Java 基础

1、Java 的跨平台原理

Java 通过不同的系统、不同的版本、不同位数的 Java 虚拟机（JVM），来屏蔽不同的系统指令集差异，而对外提供统一的接口（Java API）。对于我们普通的 Java 开发者而言，只需要按照接口开发即可。如果程序需要部署在不同的环境，只需要在系统上安装对应版本的虚拟机即可。

2、搭建一个 Java 开发环境的步骤

需要的组件：

适用于开发环境的 JDK；

相应的集成开发环境 IDE；

Web 服务器（tomcat）

下载对应的组件

JDK 安装，配置 JAVA_HOME、PATH、CLASSPATH 环境变量，IDE 和 tomcat 都依赖于环境变量；安装 IDE 并且安装 tomcat 插件。

3、JDK 与 JRE 的区别

JVM：（Java Virtual Machine），Java 虚拟机。只认识 xxx.class 这种类型的文件，能够将 class 文件中的字节码指令进行识别并调用操作系统向上的 API 完成动作。JVM 是 Java 能够跨平台的核心。

JRE（Java Runtime Environment），Java 运行环境，为 Java 运行提供所需环境，要包含两个部分，JVM 的标准实现和 Java 的一些基本类库。

JDK（Java Development Kit）

Java 开发工具包，提供 Java 的开发环境和运行环境，Java 程序调试和分析的工具

4、接口和抽象类

抽象类不可实例化；

抽象类可以有构造器，接口无构造器

接口方法默认修饰符 public，抽象方法可以有 **public**、**protected** 和 **default** 这些修饰符；

抽象方法可以有 main 方法并且可以运行。

相同：

子类需要提供抽象类或接口中所有声明的方法的实现。

5、int 数据占几个字节

8 种基本数据类型

数据类型	大小（二进制位）	范围	默认值
byte	8	-128~127	0
short	16	-32768~32767	0
int	32（四个字节）	-2^31~2^31-1	0
long	64	-2^63~2^63-1	0
float	32		0
double	64		0
char	16	0~65535	null
boolean	1		false

6、Java 面向对象的特征

封装、抽象、继承、多态

封装：将对象封装成一个高度自治和相对封闭的个体，对象的属性（状态）由对象自己的行为（方法）来改变和读取；

抽象：找出一些对象的相似和共性之处，将这些对象归为一类；

继承：在定义和实现一个类时，可以在一个已经存在的类基础上来进行；利用原有的内容，添加新的内容或者修改原有的内容以适应新的需要；

多态：程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行时才确定。

7、装箱和拆箱

基本数据类型不具备对象的特性

装箱：将基本数据类型转换为对应的包装类型

拆箱：将包装类型转换为基本数据类型

8、int 与 Integer 的区别

1、Ingeter 是 int 的包装类，int 的初值为 0，Ingeter 的初值为 null；

2、无论如何，Integer 和 new Integer 不会相等。不会经历拆箱过程，前者指向常量池，后者的引用指向堆。他们的内存地址不一样。

3、两个都是非 new 出来的 Integer，如果数在-128 到 127 之间，则是 true，否则为 false。

4、两个都是 new 出来的，为 false。

5、int 和 Integer（无论 new 否），都为 true，因为会把 Integer 自动拆箱为 int 再比较。

9、“==”和 equals 的区别

==用来判断两个变量的的值是否相等，如果是基本数据类型的值直接比较值；如果是引用类型，比较对象在内存中的地址；

equals 方法不能作用于基本数据类型的变量

如果没有对 equals 方法进行重写，则 equals 方法是用来比较两个对象的引用是否相等，即是否指向同一个对象；

String 等类对 equals 方法进行了重写，用来比较指向的字符串对象所存储的字符串是否相等。

10、为什么重写 equals 方法还要重写 hashcode 方法

Object 类默认的 equals 比较规则就是比较两个对象的内存地址；

hashcode 是根据对象的内存地址经哈希算法得来的

equals 方法重写后，也就是比较的不再是内存地址，由于 hashcode 方法必须和 equals 方法一致，由此必须重写 hashcode 方法。

11、String、StringBuilder 和 StringBuffer

String 是内容不可变的字符串，String 底层使用了一个不可字符数组（final）修饰；

StringBuffer 线程安全，效率较低（synchronized 关键字）。

12、Java 中集合类的基本接口

value 和 key-value（Collection 和 Map）

存储值的分为 List 和 Set

List 是有序的，可以重复的；

存元素：多次调用 add(Object)方法时，每次加入的对象按先来后到的顺序排序，也可以插队，即调用 add(int index,Object)方法，就可以指定当前对象在集合中的存放位置。

取元素：方法 1：Iterator 接口取得所有，逐一遍历各个元素

方法 2：调用 get(indexi)来明确说明取第几个。

Set 是无序的，不可以重复的。

存元素：add 方法有一个 boolean 的返回值，当集合中没有某个元素，此时 add 方法可成功加入该元素时，则返回 true；当集合含有与某个元素 equals 相等的元素时，此时 add 方法无法加入该元素，返回结果为 false。

取元素：没法说取第几个，只能以 Iterator 接口取得所有的元素，再逐一遍历各个元素。

Map 是双列的集合，存放用 put 方法:put(obj key,obj value)，每次存储时，要存储一对 key/value，不能存储重复的 key，这个重复的规则也是按 equals 比较相等。

取元素：用 get(Object key)方法根据 key 获得相应的 value。

也可以获得所有的 key 的集合，还可以获得所有的 value 的集合，还可以获得 key 和 value 组合成的 Map.Entry 对象的集合。

13、Array 和 ArrayList 的区别

Array 可以容纳基本类型和对象，而 ArrayList 只能容纳对象

Array 容量固定且无法动态改变，ArrayList 的大小是动态变化的

Array 无 addAll，removeAll 和 iterator 等方法

14、ArrayList 和 LinkedList

ArrayList 底层使用数组实现

LinkedList 使用链表实现

数组具有索引，查询特定的元素比较快，插入和删除比较慢；

15、HashMap 和 HashTable，HashMap 为什么使用红黑树而不使用平衡树？

HashMap 可以把 null 作为 key 和 value，

HashMap 线程不安全，效率较高；

HashMap 的迭代器(Iterator)是 fail-fast 迭代器，会抛出 ConcurrentModificationException 异常，

HashTable 的 put()、remove()等方法都加了 synchronized 关键字，保证线程安全

Hashtable 的 enumerator 迭代器不是 fail-fast 的。

红黑树牺牲了一些查找性能 但其本身并不是完全平衡的二叉树。因此插入删除操作效率略高于 AVL 树

HashMap 线程不安全的原因：

HashMap 在 put 的时候，插入的元素超过了容量（由负载因子决定）的范围就会触发扩容操作，就是 rehash，这个会重新将原数组的内容重新 hash 到新的扩容数组中，在多线程的环境下，存在同时其他的元素也在进行 put 操作，如果 hash 值相同，可能出现同时在同一数组下用链表表示，造成闭环，导致在 get 时会出现死循环，所以 HashMap 是线程不安全的。

ConcurrentHashMap 原理

线程安全且高效的 HashMap

ConcurrentHashMap 为了提高本身的并发能力，在内部采用了 Segment 结构，一个 Segment 其实就是一个类似 Hash Table 的结构，Segment 内部维护了一个链表数组；

ConcurrentHashMap 定位一个元素的过程需要进行两次 Hash 操作，第一次 Hash 定位到 Segment，第二次 Hash 定位到元素所在的链表的头部，因此，这一种结构的带来的副作用是 Hash 的过程要比普通的 HashMap 要长，但是带来的好处是写操作的时候可以只对元素所在的 Segment 进行操作即可，不会影响到其他的 Segment，这样，在最理想的情况下，ConcurrentHashMap 可以最高同时支持 Segment 数量大小的写操作（刚好这些写操作都非常平均地分布在所有的 Segment 上），所以，通过这一种结构，ConcurrentHashMap 的并发能力可以大大的提高。

fail-fast

fail-fast 机制是 java 集合(Collection)中的一种错误机制。当多个线程对同一个集合的内容进行操作时，就可能会产生 fail-fast 事件。例如：当某一个线程 A 通过 iterator 去遍历某集合的过程中，若该集合的内容被其他线程所改变了；那么线程 A 访问集合时，就会抛出 ConcurrentModificationException 异常，产生 fail-fast 事件。

fail-safe

采用安全失败机制的集合容器，在遍历时不是直接在集合内容上访问的，而是先复制原有集合内容，在拷贝的集合上进行遍历。

原理：由于迭代时是对原集合的拷贝进行遍历，所以在遍历过程中对原集合所作的修改并不能被迭代器检测到，所以不会触发 Concurrent Modification Exception。

缺点：基于拷贝内容的优点是避免了 Concurrent Modification Exception，但同样地，迭代器并不能访问到修改后的内容，即：迭代器遍历的是开始遍历那一刻拿到的集合拷贝，在遍历期间原集合发生的修改迭代器是不知道的。

16、实现一个拷贝文件的工具使用字节流还是字符流

拷贝的文件不确定是否只包含字符流，有可能有字节流（有图片、声音），为了考虑到通用性，要使用字节流。

17、线程的实现方式，线程池

继承 Thread 类和实现 Runable 接口

Java 中只支持单继承，继承的扩展性不强；

线程池的作用：

限定线程的个数，不会导致由于线程过多而系统运行缓慢或崩溃；

减少在创建和销毁线程上所花的时间以及系统资源的开销。

18、常用的设计模式

设计模式：通过实践总结出的，在设计过程中可以反复使用的解决特定问题的设计方法。

单例模式：

工厂模式：Spring IOC

代理模式： Spring AOP

包装模式：

19、lamda 表达式的优缺点

20、一个十进制数在内存中如何存在

以二进制补码形式存储，最高位是符号位，正数的补码是它的原码，负数的补码是它的反码加 1，在求反码时符号位不变，符号位为 1，其他位取反

21、值传递与引用传递

Java 中数据类型分为两大类，**基本类型和对象类型**。相应的，变量也有两种类型：基本类型和引用类型。基本类型的变量保存原始值，即它代表的值就是数值本身；而引用类型的变量保存引用值，"引用值"指向内存空间的地址，代表了某个对象的引用，而不是对象本身，对象本身存放在这个引用值所表示的地址的位置。

值传递：

方法调用时，实际参数把它的值传递给对应的形式参数，函数接收的是原始值的一个 copy，**此时内存中存在两个相等的基本类型，即实际参数和形式参数，后面方法中的操作都是对形参这个值的修改，不影响实际参数的值**。

引用传递：

也称为传地址。方法调用时，实际参数的引用(地址，而不是参数的值)被传递给方法中相对应的形式参数，函数接收的是原始值的内存地址；**在方法执行中，形参和实参内容相同，指向同一块内存地址，方法执行中对引用的操作将会影响到实际对象**。

（1）基本数据类型传值，对形参的修改不会影响实参；

（2）引用类型传引用，形参和实参指向同一个内存地址（同一个对象），所以对参数的修改会影响到实际的对象；

（3）String, Integer, Double 等类型经特殊处理，可以理解为传值，最后的操作不会修改实参对象。

22、如何输出某种编码的字符串

```
public String translate (String str) {
        String tempStr = "";
        try {
            tempStr = new String(str.getBytes("ISO-8859-1"), "GBK");
            tempStr = tempStr.trim();
        }catch (Exception e) {
        System.err.println(e.getMessage());
        }
        return tempStr;
}
```

23、&与&&的区别，|和||

&运算符有两种用法：(1)按位与；(2)逻辑与。&&运算符是短路与运算

&也是位运算符。& 表示在运算时两边都会计算，然后再判断；&&表示先运算符号左边的东西，然后判断是否为 true，是 true 就继续运算右边的然后判断并输出，是 false 就停下来直接输出不会再运行后面的东西。

