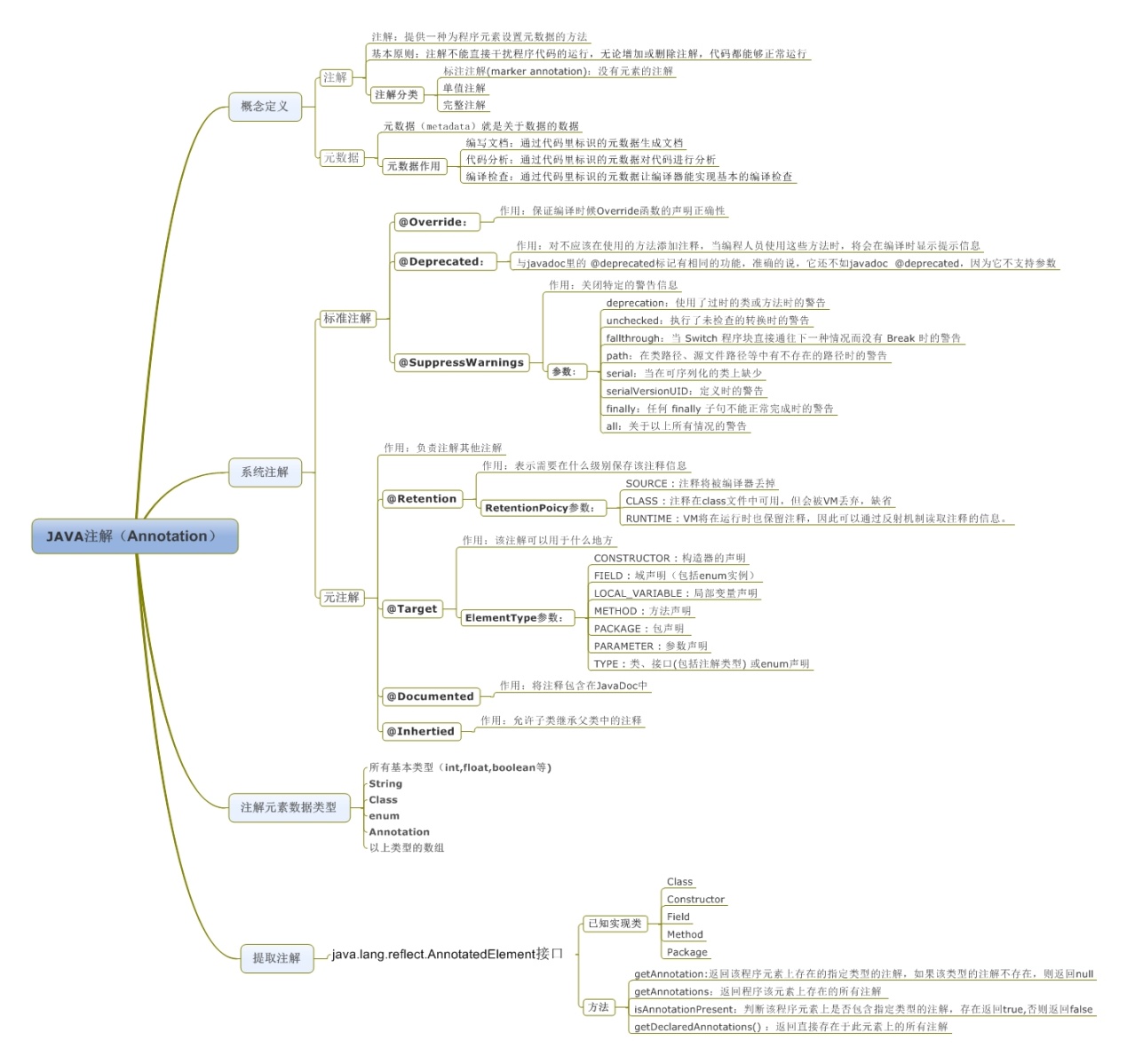
 2、Vector线程同步，ArrayList、LinkedList线程不同步。

 3、LinkedList适合指定位置插入、删除操作，不适合查找；ArrayList、Vector适合查找，不适合指定位置的插入、删除操作。

 4、ArrayList在元素填满容器时会自动扩充容器大小的50%，而Vector则是100%，因此ArrayList更节省。

**8.注解**

登陆、权限拦截、日志处理，以及各种Java框架，如Spring，Hibernate，JUnit 提到注解就不能不说反射，Java自定义注解是通过运行时靠反射获取注解。实际开发中，例如我们要获取某个方法的调用日志，可以通过AOP（动态代理机制）给方法添加切面，通过反射来获取方法包含的注解，如果包含日志注解，就进行日志记录。



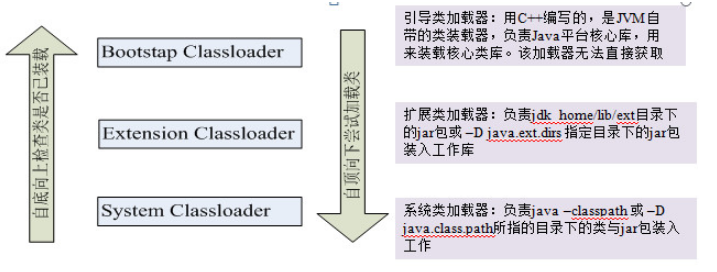
**9.反射**

反射就是把java类中的各种成分映射成一个个的Java对象

可以获取类的相关信息，可以进行设置，可以代理

spring 的 ioc/di 也是反射....  
javaBean和jsp之间调用也是反射....  
struts的 FormBean 和页面之间...也是通过反射调用....  
JDBC 的 classForName()也是反射.....  
hibernate的 find(Class clazz) 也是反射....

**10.加载器**



**11.ajax**

运用XMLHttpRequest或新的Fetch API与网页服务器进行异步资料交换；

**12.final, finally, finalize 的区别**

1. final   
   修饰符（关键字）如果一个类被声明为final，意味着它不能再派生出新的子类，不能作为父类被继承。因此一个类不能既被声明为 abstract的，又被声明为final的。将变量或方法声明为final，可以保证它们在使用中不被改变。被声明为final的变量必须在声明时给定初值，而在以后的引用中只能读取，不可修改。被声明为final的方法也同样只能使用，不能重载。
2. finally   
   在异常处理时提供 finally 块来执行任何清除操作。如果抛出一个异常，那么相匹配的 catch 子句就会执行，然后控制就会进入 finally 块（如果有的话）。
3. finalize   
   方法名。Java 技术允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。它是在 Object 类中定义的，因此所有的类都继承了它。子类覆盖 finalize() 方法以整理系统资源或者执行其他清理工作。finalize() 方法是在垃圾收集器删除对象之前对这个对象调用的。

Finalize()方法是对象逃脱死亡命运的最后一次机会，稍后GC对F-Queue中的对象进行第二次小规模的标记，如果对象要在finalize()中成功拯救自己—只要重新与引用链上的任何一个对象建立关联即可，譬如把自己(this关键字)赋值给摸个类变量或对象的成员变量，那在第二次标记时它将被移除“即将回收”的集合.

* 1. 对象可以在被GC时自我拯救
  2. 这种自救的机会只有一次，因为一个对象的finalize()方法最多只会被系统自动调用一次

#### 13．重载和重写的区别

* override（重写）

　　 1. 方法名、参数、返回值相同。

　　 2. 子类方法不能缩小父类方法的访问权限。

　　 3. 子类方法不能抛出比父类方法更多的异常(但子类方法可以不抛出异常)。

　　 4. 存在于父类和子类之间。

　　 5. 方法被定义为final不能被重写。

* overload（重载）

　　1. 参数类型、个数、顺序至少有一个不相同。

　　2. 不能重载只有返回值不同的方法名。

　　3. 存在于父类和子类、同类中。

#### 14. 抽象类和接口有什么区别

接口是公开的，里面不能有私有的方法或变量，是用于让别人使用的，而抽象类是可以有私有方法或私有变量的，   
另外，实现接口的一定要实现接口里定义的所有方法，而实现抽象类可以有选择地重写需要用到的方法，一般的应用里，最顶级的是接口，然后是抽象类实现接口，最后才到具体类实现。   
还有，接口可以实现多重继承，而一个类只能继承一个超类，但可以通过继承多个接口实现多重继承，接口还有标识（里面没有任何方法，如Remote接口）和数据共享（里面的变量全是常量）的作用。

**15. statement**

1、执行静态SQL语句。通常通过Statement实例实现。   
2、执行动态SQL语句。通常通过PreparedStatement实例实现。   
3、执行数据库存储过程。通常通过CallableStatement实例实现。

**16. 数据库基础查询**

**1、 加载JDBC驱动程序：**   
在连接数据库之前，首先要加载想要连接的数据库的驱动到JVM（Java虚拟机），   
这通过java.lang.Class类的静态方法forName(String className)实现。   
例如：

try{

//加载MySql的驱动类

Class.forName("com.mysql.jdbc.Driver") ;

}catch(ClassNotFoundException e){

System.out.println("找不到驱动程序类 ，加载驱动失败！");

e.printStackTrace() ;

}

成功加载后，会将Driver类的实例注册到DriverManager类中。

**2、 提供JDBC连接的URL**   
- 连接URL定义了连接数据库时的协议、子协议、数据源标识。   
- 书写形式：协议：子协议：数据源标识

协议：在JDBC中总是以jdbc开始 子协议：是桥连接的驱动程序或是数据库管理系统名称。   
数据源标识：标记找到数据库来源的地址与连接端口。   
例如：   
jdbc:mysql://localhost:3306/test?useUnicode=true&characterEncoding=gbk;useUnicode=true;（MySql的连接URL）   
表示使用Unicode字符集。如果characterEncoding设置为 gb2312或GBK，本参数必须设置为true 。characterEncoding=gbk：字符编码方式。

**3、创建数据库的连接**   
- 要连接数据库，需要向java.sql.DriverManager请求并获得Connection对象， 该对象就代表一个数据库的连接。   
- 使用DriverManager的getConnectin(String url , String username , String password )方法传入指定的欲连接的数据库的路径、数据库的用户名和 密码来获得。

例如： //连接MySql数据库，用户名和密码都是root

String url = "jdbc:mysql://localhost:3306/test" ;

String username = "root" ;

String password = "root" ;

try{

Connection con = DriverManager.getConnection(url , username , password ) ;

}catch(SQLException se){

System.out.println("数据库连接失败！");

se.printStackTrace() ;

}

**4、 创建一个Statement**   
•要执行SQL语句，必须获得java.sql.Statement实例，Statement实例分为以下3 种类型：   
1、执行静态SQL语句。通常通过Statement实例实现。   
2、执行动态SQL语句。通常通过PreparedStatement实例实现。   
3、执行数据库存储过程。通常通过CallableStatement实例实现。   
具体的实现方式：   
Statement stmt = con.createStatement() ; PreparedStatement pstmt = con.prepareStatement(sql) ; CallableStatement cstmt = con.prepareCall(“{CALL demoSp(? , ?)}”) ;   
**5、执行SQL语句**   
Statement接口提供了三种执行SQL语句的方法：executeQuery 、executeUpdate 和execute   
1、ResultSet executeQuery(String sqlString)：执行查询数据库的SQL语句 ，返回一个结果集（ResultSet）对象。   
2、int executeUpdate(String sqlString)：用于执行INSERT、UPDATE或 DELETE语句以及SQL DDL语句，如：CREATE TABLE和DROP TABLE等   
3、execute(sqlString):用于执行返回多个结果集、多个更新计数或二者组合的 语句。 具体实现的代码：   
ResultSet rs = stmt.executeQuery(“SELECT \* FROM …”) ; int rows = stmt.executeUpdate(“INSERT INTO …”) ; boolean flag = stmt.execute(String sql) ;   
**6、处理结果**   
两种情况：   
1、执行更新返回的是本次操作影响到的记录数。   
2、执行查询返回的结果是一个ResultSet对象。   
• ResultSet包含符合SQL语句中条件的所有行，并且它通过一套get方法提供了对这些 行中数据的访问。   
• 使用结果集（ResultSet）对象的访问方法获取数据：   
while(rs.next()){   
String name = rs.getString(“name”) ;   
String pass = rs.getString(1) ; // 此方法比较高效   
}   
（列是从左到右编号的，并且从列1开始）

**7、关闭JDBC对象**   
操作完成以后要把所有使用的JDBC对象全都关闭，以释放JDBC资源，关闭顺序和声 明顺序相反：   
1、关闭记录集   
2、关闭声明   
3、关闭连接对象

if(rs != null){ // 关闭记录集

try{

rs.close() ;

}catch(SQLException e){

e.printStackTrace() ;

}

}

if(stmt != null){ // 关闭声明

try{

stmt.close() ;

}catch(SQLException e){

e.printStackTrace() ;

}

}

if(conn != null){ // 关闭连接对象

try{

conn.close() ;

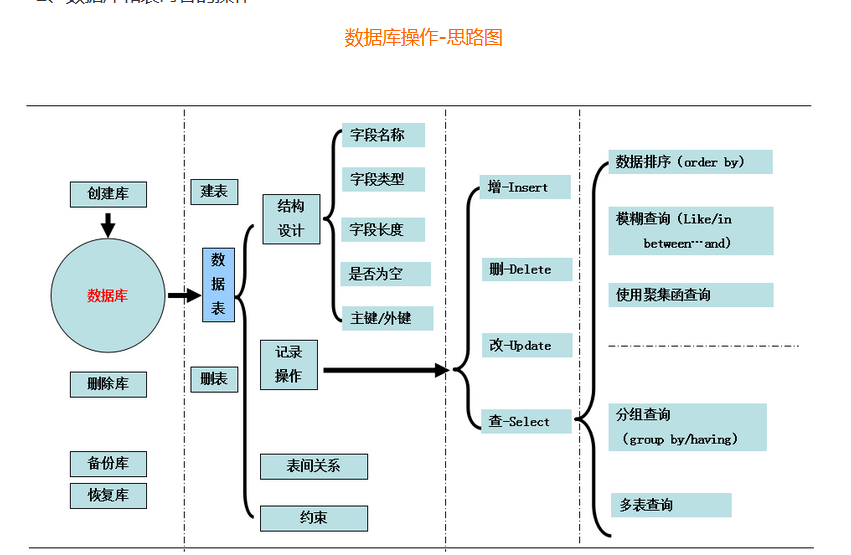
}catch(SQLException e){

e.printStackTrace() ;