3 | 9=0011（二进制） | 1001（二进制）=1011（二进制）=11（十进制）

24、正则表达式，Java 中正则表达式的实现

正则表达式一般用于字符串匹配, 字符串查找和字符串替换

Pattern 和 Matcher

25、在 JAVA 中调用 finalize 的会怎样？那你知道 finalize 有什么作用？

当一个对象变成一个垃圾对象的时候，如果此对象的内存被回收，那么就可以调用系统中定义的 finalize 方法来完成

26、NIO

缓冲区（Buffer）

缓冲区实际上是一个容器对象，更直接的说，其实就是一个数组，在 NIO 库中，所有数据都是用缓冲区处理的。在读取数据时，它是直接读到缓冲区中的； 在写入数据时，它也是写入到缓冲区中的；任何时候访问 NIO 中的数据，都是将它放到缓冲区中。而在面向流 I/O 系统中，所有数据都是直接写入或者直接将数据读取到 Stream 对象中

通道（Channel）

通道是一个对象，通过它可以读取和写入数据，所有数据都通过 Buffer 对象来处理。我们永远不会将字节直接写入通道中，相反是将数据写入包含一个或者多个字节的缓冲区。同样也不会直接从通道中读取字节，而是将数据从通道读入缓冲区，再从缓冲区获取这个字节。

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。而不是保持线程阻塞，所以直至数据变的可以读取之前，该线程可以继续做其他的事情。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。线程通常将非阻塞 IO 的空闲时间用于在其它通道上执行 IO 操作，所以一个单独的线程可以管理多个输入和输出通道（channel）

选择器（Selector）

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。选怿机制，使得一个单独的线程很容易来管理多个通道。

27、深度拷贝和浅拷贝

浅拷贝只是复制对象的引用地址，两个对象指向同一个内存地址，所以修改其中的任意的值，另一个值都会随之改变；

深拷贝是将对象及值复制过来，两个对象修改其中任意值，另一个值不会改变

28、throws 和 throw 的区别

throws 是用来声明一个方法可能抛出的所有异常信息，throws 是将异常声明但是不处理，而是将异常上传，谁调用就交给谁处理，而 throw 则是指抛出一个具体的异常类型。

29、Java 静态代理和动态代理的区别（底层代码）

静态：由程序员创建代理类或特定工具自动生成源代码再对其编译。在程序运行前代理类的.class 文件就已经存在了。

动态：在程序运行时运用反射机制动态创建而成。

30、Collection 与 Collections

1、java.util.Collection 是一个**集合接口**。它提供了对集合对象进行基本操作的通用接口方法。Collection 接口在 Java 类库中有很多具体的实现。Collection 接口的意义是为各种具体的集合提供了最大化的统一操作方式。

2、java.util.Collections 是一个包装类。它包含有各种有关集合操作的**静态多态方法**。此类**不能实例化**，作为一**个工具类**，服务于 Java 的 Collection 框架。

31、Comparable 和 Comparator

1、一个类在设计之初就要实现对该类对象的排序功能，那么这个类要实现 Comparable 接口，实现 public int compareTo(T t)方法。

2、若一个类在设计之初并没有对排序功能的需求，而是在后续的使用中想要对这个类添加排序的功能，如 Teacher 类。这时的办法是实现一个比较器类，如 TeacherComparator 类，利用 public int compare(T t1, T t2) 让该类实现 Comparator 接口。

32、泛型中<T> 与<?>的区别

T 代表一种类型

?是通配符,泛指所有类型

泛型的限定：

? extends E:接收 E 类型或者 E 的子类型。

？ super E:接收 E 类型或者 E 的父类型。

“<T>"和"<?>"，首先要区分开两种不同的场景：

第一，声明一个泛型类或泛型方法。

第二，使用泛型类或泛型方法。

类型参数“<T>”主要用于第一种，声明泛型类或泛型方法。

无界通配符“<?>”主要用于第二种，使用泛型类或泛型方法

Java 并发

1、对 volatile 的理解

volatile 原理：volatile 是 Java 提供的轻量级的同步机制

保证可见性：

不保证原子性：

原子性 不可分割，完整性

通过 AtomicInteger 的 getAndIncrement 方法解决原子性问题

禁止指令重派：

单例模式在多线程下可能出现的问题

单例模式 volatile

DCL 双端检索机制不一定线程安全，因为有指令重排序的存在，volatile 可以禁止指令重拍。

2、Syncronized

synchronized 关键字常被用于维护数据一致性，利用 synchronized 机制是给共享资源上锁，只有拿到锁的线程才可以访问共享资源，这样就可以强制使得对共享资源的访问都是顺序的。

1、Synchronized 作用于方法时，锁住的是对象的实例(this)；

2、Synchronized 作用于静态方法时，锁住的是 Class 实例，又因为 Class 的相关数据存储在永久带 PermGen（jdk1.8 则是 metaspace），永久带是全局共享的，因此静态方法锁相当于类的一个全局锁，会锁所有调用该方法的线程；

3、Synchronized 作用于一个对象实例时，锁住的是所有以该对象为锁的代码块。它有多个队列，当多个线程一起访问某个对象监视器的时候，对象监视器会将这些线程存储在不同的容器中。

原理

Java 虚拟机中的同步(Synchronization)基于进入和退出 Monitor 对象实现，无论是显式同步(有明确的 monitorenter 和 monitorexit 指令,即同步代码块)还是隐式同步都是如此。在 Java 语言中，同步用的最多的地方可能是被 synchronized 修饰的同步方法。同步方法并不是由 monitorenter 和 monitorexit 指令来实现同步的，而是由方法调用指令读取运行时常量池中方法表结构的 ACC_SYNCHRONIZED 标志来隐式实现的

Monitor

我们可以把它理解为一个同步工具，也可以描述为一种同步机制，它通常被描述为一个对象。与一切皆对象一样，所有的 Java 对象是天生的 Monitor，每一个 Java 对象都有成为 Monitor 的潜质，因为在 Java 的设计中 ，每一个 Java 对象自打娘胎里出来就带了一把看不见的锁，它叫做内部锁或者 Monitor 锁。Monitor 是线程私有的数据结构，每一个线程都有一个可用 monitor record 列表，同时还有一个全局的可用列表。每一个被锁住的对象都会和一个 monitor 关联（对象头的 MarkWord 中的 LockWord 指向 monitor 的起始地址），同时 monitor 中有一个 Owner 字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用

3、Synchronized 与 ReentrantLock

1、两者都是可重入锁

两者都是可重入锁。“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增 1，所以要等到锁的计数器下降为 0 时才能释放锁。

2、synchronized 依赖于 JVM 而 ReentrantLock 依赖于 API

synchronized 是依赖于 JVM 实现的，虚拟机团队在 JDK1.6 为 synchronized 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。ReentrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock() 方法配合 try/finally 语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。

3、 ReentrantLock 比 synchronized 增加了一些高级功能

相比 synchronized，ReentrantLock 增加了一些高级功能。主要来说主要有三点：

等待可中断；通过 lock.lockInterruptibly() 来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。

可实现公平锁；ReentrantLock 默认情况是非公平的，可以通过 ReentrantLock 类的 ReentrantLock(boolean fair)构造方法来制定是否是公平的。

可实现选择性通知（锁可以绑定多个条件）

4、Atomic 的原理

Atomic 包中的类基本的特性：

在多线程环境下，当有多个线程同时对单个（包括基本类型和引用类型）变量进行操作时，具有排他性，即对个线程同时对该变量的值进行更新时，仅有一个线程成功，而未成功的线程可以像自旋锁一样，继续尝试，一直等到执行成功。

通过 **CAS 乐观锁**保证原子性，通过**自旋**保证当次修改的最终修改成功，通过**降低锁粒度（多段锁）**增加并发性能。

5、JMM

JMM：Java 内存模型

线程解锁前，必须把共享变量的值刷新回主内存；

线程加锁前，必须读取主内存中最新的值到自己的工作内存；

加锁解锁是同一把锁

线程间的通信通过主内存完成

6、CAS

Compare And Swap

底层原理：

更新一个变量的时候，只有当变量的预期值 A 和内存地址 V 当中的实际值相同时，才会将内存地址 V 对应的值修改为 B。

自旋锁：是指当一个线程在获取锁的时候，如果锁已经被其它线程获取，那么该线程将循环等待，然后不断的判断锁是否能够被成功获取，直到获取到锁才会退出循环。

UnSafe 类

1、通过 Unsafe 类可以分配内存，可以释放内存；

利用 unsafe 提供的原子性操作方法。

Unsafe 类中提供的 3 个本地方法 allocateMemory、reallocateMemory、freeMemory 分别用于分配内存，扩充内存和释放内存

2、可以定位对象某字段的内存位置，也可以修改对象的字段值，即使它是私有的；

3、挂起与恢复

内存地址偏移量

CAS 缺点

ABA 问题：当一个值从 A 变成 B，又更新回 A，普通 CAS 机制会误判通过检测。

ABA 问题的解决：利用版本号比较可以有效解决 ABA 问题。

AtomicStampedReference/AtomicMarkableReference

通过版本号（时间戳）来解决 ABA 问题的

即乐观锁每次在执行数据的修改操作时，都会带上一个版本号，一旦版本号和数据的版本号一致就可以执行修改操作并对版本号执行+1 操作，否则就执行失败

自旋锁存在的问题：

1、多个线程争夺同一个资源时，如果自旋一直不成功，将会一直占用 CPU。

2、Java 实现的自旋锁不是公平的，即无法满足等待时间最长的线程优先获取锁。

多变量共享一致性问题：

自适应自旋锁

自旋时间不再固定，由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定

7、集合类线程不安全的场景

ArrayList 线程不安全，出现 ConcurrentModificationException

1、使用 Vector

2、Collections.synchronizedList(new ArrayList<>());

3、CopyOnWriteArrayList 写时复制

当我们往一个容器添加元素的时候，不直接往当前容器添加，而是先将当前容器进行 Copy，复制出一个新的容器，然后新的容器里添加元素，添加完元素之后，再将原容器的引用指向新的容器。这样做的好处是我们可以对 CopyOnWrite 容器进行并发的读，而不需要加锁，因为当前容器不会添加任何元素。所以 CopyOnWrite 容器也是一种读写分离的思想，读和写不同的容器。

8、公平锁与非公平锁

9、可重入锁（递归锁）

可重入就意味着：线程可以进入任何一个它已经拥有的锁所同步着的代码块。

ReentrantLock 是一个**可重入的互斥（/独占）锁**，又称为“独占锁”。

10、读写锁

读写锁是一种特殊的自旋锁，它把共享资源的访问者划分成了读者和写者，读者只对共享资源进行访问，写者则是对共享资源进行写操作。读写锁在 ReentrantLock 上进行了拓展使得该锁更适合读操作远远大于写操作对场景。一个读写锁同时只能存在一个写锁但是可以存在多个读锁，但不能同时存在写锁和读锁。

ReentrantReadWriteLock()

11、CountDownLatch、CyclicBarrier 和 Semaphore

CountDownLatch

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。

CyclicBarrier

让一组线程到达一个屏障，一个集合点时，被阻塞，直到所有的线程都到了这个集合点时，屏障才会打开，然后线程才能继续往下执行。

CyclicBarrier 可以用于多线程计算数据，最后合并计算结果的场景

Semaphore

Semaphore 类是一个计数信号量，必须由获取它的线程释放，通常用于限制可以访问某些资源（物理或逻辑的）线程数目。

12、AQS

AbstractQueuedSynchronizer 抽象队列同步器

AQS 定义两种资源共享方式：Exclusive（独占，只有一个线程能执行，如 ReentrantLock ） 和 Share （ 共 享 ， 多 个 线 程 可 同 时 执 行 ， 如 Semaphore/CountDownLatch）。

不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS 已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

isHeldExclusively()：该线程是否正在独占资源。只有用到 condition 才需要去实现它。

tryAcquire(int)：独占方式。尝试获取资源，成功则返回 true，失败则返回 false。

tryRelease(int)：独占方式。尝试释放资源，成功则返回 true，失败则返回 false。

tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0 表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。

tryReleaseShared(int)：共享方式。尝试释放资源，如果释放后允许唤醒后续等待结点返回 true，否则返回 false。

13、阻塞队列

阻塞队列是一个在队列基础上又支持了两个附加操作的队列。

支持阻塞的**插入**方法：队列满时，队列会阻塞插入元素的线程，直到队列不满。

支持阻塞的**移除**方法：队列空时，获取元素的线程会等待队列变为非空。

为什么需要：不需要关心什么时候阻塞线程，什么时候唤醒线程。通过数据结构本身的特性来解决。

多线程环境中，通过队列可以很容易实现数据共享，比如经典的“生产者”和“消费者”模型中，通过队列可以很便利地实现两者之间的数据共享。假设我们有若干生产者线程，另外又有若干个消费者线程。如果生产者线程需要把准备好的数据共享给消费者线程，利用队列的方式来传递数据，就可以很方便地解决他们之间的数据共享问题。

核心方法

架构介绍

种类分析

ArrayBlockingQueue

LinkedBlockingQueue

DelayQueue

PriorityBlockingQueue

SynchronousQueue

LinkedTransferQueue

LinkedBlockingDeque

14、Callable 接口

1、创建执行线程的方式：实现 Callable 接口。 相较于实现 Runnable 接口的方式，方法可以有返回值，并且可以抛出异常。

2、执行 Callable 方式，需要 FutureTask 实现类的支持，用于接收运算结果。FutureTask 是 Future 接口的实现类。

15、线程池状态，线程池参数的确定策略

Running，ShutDown、Stop、Tidying、Terminated

corePoolSize:

每个任务需要 tasktime 秒处理，则每个线程每钞可处理 1/tasktime 个任务。系统每秒有 tasks 个任务需要处理，则需要的线程数为：tasks/(1/tasktime)，即 tasks*tasktime 个线程数。假设系统每秒任务数为 100~1000，每个任务耗时 0.1 秒，则需要 100*0.1 至 1000*0.1，即 10~100 个线程。那么 corePoolSize 应该设置为大于 10，具体数字最好根据 8020 原则，即 80%情况下系统每秒任务数，

若系统 80%的情况下第秒任务数小于 200，最多时为 1000，则 corePoolSize 可设置为 20。

queueCapacity:

任务队列的长度要根据核心线程数，以及系统对任务响应时间的要求有关。队列长度可以设置为(corePoolSize/tasktime)*responsetime： (20/0.1)*2=400，即队列长度可设置为 400。

队列长度设置过大，会导致任务响应时间过长，切忌以下写法：

LinkedBlockingQueue queue = new LinkedBlockingQueue();

这实际上是将队列长度设置为 Integer.MAX_VALUE，将会导致线程数量永远为 corePoolSize，再也不会增加，当任务数量陡增时，任务响应时间也将随之陡增。

maxPoolSize:

当系统负载达到最大值时，核心线程数已无法按时处理完所有任务，这时就需要增加线程。每秒 200 个任务需要 20 个线程，那么当每秒达到 1000 个任务时，则需要(1000-queueCapacity)/(1/tasktime)，即 60 个线程，可将 maxPoolSize 设置为 60。

keepAliveTime:

线程数量只增加不减少也不行。当负载降低时，可减少线程数量，如果一个线程空闲时间达到 keepAliveTime，该线程就退出。默认情况下线程池最少会保持 corePoolSize 个线程。

allowCoreThreadTimeout:

默认情况下核心线程不会退出，可通过将该参数设置为 true，让核心线程也退出。

以上关于线程数量的计算并没有考虑 CPU 的情况。若结合 CPU 的情况，比如，当线程数量达到 50 时，CPU 达到 100%，则将 maxPoolSize 设置为 60 也不合适，此时若系统负载长时间维持在每秒 1000 个任务，则超出线程池处理能力，应设法降低每个任务的处理时间(tasktime)。

16、线程池中 submit 方法和 execute 方法

submit 有返回值，而 execute 没有

submit 方便 Exception 处理

17、线程池的拒绝策略你谈谈？

是什么

等待队列已经满了，再也塞不下新的任务，同时线程池中的线程数达到了最大线程数，无法继续为新任务服务。

拒绝策略

AbortPolicy：处理程序遭到拒绝将抛出运行时 RejectedExecutionException；

CallerRunsPolicy：线程调用运行该任务的 execute 本身。此策略提供简单的反馈控制机制，能够减缓新任务的提交速度；

DiscardPolicy：不能执行的任务将被删除；

DiscardOldestPolicy：如果执行程序尚未关闭，则位于工作队列头部的任务将被删除，然后重试执行程序（如果再次失败，则重复此过程）。

18、进程与线程的区别

进程：是并发执行的程序在执行过程中分配和管理资源的基本单位，是一个动态概念，竞争计算机系统资源的基本单位。

线程：是进程的一个执行单元，是进程内部的调度实体。比进程更小的独立运行的基本单位。线程也被称为轻量级进程。

调度：线程是处理器调度的基本单位，但是进程不是；

并发性：

拥有的资源：同一进程内的线程共享本进程的资源如内存、I/O、cpu 等，但是进程之间的资源是独立的。一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃整个进程都死掉。所以多进程要比多线程健壮。

系统开销：

进程间的通信方式：

管道（pipe）：

信号量（semophore）

消息队列（message queue）

信号（signal）：信号是一种比较复杂的通信方式，用于通知接收进程某一事件已经发生。

共享内存（shared memory）：

套接字（socket）

线程间的通信方式：

1、锁机制

互斥锁：提供了以排它方式阻止数据结构被并发修改的方法。

读写锁：允许多个线程同时读共享数据，而对写操作互斥。

条件变量：可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。

2、信号量机制：包括无名线程信号量与有名线程信号量

3、信号机制：类似于进程间的信号处理。

19、Java 线程上下文切换

即使是单核 CPU 也支持多线程执行代码，CPU 通过给每个线程分配 CPU 时间片来实现这个机制。时间片是 CPU 分配给各个线程的时间，因为时间片非常短，所以 CPU 通过不停地切换线程执行，让我们感觉多个线程时同时执行的，时间片一般是几十毫秒（ms）。

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会切换到下一个任务。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，可以再次加载这个任务的状态，**从任务保存到再加载的过程就是一次上下文切换**。

上下文切换可以认为是内核（操作系统的核心）在 CPU 上对于进程（包括线程）进行以下的活动：

挂起一个进程，将这个进程在 CPU 中的状态（上下文）存储于内存中的某处；

恢复一个进程，在内存中检索下一个进程的上下文并将其在 CPU 的寄存器中恢复；

跳转到程序计数器所指向的位置（即跳转到进程被中断时的代码行），以恢复该进程。

20、wait 方法和 sleep 方法的区别

sleep()方法是属于 Thread 类中的。而 wait()方法，则是属于 Object 类中的。

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

在调用 sleep()方法的过程中，线程不会释放对象锁。

而当调用 wait()方法的时候，线程会放弃对象锁，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获取对象锁进入运行状态。

21、线程 run 方法和 start 方法的区别

每个线程都是通过某个特定 Thread 对象所对应的 run 方法来完成其操作，方法 run 称为线程体。通过调用一个 Thread 类的 start 方法来启动一个线程。

22、ThreadLocal

ThreadLocal 用于保存某个线程共享变量：对于同一个 static ThreadLocal，不同线程只能从中 get，set，remove 自己的变量，而不会影响其他线程的变量。

1、ThreadLocal.get: 获取 ThreadLocal 中当前线程共享变量的值。

2、ThreadLocal.set: 设置 ThreadLocal 中当前线程共享变量的值。

3、ThreadLocal.remove: 移除 ThreadLocal 中当前线程共享变量的值。

4、ThreadLocal.initialValue: ThreadLocal 没有被当前线程赋值时或当前线程刚调用 remove 方法后调用 get 方法，返回此方法值。

ThreadLocal 的特点

用一个 ThreadLocal 也可以多次 set 一个数据，set 仅仅表示的是线程的 ThreadLocal.ThreadLocalMap 中 table 的某一位置的 value 被覆盖成最新设置的那个数据而已，**对于同一个 ThreadLocal 对象而言，set 后，table 中绝不会多出一个数据**。

1、**ThreadLocal 不是集合**，它不存储任何内容，真正存储数据的集合在 Thread 中。**ThreadLocal 只是一个工具，一个往各个线程的 ThreadLocal.ThreadLocalMap 中 table 的某一位置 set 一个值的工具而已**

2、同步与 ThreadLocal 是解决多线程中数据访问问题的两种思路，**前者是数据共享的思路，后者是数据隔离的思路**

3、同步是一种以时间换空间的思想，ThreadLocal 是一种空间换时间的思想

4、ThreadLocal 既然是与线程相关的，那么对于 Java Web 来讲，ThreadLocal 设置的值只在一次请求中有效，是不是和 request 很像？因为 request 里面的内容也只在一次请求有效，对比一下二者的区别：

23、分布式环境下，怎么保证线程安全

避免并发：设计一个规则来保证一个客户的计算工作和数据访问只会被一个线程或一台工作机完成，而不是把一个客户的计算工作分配给多个线程去 完成；

时间戳：

串行化：通过一个队列先把调用信息缓存起来，然后再串行地处理这些调用；

数据库：通过唯一性索引来解决并发过程中重复数据的生产或重复任务的执行；另外有些更新计算操作也尽量通过 sql 来完成，因为在程序段计算好后再去更新就有可能发生脏复写问题，但通过一条 sql 来完成计算和更新就可以通过数据库的锁机制来保证 update 操作的一致性；

行锁：

统一触发途径：当一个数据可能会被多个触发点或多个业务涉及到，就有并发问题产生的隐患，因此可以通过前期架构和业务设计，尽量统一触发途径，触发途径少了一是减少并发的可能，也有利于对于并发问题的分析和判断。

JVM

1、JVM 主要组成部分

类加载器

运行时数据区

执行引擎

本地库接口

2、JVM 运行时数据区

程序计数器

程序计数器是一块 较小 的内存空间，它可以看做是当前线程所执行的字节码的行号指示器 ；在虚拟机的概念模型里（仅仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），字节码解释器工作时，就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳准、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。

虚拟机栈

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同时创建一个栈帧，用来存储局部变量表，操作栈，动态链接，方法出口等信息。每个方法从调用直到执行完成的过程，都对应一个栈帧在虚拟机栈中从入栈到出栈的过程。

一个线程中的方法调用链路可能会很长，很多方法都处于同时执行的状态。对于执行引擎来说，在活动线程中，**只有处于栈顶的栈帧才是有效的，称为当前栈帧**，与这个栈帧相关联的方法称为当前方法。

本地方法栈

对于一个运行中的 Java 程序而言，它还可能会用到一些跟本地方法相关的数据区。当某个线程调用一个本地方法时，它就进入了一个全新的并且不再受虚拟机限制的世界。本地方法可以通过本地方法接口来访问虚拟机的运行时数据区

堆

Java 中的堆是用来存储对象本身以及数组（当然，数组引用是存放在 Java 栈中的）。只不过和 C 语言中的不同，在 Java 中，程序员基本不用去关心空间释放的问题，Java 的垃圾回收机制会自动进行处理。因此这部分空间也是 Java 垃圾收集器管理的主要区域。另外，堆是被所有线程共享的，在 JVM 中只有一个堆

方法区

方法区与堆一样，是被线程共享的区域。在方法区中，存储了每个类的信息（包括类的名称、方法信息、字段信息）、静态变量、常量以及编译器编译后的代码等。

在 Class 文件中除了类的字段、方法、接口等描述信息外，还有一项信息是**常量池**，用来存储编译期间生成的字面量和符号引用。

在方法区中有一个非常重要的部分就是**运行时常量池**，它是每一个类或接口的常量池的运行时表示形式，在类和接口被加载到 JVM 后，对应的运行时常量池就被创建出来。当然并非 Class 文件常量池中的内容才能进入运行时常量池，在运行期间也可将新的常量放入运行时常量池中，比如 String 的 intern 方法。