}

}

具体请看笔记。



**17.使用Spring框架的好处是什么？**

**轻量：**Spring 是轻量的，基本的版本大约2MB。

**控制反转：**Spring通过控制反转实现了松散耦合，对象们给出它们的依赖，而不是创建或查找依赖的对象们。

**面向切面的编程(AOP)：**Spring支持面向切面的编程，并且把应用业务逻辑和系统服务分开。

**容器：**Spring 包含并管理应用中对象的生命周期和配置。

**MVC框架**：Spring的WEB框架是个精心设计的框架，是Web框架的一个很好的替代品。

**事务管理：**Spring 提供一个持续的事务管理接口，可以扩展到上至本地事务下至全局事务（JTA）。

**异常处理：**Spring 提供方便的API把具体技术相关的异常（比如由JDBC，Hibernate or JDO抛出的）转化为一致的unchecked 异常。

**18.** ~**8等于多少？8>>>2等于多少？**

**第一个答案是-9，第二个答案是2，**无符号右移高位补0 。

**19. 子类能否重写父类的静态方法**

不能，类对象，从属于对应的类。

**20. 什么是线程？**

  线程是操作系统能够进行运算调度的最小单位，它被包含在进程之中，是进程中的实际运作单位。程序员可以通过它进行多处理器编程，你可以使用多线程对运算密集型任务提速。比如，如果一个线程完成一个任务要100毫秒，那么用十个线程完成改任务只需10毫秒。

**21. 线程和进程有什么区别？**

  线程是进程的子集，一个进程可以有很多线程，每条线程并行执行不同的任务。不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。每个线程都拥有单独的栈内存用来存储本地数据。

**22. 如何在Java中实现线程？**

  两种方式：java.lang.Thread 类的实例就是一个线程但是它需要调用java.lang.Runnable接口来执行，由于线程类本身就是调用的Runnable接口所以你可以继承java.lang.Thread 类或者直接调用Runnable接口来重写run()方法实现线程。

**23. Java 关键字volatile 与 synchronized 作用与区别？**

1，volatile  
    它所修饰的变量不保留拷贝，直接访问主内存中的。

2，synchronized

    当它用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码。

**24.不同的线程生命周期？**

  当我们在Java程序中新建一个线程时，它的状态是New。当我们调用线程的start()方法时，状态被改变为Runnable。线程调度器会为Runnable线程池中的线程分配CPU时间并且讲它们的状态改变为Running。其他的线程状态还有Waiting，Blocked 和Dead。

#### **25. 程优先级的理解是什么？**

  每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于线程调度的实现，这个实现是和操作系统相关的(OS dependent)。我们可以定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个int变量(从1-10)，1代表最低优先级，10代表最高优先级。

**26.是死锁(Deadlock)？如何分析和避免死锁？**

  死锁是指两个以上的线程永远阻塞的情况，这种情况产生至少需要两个以上的线程和两个以上的资源。

  分析死锁，我们需要查看Java应用程序的线程转储。我们需要找出那些状态为BLOCKED的线程和他们等待的资源。每个资源都有一个唯一的id，用这个id我们可以找出哪些线程已经拥有了它的对象锁。

  避免嵌套锁，只在需要的地方使用锁和避免无限期等待是避免死锁的通常办法。

#### 27.么是线程安全？Vector是一个线程安全类吗？

#### 所在的进程中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。如果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。一个线程安全的计数器类的同一个实例对象在被多个线程使用的情况下也不会出现计算失误。很显然你可以将集合类分成两组，线程安全和非线程安全的。Vector 是用同步方法来实现线程安全的, 而和它相似的ArrayList不是线程安全的。

**28.Java中如何停止一个线程？**

  Java提供了很丰富的API但没有为停止线程提供API。JDK 1.0本来有一些像stop(), suspend() 和 resume()的控制方法但是由于潜在的死锁威胁因此在后续的JDK版本中他们被弃用了，之后Java API的设计者就没有提供一个兼容且线程安全的方法来停止一个线程。当run() 或者 call() 方法执行完的时候线程会自动结束,如果要手动结束一个线程，你可以用volatile 布尔变量来退出run()方法的循环或者是取消任务来中断线程

**29.什么是ThreadLocal?**

  ThreadLocal用于创建线程的本地变量，我们知道一个对象的所有线程会共享它的全局变量，所以这些变量不是线程安全的，我们可以使用同步技术。但是当我们不想使用同步的时候，我们可以选择ThreadLocal变量。

  每个线程都会拥有他们自己的Thread变量，它们可以使用get()\set()方法去获取他们的默认值或者在线程内部改变他们的值。ThreadLocal实例通常是希望它们同线程状态关联起来是private static属性。

**30.Sleep()、suspend()和wait()之间有什么区别？**

  Thread.sleep()使当前线程在指定的时间处于“非运行”（Not Runnable）状态。线程一直持有对象的监视器。比如一个线程当前在一个同步块或同步方法中，其它线程不能进入该块或方法中。如果另一线程调用了interrupt()方法，它将唤醒那个“睡眠的”线程。

  注意：sleep()是一个静态方法。这意味着只对当前线程有效，一个常见的错误是调用t.sleep()，（这里的t是一个不同于当前线程的线程）。即便是执行t.sleep()，也是当前线程进入睡眠，而不是t线程。t.suspend()是过时的方法，使用suspend()导致线程进入停滞状态，该线程会一直持有对象的监视器，suspend()容易引起死锁问题。

  object.wait()使当前线程出于“不可运行”状态，和sleep()不同的是wait是object的方法而不是thread。调用object.wait()时，线程先要获取这个对象的对象锁，当前线程必须在锁对象保持同步，把当前线程添加到等待队列中，随后另一线程可以同步同一个对象锁来调用object.notify()，这样将唤醒原来等待中的线程，然后释放该锁。基本上wait()/notify()与sleep()/interrupt()类似，只是前者需要获取对象锁。

**31.什么是线程饿死，什么是活锁？**

  当所有线程阻塞，或者由于需要的资源无效而不能处理，不存在非阻塞线程使资源可用。JavaAPI中线程活锁可能发生在以下情形：

  1，当所有线程在程序中执行Object.wait(0)，参数为0的wait方法。程序将发生活锁直到在相应的对象上有线程调用Object.notify()或者Object.notifyAll()。

  2，当所有线程卡在无限循环中。

**32.什么是Java Timer类？如何创建一个有特定时间间隔的任务？**

  java.util.Timer是一个工具类，可以用于安排一个线程在未来的某个特定时间执行。Timer类可以用安排一次性任务或者周期任务。

  java.util.TimerTask是一个实现了Runnable接口的抽象类，我们需要去继承这个类来创建我们自己的定时任务并使用Timer去安排它的执行。

#### 33.Java中的同步集合与并发集合有什么区别？

  同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，不过并发集合的可扩展性更高。

  在Java1.5之前程序员们只有同步集合来用且在多线程并发的时候会导致争用，阻碍了系统的扩展性。

  Java5介绍了并发集合像ConcurrentHashMap，不仅提供线程安全还用锁分离和内部分区等现代技术提高了可扩展性。

#### 34.同步方法和同步块，哪个是更好的选择？

  同步块是更好的选择，因为它不会锁住整个对象（当然你也可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。

#### 35.么是线程池？ 为什么要使用它？

  创建线程要花费昂贵的资源和时间，如果任务来了才创建线程那么响应时间会变长，而且一个进程能创建的线程数有限。

  为了避免这些问题，在程序启动的时候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程。

  从JDK1.5开始，Java API提供了Executor框架让你可以创建不同的线程池。比如单线程池，每次处理一个任务；数目固定的线程池或者是缓存线程池（一个适合很多生存期短的任务的程序的可扩展线程池）。

#### 36．java中invokeAndWait 和 invokeLater有什么区别？

  这两个方法是Swing API 提供给Java开发者用来从当前线程而不是事件派发线程更新GUI组件用的。InvokeAndWait()同步更新GUI组件，比如一个进度条，一旦进度更新了，进度条也要做出相应改变。如果进度被多个线程跟踪，那么就调用invokeAndWait()方法请求事件派发线程对组件进行相应更新。而invokeLater()方法是异步调用更新组件的。

#### 37. 多线程中的忙循环是什么?

  忙循环就是程序员用循环让一个线程等待，不像传统方法wait(), sleep() 或 yield() 它们都放弃了CPU控制，而忙循环不会放弃CPU，它就是在运行一个空循环。这么做的目的是为了保留CPU缓存。

  在多核系统中，一个等待线程醒来的时候可能会在另一个内核运行，这样会重建缓存。为了避免重建缓存和减少等待重建的时间就可以使用它了。

**38. Array不可以用泛型？**

是的，list的可以，推荐用list，List可以提供编译期的类型安全保证，而Array却不能。

int num=Integer.valueOf("12");

int num2=Integer.parseInt("12");

double num3=Double.valueOf("12.2");

double num4=Double.parseDouble("12.2");

//其他的类似。通过基本数据类型的包装来的valueOf和parseXX来实现String转为XX

String a=String.valueOf("1234");//这里括号中几乎可以是任何类型

String b=String.valueOf(true);

String c=new Integer(12).toString();//通过包装类的toString()也可以

String d=new Double(2.3).toString();

**39. AJAX有哪些有点和缺点？**

优点：

  1、最大的一点是页面无刷新，用户的体验非常好。

  2、使用异步方式与服务器通信，具有更加迅速的响应能力。

  3、可以把以前一些服务器负担的工作转嫁到客户端，利用客户端闲置的能力来处理，减轻服务器和带宽的负担，节约空间和宽带租用成本。并且减轻服务器的负担，ajax的原则是“按需取数据”，可以最大程度的减少冗余请求，和响应对服务器造成的负担。

   4、基于标准化的并被广泛支持的技术，不需要下载插件或者小程序。

缺点：

   1、ajax不支持浏览器back按钮。

   2、安全问题 AJAX暴露了与服务器交互的细节。

   3、对搜索引擎的支持比较弱。

   4、破坏了程序的异常机制。

   5、不容易调试。

**40.集合解析**

**List 和 Set 区别**

List,Set都是继承自Collection接口

List特点：元素有放入顺序，元素可重复

Set特点：元素无放入顺序，元素不可重复，重复元素会覆盖掉

（注意：元素虽然无放入顺序，但是元素在set中的位置是有该元素的HashCode决定的，其位置其实是固定的，加入Set 的Object必须定义equals()方法 ，另外list支持for循环，也就是通过下标来遍历，也可以用迭代器，但是set只能用迭代，因为他无序，无法用下标来取得想要的值。）

**Set和List对比：**

Set：检索元素效率低下，删除和插入效率高，插入和删除不会引起元素位置改变。

List：和数组类似，List可以动态增长，查找元素效率高，插入删除元素效率低，因为会引起其他元素位置改变。

**List 和 Map 区别**

List是对象集合，允许对象重复。

Map是键值对的集合，不允许key重复。

Arraylist 与 LinkedList 区别

**Arraylist：**

优点：ArrayList是实现了基于动态数组的数据结构,因为地址连续，一旦数据存储好了，查询操作效率会比较高（在内存里是连着放的）。

缺点：因为地址连续， ArrayList要移动数据,所以插入和删除操作效率比较低。

**LinkedList**：

优点：LinkedList基于链表的数据结构,地址是任意的，所以在开辟内存空间的时候不需要等一个连续的地址，对于新增和删除操作add和remove，LinedList比较占优势。LinkedList 适用于要头尾操作或插入指定位置的场景

缺点：因为LinkedList要移动指针,所以查询操作性能比较低。

适用场景分析：

当需要对数据进行对此访问的情况下选用ArrayList，当需要对数据进行多次增加删除修改时采用LinkedList。

ArrayList 与 Vector 区别

public ArrayList(int initialCapacity)//构造一个具有指定初始容量的空列表。

public ArrayList()//构造一个初始容量为10的空列表。

public ArrayList(Collection<? extends E> c)//构造一个包含指定 collection 的元素的列表

Vector有四个构造方法：

public Vector()//使用指定的初始容量和等于零的容量增量构造一个空向量。

public Vector(int initialCapacity)//构造一个空向量，使其内部数据数组的大小，其标准容量增量为零。

public Vector(Collection<? extends E> c)//构造一个包含指定 collection 中的元素的向量

public Vector(int initialCapacity,int capacityIncrement)//使用指定的初始容量和容量增量构造一个空的向量

**ArrayList和Vector都是用数组实现的，主要有这么三个区别：**

Vector是多线程安全的，线程安全就是说多线程访问同一代码，不会产生不确定的结果。而ArrayList不是，这个可以从源码中看出，Vector类中的方法很多有synchronized进行修饰，这样就导致了Vector在效率上无法与ArrayList相比；