3、Java 中 Heap 与 Stack 的区别

1) Heap 是 Stack 的一个子集.------扩展—>从内存观点考虑。

2) Stack 存取速度仅次于寄存器，存储效率比 heap 高，可共享存储数据，但是其中数据的大小和生存期必须在运行前确定。

3) Heap 是运行时可动态分配的数据区，从速度看比 Stack 慢，Heap 里面的数据不共享，大小和生存期都可以在运行时再确定。

4) new 关键字 是运行时在 Heap 里面创建对象,每 new 一次都一定会创建新对象。

4、如何确定垃圾，GCRoot

引用计数法，（循环引用）

根搜索路径算法

哪些对象可以作为 GC Roots

1、虚拟机栈（栈帧中的局部变量，也叫做局部变量表）中引用的对象

2、方法区中的类静态属性引用的对象

3、方法区中常量引用的对象

4、本地方法栈中 JNI（Native 方法）引用的对象

5、JVM 调优和参数配置，如何盘点查看 JVM 系统默认值

JVM 参数类型

标配参数

X 参数

XX 参数：布尔类型

KV 设值类型

6、强引用、软引用、弱引用和虚引用

强引用：

软引用：

弱引用：

软引用和弱引用的应用场景：大量图片的加载，缓存

虚引用：

7、OOM

OutOfMemeoryError：

StackOverflowError

方法递归调用

Java heap space

大对象溢出

GC overhead limit exceeded

大部分资源用于垃圾回收，且回收效率不高

Metaspace

元空间溢出

unable to create new native thread

创建过多线程

Direct buffer memory

本地内存不属于 GC 管辖的范围

8、Java 垃圾回收策略

方法区回收

方法区主要回收：废弃常量和无用的类

废弃常量：例如一个字符串"abc"已经进入常量池中，但是当前系统没有其他地方引用这个字面量。

无用的类：

该类的所有实例都已经被回收，也就是 Java 堆中不存在该类的任何实例

加载该类的 ClassLoader 已经被回收

该类对应的 Java.lang.Class 对象没有在任何地方被引用，无法再任何地方通过反射访问该类的方法。

垃圾收集算法

标记-清除算法：首先标记出所有需要回收的对象，在标记完成后统一回收所有标记的对象。

不足：1.效率问题，标记和清除两个过程效率不高

2.空间问题，标记清楚产生大量内存碎片

复制算法：将内存安装容量分为大小相等的两块，每次使用其中一块。当回收时，将存活的对象复制到未使用的内存中，清除已使用过的内存。

不足：内存缩小到运来的一半

Java 在新生代中采用这种算法，不过是将内存分为 Eden 和两块 Survivor,每次使用 Eden 和其中的 Suvivor，当回收时，将 Eden 和 Suvivor 存活的内存复制到未使用的 Suvivor 空间。HotSpot 默认 Eden 与 Suvivor 比例为 8：1，相当于可以使用 90%的内存，如果存活的内存超过 Suvivor 空间，就是用老年代进行分配担保。

标记-整理算法：过程与标记-清除算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象向一端移动，然后清理掉端边界以外的内存。

分代收集算法：将内存划分成几块，各个区域采用最适当的收集算法。Java 一般把 Java 堆分为新生代和老年代，按各个年代特点采用适当的算法。

垃圾收集器：

新生代：Serial 收集器，ParNew 收集器，Parallel Scavenge 收集器（复制算法）

老年代：CMS（标记-清除）, Seral Old(MSC), Parallel Old 收集器（标记-整理）

GI 收集器

CMS

初始标记（CMS initial mark）初始标记仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，但需要“Stop The World”。

并发标记（CMS concurrent mark）并发标记阶段就是进行 GC Roots Tracing 的过程，刚才产生的集合中标记出存活对象；应用程序也在运行；并不能保证可以标记出所有的存活对象；

重新标记（CMS remark）重新标记阶段是为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录；仍然需要需要”Stop The World“，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

并发清除（CMS concurrent sweep）并发清除阶段会清除对象，回收所有的垃圾对象。

G1

初始标记（Initial Marking）初始标记阶段仅仅只是标记一下 GC Roots 能直接关联到的对象，并且修改 TAMS（Next Top at Mark Start）的值，让下一阶段用户程序并发运行时，能在正确可用的 Region 中创建新对象，这阶段需要停顿线程，但耗时很短。

并发标记（Concurrent Marking）并发标记阶段是从 GC Root 开始对堆中对象进行可达性分析，找出存活的对象，这阶段耗时较长，但可与用户程序并发执行。

最终标记（Final Marking）最终标记阶段是为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程 Remembered Set Logs 里面，最终标记阶段需要把 Remembered Set Logs 的数据合并到 Remembered Set 中，这阶段需要停顿线程，但是可并行执行。

筛选回收（Live Data Counting and Evacuation）筛选回收阶段首先对各个 Region 的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来制定回收计划，这个阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分价值高的 Region 区的垃圾对象，时间是用户可控制的，而且停顿用户线程将大幅提高收集效率。回收时，采用“复制”算法，从一个或多个 Region 复制存活对象到堆上的另一个空的 Region，并且在此过程中压缩和释放内存。

9、衡量垃圾收集器的指标有哪些，说说 CMS 收集器，CMS 为什么需要暂停 - stop the world？（菜鸟）

停顿时间：停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户的体验；

高吞吐量：高吞吐量则可以最高效率的利用 CPU 时间，尽快的完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

GC 停顿时间缩短是以牺牲吞吐量和新生代空间来换取的：系统把新生代调小一些，收集 100MB 的新生代肯定比收集 1000MB 的新生代快，这也直接导致垃圾收集器发生的更频繁，原来 10 秒收集一次，每次停顿 100 毫秒，现在 5 秒收集一次，每次停顿 70 毫秒，停顿时间在下降，但吞吐量也下降了

为什么 STW

垃圾回收并不会阻塞我们程序的线程，而是与当前程序并发执行的。所以问题就出在这里，当 GC 线程标记好了一个对象的时候，此时我们程序的线程又将该对象重新加入了“关系网”中，当执行二次标记的时候，该对象也没有重写 finalize()方法，因此回收的时候就会回收这个不该回收的对象。

虚拟机的解决方法就是在一些特定指令位置设置一些“安全点”，当程序运行到这些“安全点”的时候就会暂停所有当前运行的线程（Stop The World 所以叫 STW），暂停后再找到“GC Roots”进行关系的组建，进而执行标记和清除。

10、类初始化与实例初始化

类初始化：

1、一个类要创建实例需要先加载并初始化该类

Main() 方法所在的类需要先加载和初始化

2、一个子类要初始化需要先初始化父类

3、一个类初始化就是执行<clinit>()方法

<clinit>()方法由静态类变量显示赋值代码和静态代码块组成

类变量显示赋值代码和静态代码块从上到下顺序执行

<clinit>()方法只执行一次

实例初始化：

1、实例初始化就是执行<init>()

<init>()方法可能重载有多个，有几个构造器就有几个<init>方法

<init>()方法由非静态实例变量显示赋值代码和非静态代码块，对应构造器组成

非静态实例变量显示赋值代码和非静态代码块从上到下顺序执行，而对应构造器的代码最后执行

每次创建实例对象，调用对应构造器，执行的就是对应的<init>方法

<init>方法首行执行的是 super()，即对应的父类的<init>方法

哪些方法不能被重写

final 方法

静态方法

private 等子类不可见的方法

对象多态

子类如果重写了父类的方法，通过子类对象调用的一定是子类重写过的代码

非静态犯法默认的调用对象是 this

this 对象在构造器或者说是<init>方法中就是正在创建的对象

11、JVM 发生 CMS GC 的 5 种情况

CMS GC 在实现上分成 foreground collector 和 background collector。

foreground collector

遇到对象分配但是空间不够，就会直接触发 GC，来立即进行空间回收，采用的算法是 mark sweep

background collector

background collector 是通过 CMS 后台线程不断进行扫描，过程中主要判断是否符合 background collector 的触发条件。

每次扫描时，先等 CMSWaitDuration 时间，然后在进行一次 shouldConcurrentCollect 判断，看是否满足 CMS background collector 的触发条件。

CMSWaitDuration 默认时间为 2s

1、是否是并行 Full GC

指的是在 GC cause 是 gclocker 且配置了 GCLockerInvockesConcurrent 参数，或者 GC cause 是 javalangsystemc（就是 System.gc() 调用）且配置了 ExplicitGCInvockesConcurrent 参数，这时会触发一次 background collector。

2、根据统计数据动态计算

（未配置 UseCMSInitiatingOccupancyOnly 时）

如果预测 CMS GC 完成所需时间大于预计的老年代将要填满的时间，则进行 GC

3、根据 Old Gen 情况判断

（a）Old Gen 空间使用占比情况与阈值比较，如果大于阈值则进行 CMS GC

（b）没有配置 UseCMSInitiatingOccupancyOnly

当 Old Gen 刚因为对象分配空间而进行扩容，且成功分配空间，这时候会考虑进行一次 CMS GC

根据 CMS Gen 空闲链判断，

4、根据增量 GC 是否可能会失败（悲观策略）

两代 GC 体系中，主要指的是 Young GC 是否会失败。如果 Young GC 已经失败或者可能会失败，JVM 就认为需要进行一次 CMS GC

5、根据 meta space 判断

12、类加载器

java.lang.ClassLoader 类介绍

java.lang.ClassLoader 类的基本职责就是根据一个指定的类的名称，找到或者生成其对应的字节代码，然后从这些字节代码中定义出一个 Java 类，即 java.lang.Class 类的一个实例

引导类加载器（bootstrap class loader）：

它用来加载 Java 的核心库(jre/lib/rt.jar)，是用原生 C++代码来实现的，并不继承自 java.lang.ClassLoader。

加载扩展类和应用程序类加载器，并指定他们的父类加载器，在 java 中获取不到。

扩展类加载器（extensions class loader）：

它用来加载 Java 的扩展库(jre/ext/*.jar)。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。

系统类加载器（system class loader）：

它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。

自定义类加载器（custom class loader）：

除了系统提供的类加载器以外，开发人员可以通过继承 java.lang.ClassLoader 类的方式实现自己的类加载器，以满足一些特殊的需求。

类加载过程

装载

查找并加载类的二进制数据；

链接

验证：确保被加载类信息符合 JVM 规范、没有安全方面的问题；

准备：为类的静态变量分配内存，并将其初始化为默认值；

解析：把虚拟机常量池中的符号引用转换为直接引用。

初始化：

为类的静态变量赋予正确的初始值。

类的初始化步骤：

1、如果这个类还没有被加载和链接，那先进行加载和链接；

2、假如这个列存在直接父类，并且这个类还没有被初始化，就直接初始化直接父类（不适用于接口）；

3、如果类中存在 static 标识的块，那就依次执行这些初始化语句。

13、loadClass 和 forName 的区别

Class.forName 得到的 class 是已经初始化完成的

创建 JDBC 的 Driver

Classloader.loadClass 得到的 class 是还没有链接的

用于 Spring IOC 懒加载

14、Java 有了 GC 同样会出现内存泄露问题

一、静态集合类像 HashMap、Vector 等的使用最容易出现内存泄露，这些静态变量的生命周期和应用程序一致，所有的对象 Object 也不能被释放，因为他们也将一直被 Vector 等应用着；

二、各种连接，数据库连接，网络连接，IO 连接等没有显示调用 close 关闭，不被 GC 回收导致内存泄露；

三、监听器的使用，在释放对象的同时没有相应删除监听器的时候也可能导致内存泄露。

Java web

1、Http 中的 Get 和 Post 的区别

Get 用于获取、查询资源信息；

Post 用于更新资源信息

区别：

Get 请求提交的数据会在地址栏显示，Post 地址栏不会改变；

传输数据的大小，Get 请求由于浏览器地址长度的限制而导致传输的数据有限制；

Post 请求安全性较高，Get 请求数据会在地址栏显示，可以通过历史记录找到密码等关键信息。

2、对 Servlet 的理解

Servlet 是用 Java 编写的服务器端的程序，功能在于交互式浏览和修改数据，生成动态 Web 内容，Servlet 运行于支持 Java 的服务器中。

HttpServlet 重写 doGet 和 DoPost 方法或者也可以重写 service 方法完成对 get 和 post 请求的响应。

Servlet 生命周期：

Servlet 程序由 web 服务器调用，web 服务器收到客户端的 Servlet 访问请求后：

1、web 服务器首先检查是否已经装载并创建了该 Servlet 的实例对象。如果是，则直接执行第④步，否则，执行第②步。

2、装载并创建该 Servlet 的一个实例对象。

3、调用 Servlet 实例对象的 init()方法。

4、创建一个用于封装 HTTP 请求消息的 HttpServletRequest 对象和一个代表 HTTP 响应消息的 HttpServletResponse 对象，然后调用 Servlet 的 service()方法并将请求和响应对象作为参数传递进去。