两个都是采用的线性连续空间存储元素，但是当空间不足的时候，两个类的增加方式是不同。

Vector可以设置增长因子，而ArrayList不可以。

Vector是一种老的动态数组，是线程同步的，效率很低，一般不赞成使用。

**适用场景分析：**

Vector是线程同步的，所以它也是线程安全的，而ArrayList是线程异步的，是不安全的。如果不考虑到线程的安全因素，一般用ArrayList效率比较高。

如果集合中的元素的数目大于目前集合数组的长度时，在集合中使用数据量比较大的数据，用Vector有一定的优势。

**HashMap 和 Hashtable 的区别**

1.hashMap去掉了HashTable 的contains方法，但是加上了containsValue（）和containsKey（）方法。

2.hashTable同步的，而HashMap是非同步的，效率上逼hashTable要高。

3.hashMap允许空键值，而hashTable不允许。

注意：   
TreeMap：非线程安全基于红黑树实现。TreeMap没有调优选项，因为该树总处于平衡状态。

Treemap：适用于按自然顺序或自定义顺序遍历键(key)。

参考：<http://blog.csdn.net/qq_22118507/article/details/51576319>

**HashSet 和 HashMap 区别**

set是线性结构，set中的值不能重复，hashset是set的hash实现，hashset中值不能重复是用hashmap的key来实现的。

map是键值对映射，可以空键空值。HashMap是Map接口的hash实现，key的唯一性是通过key值hash值的唯一来确定，value值是则是链表结构。

他们的共同点都是hash算法实现的唯一性，他们都不能持有基本类型，只能持有对象

**HashMap 和 ConcurrentHashMap 的区别**

ConcurrentHashMap是线程安全的HashMap的实现。

（1）ConcurrentHashMap对整个桶数组进行了分割分段(Segment)，然后在每一个分段上都用lock锁进行保护，相对于HashTable的syn关键字锁的粒度更精细了一些，并发性能更好，而HashMap没有锁机制，不是线程安全的。

（2）HashMap的键值对允许有null，但是ConCurrentHashMap都不允许。

**ConcurrentHashMap 的工作原理及代码实现**

HashTable里使用的是synchronized关键字，这其实是对对象加锁，锁住的都是对象整体，当Hashtable的大小增加到一定的时候，性能会急剧下降，因为迭代时需要被锁定很长的时间。

ConcurrentHashMap算是对上述问题的优化，其构造函数如下，默认传入的是16，0.75，16。

ConcurrentHashMap引入了分割(Segment)，上面代码中的最后一行其实就可以理解为把一个大的Map拆分成N个小的HashTable，在put方法中，会根据hash(paramK.hashCode())来决定具体存放进哪个Segment，如果查看Segment的put操作，我们会发现内部使用的同步机制是基于lock操作的，这样就可以对Map的一部分（Segment）进行上锁，这样影响的只是将要放入同一个Segment的元素的put操作，保证同步的时候，锁住的不是整个Map（HashTable就是这么做的），相对于HashTable提高了多线程环境下的性能，因此HashTable已经被淘汰了。

**41.线程解析**

创建线程的方式及实现

**Java中创建线程主要有三种方式**

1.继承Thread类创建线程类

2.通过Runnable接口创建线程类

3.通过Callable和Future创建线程

（1）创建Callable接口的实现类，并实现call()方法，该call()方法将作为线程执行体，并且有返回值。

（2）创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值。

（3）使用FutureTask对象作为Thread对象的target创建并启动新线程。

（4）调用FutureTask对象的get()方法来获得子线程执行结束后的返回值

package com.thread;

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.FutureTask;

public class CallableThreadTest implements Callable<Integer>

{

public static void main(String[] args)

{

CallableThreadTest ctt = new CallableThreadTest();

FutureTask<Integer> ft = new FutureTask<>(ctt);

for(int i = 0;i < 100;i++)

{

System.out.println(Thread.currentThread().getName()+" 的循环变量i的值"+i);

if(i==20)

{

new Thread(ft,"有返回值的线程").start();

}

}

try

{

System.out.println("子线程的返回值："+ft.get());

} catch (InterruptedException e)

{

e.printStackTrace();

} catch (ExecutionException e)

{

e.printStackTrace();

}

}

@Override

public Integer call() throws Exception

{

int i = 0;

for(;i<100;i++)

{

System.out.println(Thread.currentThread().getName()+" "+i);

}

return i;

}

}

**创建线程的三种方式的对比**

采用实现Runnable、Callable接口的方式创见多线程时，优势是：

线程类只是实现了Runnable接口或Callable接口，还可以继承其他类。

在这种方式下，多个线程可以共享同一个target对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

劣势是：

编程稍微复杂，如果要访问当前线程，则必须使用Thread.currentThread()方法。

使用继承Thread类的方式创建多线程时优势是：

编写简单，如果需要访问当前线程，则无需使用Thread.currentThread()方法，直接使用this即可获得当前线程。

劣势是：

线程类已经继承了Thread类，所以不能再继承其他父类。

**42.** [**Java并发编程：CountDownLatch、CyclicBarrier和Semaphore**](https://www.cnblogs.com/dolphin0520/p/3920397.html)

**CountDownLatch用法**

CountDownLatch类位于java.util.concurrent包下，利用它可以实现类似计数器的功能。比如有一个任务A，它要等待其他4个任务执行完毕之后才能执行，此时就可以利用CountDownLatch来实现这种功能了。

**CyclicBarrier用法**

字面意思回环栅栏，通过它可以实现让一组线程等待至某个状态之后再全部同时执行。叫做回环是因为当所有等待线程都被释放以后，CyclicBarrier可以被重用。我们暂且把这个状态就叫做barrier，当调用await()方法之后，线程就处于barrier了。

**Semaphore用法**

Semaphore翻译成字面意思为 信号量，Semaphore可以控同时访问的线程个数，通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可。

**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 CountDownLatch 与 CyclicBarrier 区别**

| **CountDownLatch** | **CyclicBarrier** |
| --- | --- |
| 减计数方式 | 加计数方式 |
| 计算为0时释放所有等待的线程 | 计数达到指定值时释放所有等待线程 |
| 计数为0时，无法重置 | 计数达到指定值时，计数置为0重新开始 |
| 调用countDown()方法计数减一，调用await()方法只进行阻塞，对计数没任何影响 | 调用await()方法计数加1，若加1后的值不等于构造方法的值，则线程阻塞 |
| 不可重复利用 | 可重复利用 |

[java.util.concurrent.Exchanger应用范例与原理浅析](http://lixuanbin.iteye.com/blog/2166772)

* 此类提供对外的操作是同步的；
* 用于成对出现的线程之间交换数据；
* 可以视作双向的同步队列；
* 可应用于基因算法、流水线设计等场景。

**ThreadLocal 原理分析**

ThreadLocal，很多地方叫做线程本地变量，也有些地方叫做线程本地存储，其实意思差不多。可能很多朋友都知道ThreadLocal为变量在每个线程中都创建了一个副本，那么每个线程可以访问自己内部的副本变量。

在每个线程Thread内部有一个ThreadLocal.ThreadLocalMap类型的成员变量threadLocals，这个threadLocals就是用来存储实际的变量副本的，键值为当前ThreadLocal变量，value为变量副本（即T类型的变量）。

初始时，在Thread里面，threadLocals为空，当通过ThreadLocal变量调用get()方法或者set()方法，就会对Thread类中的threadLocals进行初始化，并且以当前ThreadLocal变量为键值，以ThreadLocal要保存的副本变量为value，存到threadLocals。

然后在当前线程里面，如果要使用副本变量，就可以通过get方法在threadLocals里面查找。

最常见的ThreadLocal使用场景为 用来解决 数据库连接、Session管理等。

**讲讲线程池的实现原理**

**线程池的几种方式**

**newFixedThreadPool(int nThreads)**   
创建一个固定长度的线程池，每当提交一个任务就创建一个线程，直到达到线程池的最大数量，这时线程规模将不再变化，当线程发生未预期的错误而结束时，线程池会补充一个新的线程

**newCachedThreadPool()**   
创建一个可缓存的线程池，如果线程池的规模超过了处理需求，将自动回收空闲线程，而当需求增加时，则可以自动添加新线程，线程池的规模不存在任何限制

**newSingleThreadExecutor()**   
这是一个单线程的Executor，它创建单个工作线程来执行任务，如果这个线程异常结束，会创建一个新的来替代它；它的特点是能确保依照任务在队列中的顺序来串行执行

**newScheduledThreadPool(int corePoolSize)**   
创建了一个固定长度的线程池，而且以延迟或定时的方式来执行任务，类似于Timer。

**线程的生命周期**

**新建**(New)、**就绪**（Runnable）、**运行**（Running）、**阻塞**(Blocked)和**死亡**(Dead)**5种状态**

(1)生命周期的五种状态

**新建（new Thread）**   
当创建Thread类的一个实例（对象）时，此线程进入新建状态（未被启动）。   
例如：Thread t1=new Thread();

**就绪（runnable）**   
线程已经被启动，正在等待被分配给CPU时间片，也就是说此时线程正在就绪队列中排队等候得到CPU资源。例如：**t1.start();**

**运行（running）**   
线程获得CPU资源正在执行任务（run()方法），此时除非此线程自动放弃CPU资源或者有优先级更高的线程进入，线程将一直运行到结束。

**死亡（dead）**   
当线程执行完毕或被其它线程杀死，线程就进入死亡状态，这时线程不可能再进入就绪状态等待执行。

自然终止：正常运行run()方法后终止

异常终止：调用**stop()**方法让一个线程终止运行

**堵塞（blocked）**   
由于某种原因导致正在运行的线程让出CPU并暂停自己的执行，即进入堵塞状态。

正在睡眠：用sleep(long t) 方法可使线程进入睡眠方式。一个睡眠着的线程在指定的时间过去可进入就绪状态。

正在等待：调用wait()方法。（调用motify()方法回到就绪状态）

被另一个线程所阻塞：调用suspend()方法。（调用resume()方法恢复）

**43. 锁机制**

#### 说说线程安全问题

线程安全是指要控制多个线程对某个资源的有序访问或修改，而在这些线程之间没有产生冲突。   
在Java里，线程安全一般体现在两个方面：   
1、多个thread对同一个java实例的访问（read和modify）不会相互干扰，它主要体现在关键字synchronized。如ArrayList和Vector，HashMap和Hashtable（后者每个方法前都有synchronized关键字）。如果你在interator一个List对象时，其它线程remove一个element，问题就出现了。   
2、每个线程都有自己的字段，而不会在多个线程之间共享。它主要体现在java.lang.ThreadLocal类，而没有Java关键字支持，如像static、transient那样。

Volatile

在多线程并发编程中synchronized和Volatile都扮演着重要的角色，Volatile是**轻量级的synchronized**，它在多处理器开发中保证了共享变量的“可见性”。可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。

在多个线程之间能够被共享的变量被称为共享变量。共享变量包括所有的实例变量，静态变量和数组元素。他们都被存放在堆内存中，Volatile只作用于**共享变量**。

**悲观锁 乐观锁**

乐观锁 悲观锁   
是一种思想。可以用在很多方面。

比如数据库方面。   
悲观锁就是for update（锁定查询的行）   
乐观锁就是 version字段（比较跟上一次的版本号，如果一样则更新，如果失败则要重复读-比较-写的操作。）

JDK方面：   
悲观锁就是sync   
乐观锁就是原子类（内部使用CAS实现）

本质来说，就是悲观锁认为总会有人抢我的。   
乐观锁就认为，基本没人抢。

乐观锁（Optimistic Lock）：   
每次获取数据的时候，都不会担心数据被修改，所以每次获取数据的时候都不会进行加锁，但是在更新数据的时候需要判断该数据是否被别人修改过。如果数据被其他线程修改，则不进行数据更新，如果数据没有被其他线程修改，则进行数据更新。由于数据没有进行加锁，期间该数据可以被其他线程进行读写操作。

乐观锁：比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。

**数据存储分析**

[mysql索引使用技巧及注意事项](https://www.cnblogs.com/heyonggang/p/6610526.html)

 INSERT 与 UPDATE 语句在拥有索引的表中执行会花费更多的时间，而SELECT 语句却会执行得更快。这是因为，在进行插入或更新时，数据库也需要插入或更新索引值。

索引的类型：

UNIQUE(唯一索引)：不可以出现相同的值，可以有NULL值

INDEX(普通索引)：允许出现相同的索引内容

PROMARY KEY(主键索引)：不允许出现相同的值

fulltext index(全文索引)：可以针对值中的某个单词，但效率确实不敢恭维

组合索引：实质上是将多个字段建到一个索引里，列值的组合必须唯一

## (1)使用ALTER TABLE语句创建索性

ALTER TABLE 表名 ADD 索引类型 （unique,primary key,fulltext,index）[索引名]（字段名）

## (2)使用CREATE INDEX语句对表增加索引

CREATE INDEX index\_name ON table\_name(username(length));

//create只能添加这两种索引;

CREATE INDEX index\_name ON table\_name (column\_list)

CREATE UNIQUE INDEX index\_name ON table\_name (column\_list)

## (3)删除索引

删除索引可以使用ALTER TABLE或DROP INDEX语句来实现。DROP INDEX可以在ALTER TABLE内部作为一条语句处理，其格式如下：

drop index index\_name on table\_name ;

alter table table\_name drop index index\_name ;

alter table table\_name drop primary key ;

## (4) 组合索引与前缀索引

create table USER\_DEMO

(

ID int not null auto\_increment comment '主键',

LOGIN\_NAME varchar(**100) not null comment '登录名',**

**PASSWORD varchar(100) not null comment '密码',**

**CITY varchar(30) not null comment '城市',**

**AGE int not null comment '年龄',**

**SEX int not null comment '性别(0:女 1：男)',**

**primary key (ID)**

**);**

ALTER TABLE USER\_DEMO ADD INDEX name\_city\_age (LOGIN\_NAME(**16),CITY,AGE);**

**索引的使用及注意事项**

Explain select \* from user where id=**1;**

    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。

**分表与分库使用场景以及设计方式**

**对于访问极为频繁且数据量巨大的单表来说，我们首先要做的就是减少单表的记录条数，以便减少数据查询所需要的时间，提高数据库的吞吐，这就是所谓的分表！**

场景：分表能够解决单表数据量过大带来的查询效率下降的问题，但是，却无法给数据库的并发处理能力带来质的提升。面对高并发的读写访问，当数据库master

服务器无法承载写操作压力时，不管如何扩展slave服务器，此时都没有意义了。

因此，我们必须换一种思路，对数据库进行拆分，从而提高数据库写入能力，这就是所谓的分库!