5、web 应用程序被停止或重新启动之前，Servlet 引擎将卸载 Servlet，并在卸载之前调用 Servlet 的 destroy()方法。

3、forward 和 redirect

转发（Forword）

通过 RequestDispatcher 对象的 Forward(HttpServletRequest request, HttpServletResponse response) 方法实现。RequestDispatcher 可以通过 HttpServletRequest 的 getRequestDispatcher()方法获得。

重定向（Redirect）

利用服务器返回的状态码来实现。客户端浏览器请求服务器的时候，服务器会返回一个状态码。服务器通过 HttpServletRequestResponse 的 setStatus(int status)方法设置状态码。如果服务器返回 301 或者 302，则浏览器会到新的网址重新请求该资源。

地址栏显示：

forward()是容器中控制权的转向，在客户端浏览器地址栏中不会显示转向后的地址；redirect()是完全的跳转，浏览器（客户端）将会得到跳转的地址，并且重新发送请求链接。

效率：

forward()是服务器端的转向，还是原来的请求，更加高效，也有助于隐藏实际的链接；

redirect()是客户端的跳转，改变浏览器的地址，是重新发起请求。

运用地方：forward：转发页面和转发到的页面可以共享 request 里面的数据。redirect：不能共享数据。

forward：一般用于用户登陆的时候，根据角色转发到相应的模块。redirect：一般用于用户注销登陆时返回主页面和跳转到其它的网站等。

数据共享：

4、JSP 和 Servlet

JSP 是 Servlet 的扩展，继承 HttpServlet 类，JSP 侧重于视图，Servlet 侧重于控制逻辑。

JSP 内置对象，request 和 response

JSP 作用域：page、request、session、application

JSP 传值方式：

5、Session 和 Cookie

都是会话跟踪技术。

Cookie 在客户端记录信息确定用户身份，Session 通过在服务器端记录信息确定用户身份；

Cookie 不安全，可以通过分析存放在本地的 Cookie 进行 Cookie 欺骗；

Session 会在一定时间内保存在服务器上，访问增多会占用服务器性能。

6、Socket 网络编程

网络进程间的通信通过套接字（socket）

socket 起源于 Unix，而 Unix/Linux 基本哲学之一就是“一切皆文件”，都可以用“打开 open –> 读写 write/read –> 关闭 close”模式来操作。Socket 可以理解

为该模式的一个实现，socket 即是一种特殊的文件，一些 socket 函数就是对其进行的操作（读/写 IO、打开、关闭）

作为一个服务器，在调用 socket()、bind()之后就会调用 listen()来监听指定的 socket 地址，如果客户端这时调用 connect()发出连接请求，服务器端就会接收到这个请求，服务端调用 accept()函数取接收请求，这样连接就建立好了。之后就可以开始网络 I/O 操作了，即类同于普通文件的读写 I/O 操作。

Spring 相关

1、Spring 框架是什么

Spring 是一种轻量级的开发框架，旨在提高开发人员的开发效率以及系统的可维护性。

Spring 的 6 个特征

核心技术：依赖注入（DI），AOP，事件（events），资源，i18n，验证，数据绑定，类型转换，SpEL；

测试：模拟对象，TestContext 框架，SpringMVC 测试，WebTestClient；

数据访问：

Web 支持：Spring MVC

集成：

语言：

2、Spring 模块

Spring Core：

Spring Aspects：

Spring AOP：

Spring JDBC：

Spring JMS：

Spring ORM：

Spring Web：

Spring Test：

3、IOC

IoC（Inverse of Control）控制反转，可以理解为将原本在程序中手动创建的对象的控制权，交由 Spring 框架来管理。

IoC 容器控制了对象；主要控制了外部资源获取（不只是对象包括比如文件等）。

由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转

4、依赖注入

DI—Dependency Injection，即“依赖注入”：组件之间依赖关系由容器在运行期决定，形象的说，即**由容器动态的将某个依赖关系注入到组件之中。依赖注入的目的并非为软件系统带来更多功能，而是为了提升组件重用的频率，并为系统搭建一个灵活、可扩展的平台；**

谁依赖于谁：当然是**应用程序依赖于 IoC 容器**；

为什么需要依赖：**应用程序需要 IoC 容器来提供对象需要的外部资源**；

谁注入谁：很明显是 **IoC 容器注入应用程序某个对象，应用程序依赖的对象**；

注入了什么：就是**注入某个对象所需要的外部资源（包括对象、资源、常量数据）**。

依赖注入的方式：

1、使用属性的 setter 方法注入 这是最常用的方式；

2、使用构造器注入；

3、使用 Filed 注入（用于注解方式

4、自动装配

5、AOP

AOP（Aspect-Oriented Programming）面向切面编程，将业务模块共同调用的逻辑或责任（例如事务处理、日志管理、权限管理等）封装起来，便于减少系统重复的代码，降低模块间的耦合度，有利于拓展和维护。

Spring AOP 基于动态代理，如果要代理的对象实现了某个接口，那么 Spring AOP 会使用 JDK Proxy 去创建代理对象，而对于没有实现接口的对象，就无法使用 JDK Proxy 去进行代理，此时 AOP 会使用 Cglib 生成一个被代理对象的子类来作为代理。

Cglib

如果针对类做代理使用的是 Cglib；

即使针对接口做代理，也可以将代理方式配置成走 Cglib 的。

6、Spring MVC 工作原理（运行流程）

第一步：发起请求到前端控制器(DispatcherServlet)；

第二步：前端控制器请求 HandlerMapping 查找 Handler （可以根据 xml 配置、注解进行查找）；

第三步：处理器映射器 HandlerMapping 向前端控制器返回 Handler，HandlerMapping 会把请求映射为 HandlerExecutionChain 对象（包含一个 Handler 处理器（页面控制器）对象，多个 HandlerInterceptor 拦截器对象），通过这种策略模式，很容易添加新的映射策略；

第四步：前端控制器调用处理器适配器去执行 Handler；

第五步：处理器适配器 HandlerAdapter 将会根据适配的结果去执行 Handler；

第六步：Handler 执行完成给适配器返回 ModelAndView；

第七步：处理器适配器向前端控制器返回 ModelAndView （ModelAndView 是 springmvc 框架的一个底层对象，包括 Model 和 view）；

第八步：前端控制器请求视图解析器去进行视图解析 （根据逻辑视图名解析成真正的视图(jsp)），通过这种策略很容易更换其他视图技术，只需要更改视图解析器即可；

第九步：视图解析器向前端控制器返回 View；

第十步：前端控制器进行视图渲染（视图渲染将模型数据(在 ModelAndView 对象中)填充到 request 域）；

第十一步：前端控制器向用户响应结果。

7、@Autowired 注解和@Resource 注解的区别

1. @Autowired

由 spring 提供，只按照 byType 注入

2. @Resource

由 J2EE 提供，默认是按照 byName 自动注入

@Resource 有两个重要的属性，name 和 type：

Spring 将@Resource 注解的 name 属性解析为 bean 的名字，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，这时通过反射机制使用 byName 自动注入策略。

8、@Controller 和@RestController 的区别

1、@RestController 是@Controller，都可以用来表示 Spring 某个类的是否可以接收 HTTP 请求

2、@RestController 是@Controller 和@ResponseBody 的结合体，两个标注合并的作用

@ResponseBody 这个注解通常使用在控制层（controller）的方法上，其作用是将方法的返回值以特定的格式写入到 response 的 body 区域，进而将数据返回给客户端。

9、Spring 中的 BeanFactory 和 ApplicationContext 的区别

BeanFactory：

是 Spring 里面最低层的接口，提供了最简单的容器的功能，只提供了实例化对象和拿对象的功能；

ApplicationContext：

应用上下文，继承 BeanFactory 接口，它是 Spring 的一各更高级的容器，提供了更多的有用的功能；

1) 国际化（MessageSource）

2) 访问资源，如 URL 和文件（ResourceLoader）

3) 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

4) 消息发送、响应机制（ApplicationEventPublisher）

5) AOP（拦截器）

两者装载 bean 的区别

BeanFactory：

BeanFactory 在启动的时候不会去实例化 Bean，只有从容器中拿 Bean 的时候才会去实例化；

ApplicationContext：

ApplicationContext 在启动的时候就把所有的 Bean 全部实例化了。它还可以为 Bean 配置 lazy-init=true 来让 Bean 延迟实例化。

延迟实例化的优点：（**BeanFactory**）

应用启动的时候占用资源很少；对资源要求较高的应用，比较有优势；

不延迟实例化的优点： （**ApplicationContext**）

1. 所有的 Bean 在启动的时候都加载，系统运行的速度快；

2. 在启动的时候所有的 Bean 都加载了，我们就能在系统启动的时候，尽早的发现系统中的配置问题

3. 建议 web 应用，在启动的时候就把所有的 Bean 都加载了。（把费时的操作放到系统启动中完成）

10、Spring 的自动装配

隐式的 bean 发现机制和自动装配

在 Java 代码或者 XML 中显式配置

11、Bean 的生命周期

Spring IOC 容器对 Bean 的生命周期进行管理的过程如下：

1、通过构造器或工厂方法创建 Bean 实例

2、为 Bean 的属性设置值和对其它 Bean 的引用

3、调用 Bean 的初始化方法

4、Bean 可以使用了

5、当容器关闭时，调用 Bean 的销毁方法

在 Bean 的声明里设置 init-method 和 destroy-method 属性, 为 Bean 指定初始化和销毁方法。

12、Spring Bean 作用域的区别

singleton：单例模式，在整个 Spring IoC 容器中，使用 singleton 定义的 Bean 将只有一个实例

prototype：原型模式，每次通过容器的 getBean 方法获取 prototype 定义的 Bean 时，都将产生一个新的 Bean 实例

request：对于每次 HTTP 请求，使用 request 定义的 Bean 都将产生一个新实例，即每次 HTTP 请求将会产生不同的 Bean 实例。只有在 Web 应用中使用 Spring 时，该作用域才有效

session：对于每次 HTTP Session，使用 session 定义的 Bean 豆浆产生一个新实例。同样只有在 Web 应用中使用 Spring 时，该作用域才有效

globalsession：每个全局的 HTTP Session，使用 session 定义的 Bean 都将产生一个新实例。典型情况下，仅在使用 portlet context 的时候有效。同样只有在 Web 应用中使用 Spring 时，该作用域才有效

13、Spring 支持的常用数据库事务传播属性和事务隔离级别

Spring 实现方式

1、编程式事务管理对基于 POJO 的应用来说事唯一选择，需要在代码中调用 beginTransaction、commit、rollback 等事务管理相关的方法

2、基于 TransactionProxyFactoryBean 的声明式事务管理

3、基于 Transcational 的声明式事务管理

4、基于 Aspectj AOP 配置事务

事务传播行为

事务传播行为（为了解决业务层方法之间互相调用的事务问题）： 当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。在 TransactionDefinition 定义中包括了如下几个表示传播行为的常量：

1） PROPAGATION_REQUIRED，默认的 spring 事务传播级别，使用该级别的特点是，如果上下文中已经存在事务，那么就加入到事务中执行，如果当前上下文中不存在事务，则新建事务执行。所以这个级别通常能满足处理大多数的业务场景。

2）PROPAGATION_SUPPORTS ，从字面意思就知道，supports，支持，该传播级别的特点是，如果上下文存在事务，则支持事务加入事务，如果没有事务，则使用非事务的方式执行。所以说，并非所有的包在 transactionTemplate.execute 中的代码都会有事务支持。这个通常是用来处理那些并非原子性的非核心业务逻辑操作。应用场景较少。