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

**存储引擎的 InnoDB 与 MyISAM**

1）InnoDB支持事务，MyISAM不支持，这一点是非常之重要。事务是一种高级的处理方式，如在一些列增删改中只要哪个出错还可以回滚还原，而MyISAM就不可以了。

2）MyISAM适合查询以及插入为主的应用，InnoDB适合频繁修改以及涉及到安全性较高的应用

3）InnoDB支持外键，MyISAM不支持

4）从MySQL5.5.5以后，InnoDB是默认引擎

5）InnoDB不支持FULLTEXT类型的索引

6）InnoDB中不保存表的行数，如select count(*) from table时，InnoDB需要扫描一遍整个表来计算有多少行，但是MyISAM只要简单的读出保存好的行数即可。注意的是，当count(*)语句包含where条件时MyISAM也需要扫描整个表

7）对于自增长的字段，InnoDB中必须包含只有该字段的索引，但是在MyISAM表中可以和其他字段一起建立联合索引

8）清空整个表时，InnoDB是一行一行的删除，效率非常慢。MyISAM则会重建表

9）InnoDB支持行锁（某些情况下还是锁整表，如 update table set a=1 where user like ‘%lee%’

## 索引数据结构设相关的计算机原理

上文说过，二叉树、红黑树等数据结构也可以用来实现索引，但是文件系统及数据库系统普遍采用**B-/+Tree**作为索引结构，这一节将结合计算机组成原理相关知识讨论B-/+Tree作为索引的理论基础。

当一个数据库表过于庞大，LIMIT offset, length中的offset值过大，则SQL查询语句会非常缓慢，你需增加order by，并且order by字段需要建立索引。

   如果使用子查询去优化LIMIT的话，则子查询必须是连续的，某种意义来讲，子查询不应该有where条件，where会过滤数据，使数据失去连续性。

   如果你查询的记录比较大，并且数据传输量比较大，比如包含了text类型的field，则可以通过建立子查询。

[分布式系统唯一ID生成方案汇总](https://www.cnblogs.com/haoxinyue/p/5208136.html)

* 关系型数据库 MySQL

MySQL 是一个最流行的关系型数据库，在互联网产品中应用比较广泛。一般情况下，MySQL 数据库是选择的第一方案，基本上有 80% ~ 90% 的场景都是基于 MySQL 数据库的。因为，需要关系型数据库进行管理，此外，业务存在许多事务性的操作，需要保证事务的强一致性。同时，可能还存在一些复杂的 SQL 的查询。值得注意的是，前期尽量减少表的联合查询，便于后期数据量增大的情况下，做数据库的分库分表。

* 内存数据库 Redis

随着数据量的增长，MySQL 已经满足不了大型互联网类应用的需求。因此，Redis 基于内存存储数据，可以极大的提高查询性能，对产品在架构上很好的补充。例如，为了提高服务端接口的访问速度，尽可能将读频率高的热点数据存放在 Redis 中。这个是非常典型的以空间换时间的策略，使用更多的内存换取 CPU 资源，通过增加系统的内存消耗，来加快程序的运行速度。

在某些场景下，可以充分的利用 Redis 的特性，大大提高效率。这些场景包括缓存，会话缓存，时效性，访问频率，计数器，社交列表，记录用户判定信息，交集、并集和差集，热门列表与排行榜，最新动态等。

使用 Redis 做缓存的时候，需要考虑数据不一致与脏读、缓存更新机制、缓存可用性、缓存服务降级、缓存穿透、缓存预热等缓存使用问题。

* 文档数据库 MongoDB

MongoDB 是对传统关系型数据库的补充，它非常适合高伸缩性的场景，它是可扩展性的表结构。基于这点，可以将预期范围内，表结构可能会不断扩展的 MySQL 表结构，通过 MongoDB 来存储，这就可以保证表结构的扩展性。

此外，日志系统数据量特别大，如果用 MongoDB 数据库存储这些数据，利用分片集群支持海量数据，同时使用聚集分析和 MapReduce 的能力，是个很好的选择。

MongoDB 还适合存储大尺寸的数据，GridFS 存储方案就是基于 MongoDB 的分布式文件存储系统。

* 列族数据库 HBase

HBase 适合海量数据的存储与高性能实时查询，它是运行于 HDFS 文件系统之上，并且作为 MapReduce 分布式处理的目标数据库，以支撑离线分析型应用。在数据仓库、数据集市、商业智能等领域发挥了越来越多的作用，在数以千计的企业中支撑着大量的大数据分析场景的应用。

* 全文搜索引擎 ElasticSearch

在一般情况下，关系型数据库的模糊查询，都是通过 like 的方式进行查询。其中，like “value%” 可以使用索引，但是对于 like “%value%” 这样的方式，执行全表查询，这在数据量小的表，不存在性能问题，但是对于海量数据，全表扫描是非常可怕的事情。ElasticSearch 作为一个建立在全文搜索引擎 Apache Lucene 基础上的实时的分布式搜索和分析引擎，适用于处理实时搜索应用场景。此外，使用 ElasticSearch 全文搜索引擎，还可以支持多词条查询、匹配度与权重、自动联想、拼写纠错等高级功能。因此，可以使用 ElasticSearch 作为关系型数据库全文搜索的功能补充，将要进行全文搜索的数据缓存一份到 ElasticSearch 上，达到处理复杂的业务与提高查询速度的目的。

ElasticSearch 不仅仅适用于搜索场景，还非常适合日志处理与分析的场景。著名的 ELK 日志处理方案，由 ElasticSearch、Logstash 和 Kibana 三个组件组成，包括了日志收集、聚合、多维度查询、可视化显示等。

| **MongoDB 特性** | **优势** |
| --- | --- |
| 事务支持 | MongoDB 目前只支持单文档事务，需要复杂事务支持的场景暂时不适合 |
| 灵活的文档模型 | JSON 格式存储最接近真实对象模型，对开发者友好，方便快速开发迭代 |
| 高可用复制集 | 满足数据高可靠、服务高可用的需求，运维简单，故障自动切换 |
| 可扩展分片集群 | 海量数据存储，服务能力水平扩展 |
| 高性能 | mmapv1、wiredtiger、mongorocks（rocksdb）、in-memory 等多引擎支持满足各种场景需求 |
| 强大的索引支持 | 地理位置索引可用于构建 各种 O2O 应用、文本索引解决搜索的需求、TTL索引解决历史数据自动过期的需求 |
| Gridfs | 解决文件存储的需求 |
| aggregation & mapreduce | 解决数据分析场景需求，用户可以自己写查询语句或脚本，将请求都分发到 MongoDB 上完成 |

从目前[阿里云 MongoDB 云数据库](https://www.aliyun.com/product/mongodb)上的用户看，MongoDB 的应用已经渗透到各个领域，比如游戏、物流、电商、内容管理、社交、物联网、视频直播等，以下是几个实际的应用案例。