3）PROPAGATION_MANDATORY ， 该级别的事务要求上下文中必须要存在事务，否则就会抛出异常！配置该方式的传播级别是有效的控制上下文调用代码遗漏添加事务控制的保证手段。比如一段代码不能单独被调用执行，但是一旦被调用，就必须有事务包含的情况，就可以使用这个传播级别。

4）PROPAGATION_REQUIRES_NEW ，从字面即可知道，new，每次都要一个新事务，该传播级别的特点是，每次都会新建一个事务，并且同时将上下文中的事务挂起，执行当前新建事务完成以后，上下文事务恢复再执行。

这是一个很有用的传播级别，举一个应用场景：现在有一个发送 100 个红包的操作，在发送之前，要做一些系统的初始化、验证、数据记录操作，然后发送 100 封红包，然后再记录发送日志，发送日志要求 100%的准确，如果日志不准确，那么整个父事务逻辑需要回滚。怎么处理整个业务需求呢？就是通过这个 PROPAGATION_REQUIRES_NEW 级别的事务传播控制就可以完成。发送红包的子事务不会直接影响到父事务的提交和回滚。

5）PROPAGATION_NOT_SUPPORTED ，这个也可以从字面得知，not supported ，不支持，当前级别的特点就是上下文中存在事务，则挂起事务，执行当前逻辑，结束后恢复上下文的事务。

这个级别有什么好处？可以帮助你将事务极可能的缩小。我们知道一个事务越大，它存在的风险也就越多。所以在处理事务的过程中，要保证尽可能的缩小范围。比如一段代码，是每次逻辑操作都必须调用的，比如循环 1000 次的某个非核心业务逻辑操作。这样的代码如果包在事务中，势必造成事务太大，导致出现一些难以考虑周全的异常情况。所以这个事务这个级别的传播级别就派上用场了。用当前级别的事务模板抱起来就可以了。

6）PROPAGATION_NEVER ，该事务更严格，上面一个事务传播级别只是不支持而已，有事务就挂起，而 PROPAGATION_NEVER 传播级别要求上下文中不能存在事务，一旦有事务，就抛出 runtime 异常，强制停止执行！这个级别上辈子跟事务有仇。

7）PROPAGATION_NESTED ，字面也可知道，nested，嵌套级别事务。该传播级别特征是，如果上下文中存在事务，则嵌套事务执行，如果不存在事务，则新建事务。

数据可事务并发

脏读：事务 A 读取了事务 B 已经修改但尚未提交的数据。若事务 B 回滚数据，事务 A 的数据存在不一致性的问题。

可重复读：事务 A 第一次读取最初数据，第二次读取事务 B 已经提交的修改或删除数据。导致两次读取数据不一致。不符合事务的隔离性。

幻读：事务 A 根据相同条件第二次查询到事务 B 提交的新增数据，两次数据结果集不一致。不符合事务的隔离性。

隔离级别

TransactionDefinition 接口中定义了五个表示隔离级别的常量：

1、TransactionDefinition.ISOLATION_DEFAULT: 使用后端数据库默认的隔离级别，Mysql 默认采用的 REPEATABLE_READ 隔离级别 Oracle 默认采用的 READ_COMMITTED 隔离级别.

2、TransactionDefinition.ISOLATION_READ_UNCOMMITTED: 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读

3、TransactionDefinition.ISOLATION_READ_COMMITTED: 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生

4、TransactionDefinition.ISOLATION_REPEATABLE_READ: 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。

5、TransactionDefinition.ISOLATION_SERIALIZABLE 最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。但是这将严重影响程序的性能。通常情况下也不会用到该级别。

14、SpringMVC 中如何解决 post 请求中文乱码问题

15、ASM

汇编

ASM 是一个 java 字节码操纵框架，它能被用来动态生成类或者增强既有类的功能。ASM 可以直接产生二进制 class 文件，也可以在类被加载入 Java 虚拟机之前动态改变类行为。Java class 被存储在严格格式定义的 .class 文件里，这些类文件拥有足够的元数据来解析类中的所有元素：类名称、方法、属性以及

Java 字节码（指令）。ASM 从类文件中读入信息后，能够改变类行为，分析类信息，甚至能够根据用户要求生成新类。

ASM 框架中的核心类有以下几个：

1、ClassReader:该类用来解析编译过的 class 字节码文件。

2、ClassWriter:该类用来重新构建编译后的类，比如说修改类名、属性以及方法，甚至可以生成新的类的字节码文件。

3、ClassAdapter:该类也实现了 ClassVisitor 接口，它将对它的方法调用委托给另一个 ClassVisitor 对象。

16、Spring 基础配置

1、依赖注入

声明 Bean 的注解

@Component 组件　　没有明确的角色

@Service　　在业务逻辑层（service 层）使用

@Respository　　在数据访问层（dao 层）使用

@Controller　　在展现层（MVC）使用

注入 Bean 的注解，

@Autowired　　Spring 提供的注解

@Inject　　JSR-330 提供的注解

@Resource　　JSR-250 提供的注解

2、Java 配置

@Configuration　　声明当前类是一个配置类，相当于一个 Spring 配置的 xml 文件

@Bean　　注解在方法上，声明当前方法的返回值为一个 Bean

3、AOP

Spring 支持 AspectJ 的注解式切面编程

1、使用@Aspect 声明是一个切面

2、使用@After，@Before，@Around 定义建言（advice），可直接将拦截规则（切点）作为参数

3、其中@After，@Before，@Around 参数的拦截规则为（PointCut），为了使切点复用，可使用@PonitCut 专门定义拦截规则，然后在@After，@Before，@Around 的参数中调用

4、其中符合条件的每个被拦截处为连接点（JoinPoint）

4、SpringMVC 常用注解

@ResquestMapping　用于映射 Web 请求（访问路径和参数）、处理类和方法的。@ ResquestMapping 支持 Servlet 的 request 和 response 作为参数，也支持对 request 和 response 的媒体类型进行配置

@ResponseBody　支持将返回值放在 response 体内，而不是返回一个页面。

@ResquestBody

@PathVariable　用来接收路径参数，此注解放置在参数前

@RestController　是一个组合注解，组合了@Controller 和@ResponseBody，开发一个和页面交互数据的控制时，需要使用此注解

5、SpringMVC 基本配置

SpringMVC 的定制配置需要配置类继承一个 WebMvcConfigurerAdapter 类，并在此类使用@EnableWebMvc 注解，开启对 SpringMVC 的配置支持

6、SpringBoot 基本配置

SpringBoot 通常有一个名为*Application 的入口类，入口类里有个 main 方法，在 main 方法中使用 SpringApplication.run()方法，启动 SpringBoot 项目

@SpringBootApplication 是 SpringBoot 的核心注解

7、SpringBoot 的配置文件

SpringBoot 使用一个全局的配置文件 application.properties 或 application.yml

8、类型安全的配置

SpringBoot 提供基于类型安全的配置方式，通过@ConfigurationProperties 将 properties 属性和一个 Bean 及其属性关联，从而实现类型安全的配置

数据库

1、关系型数据库的三个范式

第一范式（1NF）：要求数据库表的每一列都是不可分割的原子数据项。

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）；

第二范式需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。

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）；

第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。

2、事务的四大特征

事务：并发控制的单位，是用户定义的一个操作序列。

原子性：事物内部的操作不可分割；

一致性：要么都成功，要么都失败，后面的失败要对前面的操作进行回滚；

隔离性：一个事务开始后，不能对其他事务干扰；

持久性：事务开始后就不能终止。

3、char 与 varchar 的区别、float 与 double 的区别

1、char 是定长的，也就是当你输入的字符小于你指定的数目时，char(8)，你输入的字符小于 8 时，它会再后面补空值。当你输入的字符大于指定的数时，它会截取超出的字符。

2、varchar 存储变长数据，但存储效率没有 char 高。

4、MySql 的默认最大连接数

100

5、MySql 的分页

MySql.使用关键字 limit 来进行分页，limit offset，size，从多少索引去多少位。

6、如何避免 SQL 的注入

1、PreparedStatement

2、使用正则表达式过滤传入的参数

3、字符串过滤

4、JSP 中调用该函数检查是否包含非法字符

5、JSP 页面判断代码

7、数据库触发器的使用机制

8、数据库事务隔离

Read Uncommitted（读取未提交内容）

在该隔离级别，所有事务都可以看到其他未提交事务的执行结果。本隔离级别很少用于实际应用，因为它的性能也不比其他级别好多少。读取未提交的数据，也被称之为脏读（Dirty Read）；

Read Committed（读取提交内容）

这是大多数数据库系统的默认隔离级别（但不是 MySQL 默认的）。它满足了隔离的简单定义：一个事务只能看见已经提交事务所做的改变。这种隔离级别 也支持所谓的不可重复读（Nonrepeatable Read），因为同一事务的其他实例在该实例处理其间可能会有新的 commit，所以同一 select 可能返回不同结果；

Repeatable Read（可重读）

这是 MySQL 的默认事务隔离级别，它确保同一事务的多个实例在并发读取数据时，会看到同样的数据行。不过理论上，这会导致另一个棘手的问题：幻读 （Phantom Read）。简单的说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影” 行。

InnoDB 和 Falcon 存储引擎通过多版本并发控制（MVCC，Multiversion Concurrency Control）机制解决了该问题

Serializable（可串行化）

这是最高的隔离级别，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，它是在每个读的数据行上加上共享锁。在这个级别，可能导致大量的超时现象和锁竞争。

9、数据库存储过程

存储过程的优点：

只在创建时进行编译，以后每次执行都不再重新编译，提高执行速度；

复杂的业务逻辑需要多条 SQL 语句，将这些操作存放在一个存储过程中，客户端和服务器的网络传输会大大减少，降低网络负载；

存储过程创建一次就可以重复使用，减少工作量；

存储过程可以屏蔽对底层数据库对象的直接访问，安全性高。

```
CREATE PROCEDURE GetContactFormalNames
AS
BEGIN
SELECT TOP 10 Title + ' ' + FirstName + ' ' + LastName AS FormalName FROM Person.Contact
END
```

调用语句 call GetContactFormalNames

10、对 JDBC 的理解

Java 只定义接口，数据库厂商自己实现接口，开发者只需要导入对应厂商开发的实现，然后以接口方式进行调用。

PreparedStatement 和 Statement

PreparedStatement 是预编译的，比 Statement 速度快；

PreparedStatement 可读性和可维护性更好；

PreparedStatement 可防止 SQL 注入攻击。

11、数据库连接池

限定数据库连接的个数，不会导致由于数据库连接过多而系统运行缓慢或崩溃；

减少在创建和销毁数据库连接所需的开销；

数据库连接不需要每次都去创建。

12、数据库优化方法

定位：查询、定位慢查询，并优化

在项目自验项目转测试之前，在启动 mysql 数据库时开启慢查询，并且把执行慢的语句写到日志中，在运行一段时间后，通过查看日志找到慢查询语句。

使用 explain 慢查询语句，详细分析语句的问题。

优化手段：

数据库设计需要遵循范式；

选择合适的数据库引擎

1、创建索引：

索引的弊端：

占用磁盘空间；

对插入，修改和删除操作有影响，变慢

使用场景：

a. where 条件经常使用

b. 该字段内容不是唯一的几个值

c. 字段内容不是频繁变化

具体技巧：

2、分页：

水平分表（按行），垂直分表（按列）

3、读写分离：

4、缓存：

5、一些常用的优化技巧：

13、如何实现批量插入几百万条数据

变多次提交为一次

使用批量操作

14、MyBatis 中当实体类中的属性名与表中的字段名不一致

1、写 sql 语句时起别名

2、在 MyBatis 全局配置文件中开启驼峰命名规则

3、在 mapper 映射文件中使用 resultMap 来定义映射规则

15、Mysql 什么时候建立索引，有哪几种索引

索引（index）是帮助数据库高效获取数据的数据结构。

索引种类

FULLTEXT，HASH，BTREE，RTREE

FULLTEXT：即为全文索引，目前只有 MyISAM 引擎支持，目前只有 CHAR、VARCHAR，TEXT 列上可以创建全文索引;

Hash :

BTREE：将索引值按一定的算法，存入一个树形的数据结构中。

在 Innodb 里，有两种形态：一是 primary key 形态，其 leaf node 里存放的是数据，而且不仅存放了索引键的数据，还存放了其他字段的数据。二是 secondary index，其 leaf node 和普通的 BTREE 差不多，只是还存放了指向主键的信息.

而在 MyISAM 里，主键和其他的并没有太大区别。不过和 Innodb 不太一样的地方是在 MyISAM 里，leaf node 里存放的不是主键的信息，而是指向数据文件里的对应数据行的信息。

RTREE:

为什么建立索引：

1、通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

2、可以大大加快 数据的检索速度（大大减少的检索的数据量），这也是创建索引的最主要的原因。

3、帮助服务器避免排序和临时表

4、将随机 IO 变为顺序 IO

5、可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

16、Mysql 常用引擎

InnoDB 存储引擎

MyISAM 存储引擎

count 运算上的区别： 因为 MyISAM 缓存有表 meta-data（行数等），因此在做 COUNT(*)时对于一个结构很好的查询是不需要消耗多少资源的。而对于 InnoDB 来说，则没有这种缓存

是否支持事务和崩溃后的安全恢复： MyISAM 强调的是性能，每次查询具有原子性，其执行速度比 InnoDB 类型更快，但是不提供事务支持。但是 InnoDB 提供事务支持，外部键等高级数据库功能。 具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。

是否支持外键： MyISAM 不支持，而 InnoDB 支持。

总结：

MyISAM 更适合读密集的表，而 InnoDB 更适合写密集的的表。 在数据库做主从分离的情况下，经常选择 MyISAM 作为主库的存储引擎。

一般来说，如果需要事务支持，并且有较高的并发读取频率(MyISAM 的表锁的粒度太大，所以当该表写并发量较高时，要等待的查询就会很多了)，InnoDB 是不错的选择。如果你的数据量很大（MyISAM 支持压缩特性可以减少磁盘的空间占用），而且不需要支持事务时，MyISAM 是最好的选择。

MyISAM 与 InnoDB 关于锁方面的区别

MyISAM 默认使用表级锁，不支持行级锁

InnoDB 默认使用行级锁，也支持表级锁

17、Mysql 行锁和表锁

MySQL 表级锁有两种模式：表共享锁（Table Read Lock）和表独占写锁（Table Write Lock）。

对 MyISAM 的读操作，不会阻塞其他用户对同一表请求，但会阻塞对同一表的写请求；

对 MyISAM 的写操作，则会阻塞其他用户对同一表的读和写操作；

MyISAM 表的读操作和写操作之间，以及写操作之间是串行的。

当一个线程获得对一个表的写锁后，只有持有锁线程可以对表进行更新操作。其他线程的读、写操作都会等待，直到锁被释放为止。

18、SQL 关键字的执行顺序

FROM>ON>JOIN>WHERE>GROUP BY>WITH CUBE or WITH ROLLUP>HAVING>SELECT>DISTINCT>UNION>ORDER BY>TOP

19、MySQL 内连接、左连接和右连接

内连接是把匹配的关联数据显示出来；左连接是左边的表全部显示出来，右边的表显示符合条件的数据；右连接正好相反。

20、Explain 包含哪些列，Explain 的 Type 列有哪几种值

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、indexhe 和 ALL

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。这个列表是在优化过程的早期创建的，因此有些列出来的索引可能对于后续优化过程是没有用的。

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

如果该索引没有出现在 possible_keys 中，那么 mysql 选用的是出于另外的原因——例如，他可能选择了一个覆盖索引，哪怕没有 where 字句，换句话说，possible_keys 揭示了哪一个索引能有助于提高查询效率，而 key 显示的是优化采用哪一个索引可以最小化查询成本。

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。

Type 列的解释

Type：告诉我们对表使用的访问方式，主要包含如下集中类型；

all：全表扫描；

const：读常量，最多只会有一条记录匹配，由于是常量，实际上只须要读一次；

eq_ref：最多只会有一条匹配结果，一般是通过主键或唯一键索引来访问；

fulltext：进行全文索引检索；

index：全索引扫描；

index_merge：查询中同时使用两个（或更多）索引，然后对索引结果进行合并（merge），再读取表数据；

index_subquery：子查询中的返回结果字段组合是一个索引（或索引组合），但不是一个主键或唯一索引；

rang：索引范围扫描；

ref：Join 语句中被驱动表索引引用的查询；

ref_or_null：与 ref 的唯一区别就是在使用索引引用的查询之外再增加一个空值的查询；

system：系统表，表中只有一行数据；

unique_subquery：子查询中的返回结果字段组合是主键或唯一约束。

21、Hash 索引的缺点

仅能满足“=”，“IN”，不能使用范围查询

无法被用来避免数据的的排序操作

不能利用部分索引键查询

不能避免表扫描

遇到大量 Hash 值相等的情况后性能不一定就比 B-Tree 索引号

23、MyBatis 中的#{}和${}的区别

1、#{}是编译预处理，${}是字符串转；

2、使用#{}可以有效防止 SQL 注入，提高系统安全性

3、Mybatis 在处理${}时，就是把${}替换成变量的值；

4、MyBatis 在处理#{}时，会将 sql 中的#{}替换为？，调用 PreparedStatement 的 set 方法来赋值

24、MyBatis 的分页方式

数组分页

Sql 分页 （物理分页）

拦截器分页

RowBounds 分页（逻辑分页）针对 ResultSet 结果集执行的内存分页

25、MyBatis 逻辑分页和物理分页

物理分页：

物理分页依赖于某一物理实体，这个实体就是数据库，比如 MySQl 数据库提供了 limit 关键字，程序只需要编写带有 limit 的 SQL 语句，数据库返回的就是分页结果。

逻辑分页：

逻辑分页依赖的是程序员编写的代码。数据库返回的不是分页结果，二是全部数据，然后再由程序员通过代码获取分页数据，常用的操作时一次性从数据库中查询出全部数据并存储到 list 集合中，因为 list 集合有序，再根据索引获取指定范围的数据。

物理分页速度不一定快于逻辑分页，逻辑分页速度不一定快于物理分页

物理分页优于逻辑分页：没必要将属于数据库端的压力加到应用端

1：逻辑分页 内存开销比较大,在数据量比较小的情况下效率比物理分页高;在数据量很大的情况下,内存开销过大,容易内存溢出,不建议使用

2：物理分页 内存开销比较小,在数据量比较小的情况下效率比逻辑分页还是低,在数据量很大的情况下,建议使用物理分页

26、Mybatis 延迟加载

应用场景：

1.假如一个用户他有 N 个订单（N>=1000）,那么如果一次性加载的话，一个用户对象的订单集合 OrderList 里面就会有 1000 多个 Order 的对象。计算：一个订单对象里面数据有多大

2.从优化考虑，我们会使用延时加载去降低内存的占用，即先查询用户，不查询订单，当你通过 用户.getOrderList（）的时候，再查询订单集合

总而言之一句话：查询的时候，只查询当前对象，不查询关联对象，即为延时加载。

27、MyBatis 一级缓存和二级缓存

一级缓存

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

二级缓存

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

UserMapper 有一个二级缓存区域（按 namespace 分），其它 mapper 也有自己的二级缓存区域（按 namespace 分）。每一个 namespace 的 mapper 都有一个

二级缓存区域，两个 mapper 的 namespace 如果相同，这两个 mapper 执行 sql 查询到数据将存在相同的二级缓存区域中。

Redis

1、Redis 和 Redis 使用场景

Redis 是一种基于键值对（key-value）的 NoSQL 数据库，与很多键值对数据库不同的地方在于，Redis 中的值可以是由 string（字符串）、hash（哈希）、list（列表）、set（集合）、zset（有序集合）等数据结构组成。因此，Redis 可以满足很多的应用场景。

应用场景：

1、缓存

2、排行榜系统

由于一个玩家名次上升 x 位将会引起 x+1 位玩家的名次发生变化（包括该玩家），如果采用传统数据库（比如 MySQL）来实现排行榜，当玩家人数较多时，将会导致对数据库的频繁修改，性能得不到满足。

3、计数器应用

4、社交网络

5、消息队列系统

2、Redis 线程模型，异步队列的实现

redis 与 memcached 区别

redis 数据类型更多

redis 单线程，memcached 多线程

memcached 无原生集群

单线程模型

非阻塞 IO 复用

文件事件处理器，基于内存操作

无多线程上下文切换的问题

如何使用 Redis 实现异步队列

使用 List 作为队列，RPUSH 生产消息，LPOP 消费消息

缺点：没有等待队列里有值就直接消费

弥补：可以通过在应用层引入 Sleep 机制调用 LPOP 重试

BLPOP key[key] timeout: 阻塞队列直到队列有消息或超时

缺点：只能一个消费者

pub/sub:发布订阅者

缺点：消息发布无状态，无法保证可达

3、Redis 对象保存方式

1.Json 字符串：

需要把对象转换为 json 字符串，当做字符串处理。直接使用 set get 来设置。

优点：设置和获取比较简单

缺点：没有提供专门的方法，需要把把对象转换为 json。

2.字节：

需要做序列号，就是把对象序列化为字节保存。

如果是担心 JSON 转对象会消耗资源的情况，这个问题需要考量几个地方，

第一点：就是使用的 JSON 转换 lib 是否就会存在性能问题。

第二点：就是数据的数据量级别，如果是存储百万级的大数据对象，建议采用存储序列化对象方式。如果是少量的数据级对象，或者是数据对象字段不多，还是建议采用 JSON 转换成 String 方式。毕竟 redis 对存储字符类型这部分优化的非常好。具体采用的方式与方法，还要看你所使用的场景。

4、Redis 数据淘汰机制

volatile-lru：使用 LRU 算法进行数据淘汰（淘汰上次使用时间最早的，且使用次数最少的 key），只淘汰设定了有效期的 key

allkeys-lru：使用 LRU 算法进行数据淘汰，所有的 key 都可以被淘汰

volatile-random：随机淘汰数据，只淘汰设定了有效期的 key

allkeys-random：随机淘汰数据，所有的 key 都可以被淘汰

volatile-ttl：淘汰剩余有效期最短的 key

5、Redis 高并发高可用

主从复制

主 redis 中的数据有两个副本（replication）即从 redis1 和从 redis2，即使一台 redis 服务器宕机其它两台 redis 服务也可以继续提供服务。

主 redis 中的数据和从 redis 上的数据保持实时同步，当主 redis 写入数据时通过主从复制机制会复制到两个从 redis 服务上。

只有一个主 redis，可以有多个从 redis。

主从复制不会阻塞 master，在同步数据时，master 可以继续处理 client 请求。

一主一从：用于主节点故障转移从节点，当主节点的“写”命令并发高且需要持久化，可以只在从节点开启 AOF（主节点不需要），这样即保证了数据的安全性，也避免持久化对主节点的影响。

一主多从：针对“读”较多的场景，“读”由多个从节点来分担，但节点越多，主节点同步到多节点的次数也越多，影响带宽，也加重主节点的稳定。

主从复制断点续传

异步复制导致数据丢失

Redis 哨兵主要功能

（1）集群监控：负责监控 Redis master 和 slave 进程是否正常工作

（2）消息通知：如果某个 Redis 实例有故障，那么哨兵负责发送消息作为报警通知给管理员

（3）故障转移：如果 master node 挂掉了，会自动转移到 slave node 上

（4）配置中心：如果故障转移发生了，通知 client 客户端新的 master 地址

Redis 哨兵的高可用

原理：当主节点出现故障时，由 Redis Sentinel 自动完成故障发现和转移，并通知应用方，实现高可用性。

缓存一致性

集群脑裂

6、Redis 持久化

RDB

在指定的时间间隔内将内存中的数据集快速写入磁盘，也就是 Snapshot 快照，恢复时将快照文件直接读到内存中。

优点：节省磁盘空间，回复速度快

缺点：虽然 Redis 在 fork 时使用了写时拷贝技术，但是如果数据量庞大时，比较消耗性能；

可能会丢失最后一次快照之后的数据。

AOF

以日志的方式记录每个写操作，只追加文件。

优点：丢失数据概率更低，

可读的日志文件，可以处理误操作

缺点：比起 RDB 占用更多的磁盘空间；

恢复备份数据速度慢

每次读写同步的话，有一定的性能压力

存在个别 bug，造成无法恢复

7、Redis 分布式锁

分布式锁需要解决的问题

互斥性

安全性

死锁

容错

SETNX key value：如果 key 不存在，则创建并复制

时间复杂度：O(1)