* 游戏场景，使用 MongoDB 存储游戏用户信息，用户的装备、积分等直接以内嵌文档的形式存储，方便查询、更新
* 物流场景，使用 MongoDB 存储订单信息，订单状态在运送过程中会不断更新，以 MongoDB 内嵌数组的形式来存储，一次查询就能将订单所有的变更读取出来。
* 社交场景，使用 MongoDB 存储存储用户信息，以及用户发表的朋友圈信息，通过地理位置索引实现附近的人、地点等功能
* 物联网场景，使用 MongoDB 存储所有接入的智能设备信息，以及设备汇报的日志信息，并对这些信息进行多维度的分析
* 视频直播，使用 MongoDB 存储用户信息、礼物信息等

[redis内部数据结构深入浅出](https://www.cnblogs.com/chenpingzhao/p/6965164.html)

redis 是 key-value 存储系统，其中 key 类型一般为字符串，而 value 类型则为 redis 对象（redis object），可以绑定各种类型的数据，譬如 string、list 和set，redis.h 中定义了 struct redisObject，它是一个简单优秀的数据结构

Redis 基于内存存储数据，可以极大的提高查询性能，对产品在架构上很好的补充。例如，为了提高服务端接口的访问速度，尽可能将读频率高的热点数据存放在 Redis 中。这个是非常典型的以空间换时间的策略，使用更多的内存换取 CPU 资源，通过增加系统的内存消耗，来加快程序的运行速度。

在某些场景下，可以充分的利用 Redis 的特性，大大提高效率。这些场景包括缓存，会话缓存，时效性，访问频率，计数器，社交列表，记录用户判定信息，交集、并集和差集，热门列表与排行榜，最新动态等。

使用 Redis 做缓存的时候，需要考虑数据不一致与脏读、缓存更新机制、缓存可用性、缓存服务降级、缓存穿透、缓存预热等缓存使用问题。

**Redis 如何实现持久化**

## 1. snapshotting(快照)

也是默认方式.(把数据做一个备份，将数据存储到文件)

快照是默认的持久化方式，这种方式是将内存中数据以快照的方式写到二进制文件中，默认的文件名称为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键修改就自动做快照.

## 2. Append-onlyfile(缩写aof)的方式

aof方式:由于快照方式是在一定间隔时间做一次的，所以如果redis意外down掉的话，就会丢失最后一次快照后的所有修改。aof比快照方式有更好的持久化性，是由于在使用aof时，redis会将每一个收到的写命令都通过write函数追加到文件中，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。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。

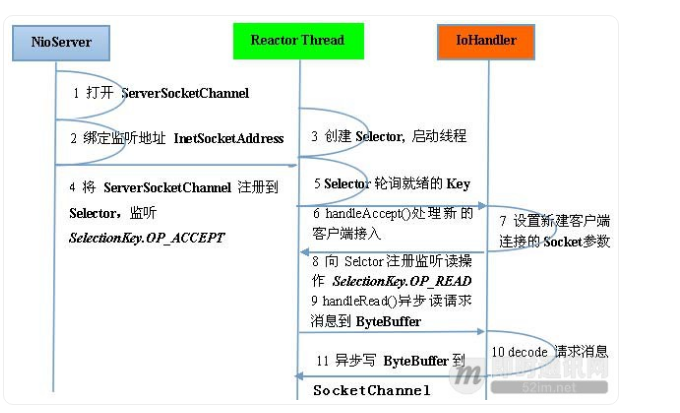
**Redis 为什么是单线程的**

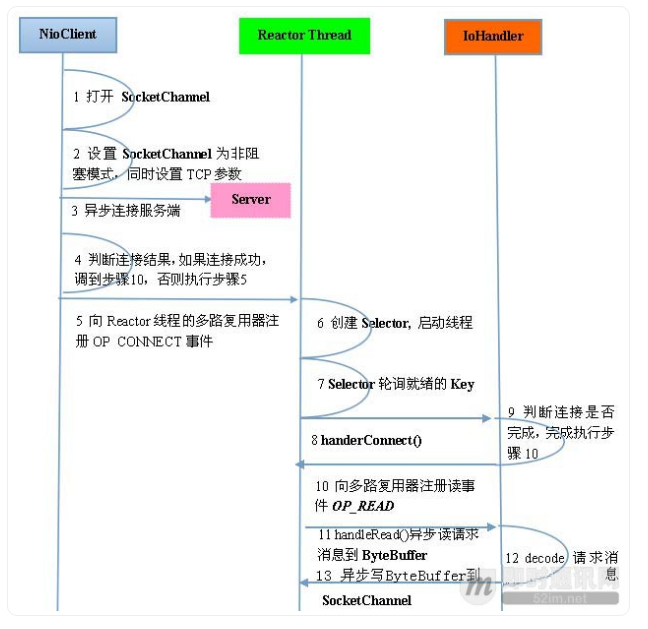
要知道Redis的数据结构并不全是简单的Key-Value，还有列表，hash，map等等复杂的结构，这些结构有可能会进行很细粒度的操作，比如在很长的列表后面添加一个元素，在hash当中添加或者删除一个对象，等等。这些操作还可以合成MULTI/EXEC的组。这样一个操作中可能就需要加非常多的锁，导致的结果是同步开销大大增加。这还带来一个恶果就是吞吐量虽然增大，但是响应延迟可能会增加。   
Redis在权衡之后的选择是用单线程，突出自己功能的灵活性。在单线程基础上任何原子操作都可以几乎无代价地实现，多么复杂的数据结构都可以轻松运用，甚至可以使用Lua脚本这样的功能。对于多线程来说这需要高得多的代价。

**44.使用传统的 Socket 开发挺简单的，我为什么要切换到 NIO 进行编程呢？**

1. 线程模型存在致命缺陷：一连接一线程的模型导致服务端无法承受大量客户端的并发连接；
2. 性能差：频繁的线程上下文切换导致 CPU 利用效率不高；
3. 可靠性差：由于所有的 IO 操作都是同步的，所以业务线程只要进行 IO 操作，也会存在被同步阻塞的风险，这会导致系统的可靠性差，依赖外部组件的处理能力和网络的情况。
4. 采用非阻塞 IO（NIO）之后，同步阻塞 IO 的三个缺陷都将迎刃而解：
5. Nio 采用 Reactor 模式，一个 Reactor 线程聚合一个多路复用器 Selector，它可以同时注册、监听和轮询成百上千个 Channel，一个 IO 线程可以同时并发处理N个客户端连接，线程模型优化为1：N（N < 进程可用的最大句柄数）或者 M : N (M通常为 CPU 核数 + 1， N < 进程可用的最大句柄数)；
6. 由于 IO 线程总数有限，不会存在频繁的 IO 线程之间上下文切换和竞争，CPU 利用率高；
7. 所有的 IO 操作都是异步的，即使业务线程直接进行 IO 操作，也不会被同步阻塞，系统不再依赖外部的网络环境和外部应用程序的处理性能。

由于切换到 NIO 编程之后可以为系统带来巨大的可靠性、性能提升，所以，目前采用 NIO 进行通信已经逐渐成为主流。

****

****