返回值：设置成功，返回 1；设置失败，返回 0。

如何解决 SETNX 长期有效的问题

大量 key 同时过期的注意事项

集中过期，由于清除大量的 key 很耗时，会出现短暂的卡顿现象

解决方案：在设置 key 的过期事件的时候，给 每个 key 加上随机值

8、Redis 内存优化

Redis 主进程的内存消耗：

Redis 自身使用的内存：消耗很少，3MB 多点

对象内存、缓冲内存、内存碎片

内存管理：

1、设置内存上限，并指定内存回收策略；

2、maxmemory 配置参数可限制当前 Redis 实例可使用的最大内存；

3、通过 config set maxmemory 可根据业务需求，动态调整内存限制；

4、通过设置内存上限，可方便地在一台服务器上部署多个 Redis 实例

内存优化：

1、Redis 存储的所有数据都使用 redisObject 来封装，包括 string、hash、list、set、zset；

2、字符串长度在 39 字节以内对象，在创建 redisObject 封装对象时只需分配内存 1 次，可提高性能；

3、缩减键、值对象的长度：简化键名，使用高效的序列化工具来序列化值对象，还可使用压缩工具(Google Snappy)压缩序列化后的数据；

4、共享对象池：Redis 内部维护[0-9999]的整数对象池，对于 0-9999 的内部整数类型的元素、整数值对象都会直接引用整数对象池中的对象，因此尽量使用整数对象可节省内存；

消息队列

1、消息队列的选型

解耦：使利用发布-订阅模式工作，消息发送者（生产者）发布消息，一个或多个消息接受者（消费者）订阅消息。消息发送者将消息发送至分布式消息队列即结束对消息的处理，消息接受者从分布式消息队列获取该消息后进行后续处理，并不需要知道该消息从何而来。对新增业务，只要对该类消息感兴趣，即可订阅该消息，对原有系统和业务没有任何影响，从而实现网站业务的可扩展性设计。

异步：在使用消息队列之后，用户的请求数据发送给消息队列之后立即 返回，再由消息队列的消费者进程从消息队列中获取数据，异步写入数据库。由于消息队列服务器处理速度快于数据库（消息队列也比数据库有更好的伸缩性），因此响应速度得到大幅改善。

削峰：通过异步处理，将短时间高并发产生的事务消息存储在消息队列中，从而削平高峰期的并发事务。

问题：

消息存放在哪里？特别是高峰期巨量的消息占用多少空间？会爆满么？

消息队列的 topic

2、消息队列如何保证高可用性

RabbitMQ 的高可用性

镜像集群模式

Kafka 的高可用性

避免消息丢失

为了避免消息队列服务器宕机造成消息丢失，会将成功发送到消息队列的消息存储在消息生产者服务器上，等消息真正被消费者服务器处理后才删除消

息。在消息队列服务器宕机后，生产者服务器会选择分布式消息队列服务器集群中的其他服务器发布消息。

消息队列的问题：

系统可用性降低： 系统可用性在某种程度上降低，为什么这样说呢？在加入 MQ 之前， 不用考虑消息丢失或者说 MQ 挂掉等等的情况，但是，引入 MQ 之后你就需要去考虑了！

系统复杂性提高： 加入 MQ 之后， 需要保证消息没有被重复消费、处理消息丢失的情况、保证消息传递的顺序性等等问题！

一致性问题： 消息队列可以实现异步，消息队列带来的异步确实可以提高系统响应速度。但是，万一消息的真正消费者并没有正确消费消息怎么办？这样就会导致数据不一致的情况了!

3、如何保证消息不被重复消费？

1、比如你拿个数据要写库，你先根据主键查一下，如果这数据都有了，你就别插入了，update 一下好吧。

2、比如你是写 Redis，那没问题了，反正每次都是 set，天然幂等性。

3、比如你不是上面两个场景，那做的稍微复杂一点，你需要让生产者发送每条数据的时候，里面加一个全局唯一的 id，类似订单 id 之类的东西，然后你这里消费到了之后，先根据这个 id 去比如 Redis 里查一下，之前消费过吗？如果没有消费过，你就处理，然后这个 id 写 Redis。如果消费过了，那你就别处理了，保证别重复处理相同的消息即可。

4、比如基于数据库的唯一键来保证重复数据不会重复插入多条。因为有唯一键约束了，重复数据插入只会报错，不会导致数据库中出现脏数据。

4、如何保证消息的可靠性传输（如何处理消息丢失的问题）？

5、怎么保证从消息队列里拿到的数据按顺序执行？

（1）rabbitmq

1、拆分多个 queue，每个 queue 一个 consumer，就是多一些 queue 而已，确实是麻烦点；这样也会造成吞吐量下降，可以在消费者内部采用多线程的方式取消费。

2、或者就一个 queue 但是对应一个 consumer，然后这个 consumer 内部用内存队列做排队，然后分发给底层不同的 worker 来处理

（2）kafka

1、确保同一个消息发送到同一个 partition，一个 topic，一个 partition，一个 consumer，内部单线程消费。

2、写 N 个内存 queue，然后 N 个线程分别消费一个内存 queue 即可

6、如何解决消息队列的延时以及过期失效问题？消息队列满了以后该怎么处理？有几百万消息持续积压几小时，说说怎么解决？

延时问题：

先修复 consumer 的问题，确保其恢复消费速度，然后将现有 cnosumer 都停掉。

新建一个 topic，partition 是原来的 10 倍，临时建立好原先 10 倍的 queue 数量。

然后写一个临时的分发数据的 consumer 程序，这个程序部署上去消费积压的数据，消费之后不做耗时的处理，直接均匀轮询写入临时建立好的 10 倍数量的 queue。

接着临时征用 10 倍的机器来部署 consumer，每一批 consumer 消费一个临时 queue 的数据。这种做法相当于是临时将 queue 资源和 consumer 资源扩大 10 倍，以正常的 10 倍速度来消费数据。

等快速消费完积压数据之后，得恢复原先部署的架构，重新用原先的 consumer 机器来消费消息。

过期失效：

批量重导

7、如果开发一个消息队列中间件，如何设计架构？

1、首先这个 mq 得支持可伸缩性吧，就是需要的时候快速扩容，就可以增加吞吐量和容量，那怎么搞？设计个分布式的系统呗，参照一下 kafka 的设计理念，broker -> topic -> partition，每个 partition 放一个机器，就存一部分数据。如果现在资源不够了，简单啊，给 topic 增加 partition，然后做数据迁移，增加机器，不就可以存放更多数据，提供更高的吞吐量了？

2、其次你得考虑一下这个 mq 的数据要不要落地磁盘吧？那肯定要了，落磁盘才能保证别进程挂了数据就丢了。那落磁盘的时候怎么落啊？顺序写，这样就没有磁盘随机读写的寻址开销，磁盘顺序读写的性能是很高的，这就是 kafka 的思路。

3、其次你考虑一下你的 mq 的可用性啊？这个事儿，具体参考之前可用性那个环节讲解的 kafka 的高可用保障机制。多副本 -> leader & follower -> broker 挂了重新选举 leader 即可对外服务。

4、能不能支持数据 0 丢失啊？可以的，参考我们之前说的那个 kafka 数据零丢失方案。

Linux

1、Linux 体系结构

用户态和内核态

内核：本质是一段管理计算机硬件设备的程序

系统调用：

公共函数库:

Shell:命令解释器

2、Linux 下如何进行进程调度

Linux 进程的时间片与权重参数

在处理器资源有限的系统中，所有进程都以轮流占用处理器的方式交叉运行。为使每个进程都有运行的机会，调度器为每个进程分配了一个占用处理器的时间额度，这个额度叫做进程的“时间片”，其初值就存放在进程控制块的 counter 域中。进程每占用处理器一次，系统就将这次所占用时间从 counter 中扣除，因为 counter 反映了进程时间片的剩余情况，所以叫做剩余时间片。

Linux 调度的主要思想为：调度器大致以所有进程时间片的总和为一个调度周期；在每个调度周期内可以发生若干次调度，每次调度时，所有进程都以 counter 为资本竞争处理器控制权，counter 值大者胜出，优先运行；凡是已耗尽时间片（即 counter=0）的，则立即退出本周期的竞争；当所有未被阻塞进程的时间片都耗尽，那就不等了。然后，由调度器重新为进程分配时间片，开始下一个调度周期。

如果一个进程的剩余时间片多，那么它在前期获得运行的时间片就少，为了公平性，就应该适当的赋予它更高的优先级。但是这仅仅是一个总体的原则，为了应付实际问题中的特殊状况，在上述平等原则的基础上，为了给特殊进程

一些特殊照顾，在为进程分配时间片时，Linux 允许用户通过一个系统调用 nice()来设置参数 nice 影响进程的优先权。所以，系统真正用来确定进程的优先权时，使用的依据为权重参数 weight，weight 大的进程优先运行。

weight 与 nice、counter 之间的关系大体如下：

weight 正比于 [counter+(20-nice)]

nice 是用户在创建进程时确定的，在进程的运行过程中一般不会改变，所以叫做**静态优先级**；counter 则随着进程时间片的小号在不断减小，是变化的，所以叫做**动态优先级**。

普通进程调度策略

实时进程调度策略

凡是进程控制块的 policy 的值为 SCHED_FIFO 或 SCHED_RR 的进程，调度器都将其视为实时进程。

进程控制块中的域 rt_pripority 就是实时进程的优先级，其范围时 1~99。这一属性将在调度时用于对进程权重参数 weight 的计算。

普通进程参数 counter、nice，实时进程参数 rt_pripority

Linux 是按照严格的优先级别来选择待运行进程的，并且实时进程的权重 weight 远高于普通进程

Linux 把所有的实时进程按照其优先级组织成若干个队列，**对于同一队列的实时进程可以采用先来先服务和时间片轮转调度两种调度策略。**

先来先服务调度

该策略是符合 POSIX 标准的 FIFO（先入先出）调度规则。即在同一级别的队列中，先就绪的进程先入队，后就绪的进程后入队。

时间片轮转调度

根据优先级别把就绪进程分成若干个队列，但每个队列被组织成一个循环队列，并且每个进程都拥有一个时间片。调度时，调度器仍然以优先级别为序查询就绪进程对列。当发现就绪进程队列后，也是运行处在队列头部的进程，但是当这个进程将自己的时间片耗完之后，调度器把这个进程放到其队列的队尾，并且再选择处在队头的进程来运行，当然前提条件是这时没有更高优先级别的实时进程就绪。

3、Linux 中常用命令

1、服务类的相关命令

chkconfig

chkconfig 命令主要用来更新（启动或停止）和查询系统服务的运行级信息。谨记 chkconfig 不是立即自动禁止或激活一个服务，它只是简单的改变了符号连接。

service

service 服务名 status

systemctl

systemctl status 服务名

2、查询帮助命令

Help　查看 Linux 内置命令的帮助

3、文件和目录操作命令：

ls　列出目录内容以及内容属性信息

cd　切换文件夹

cp 复制文件或目录

find 查找

mkdir 创建目录

mv 移动或重命名文件

rename 重命名文件

rm 删除

touch 创建空文件，改变已有文件的时间戳属性

dirname 显示文件或者目录路径

4、查看文件及内容处理命令

cat 连接多个文件并且打印到屏幕输出或重定向到指定文件

more 分页显示文件内容

head 显示文件头部

tail 显示文件尾部

wc 统计文件的行数，单词数或字节数

grep | 过滤字符串

vi/vim 命令行文本编辑器

6、信息显示命令：

dmesg 显示开机信息

uptime 显示系统运行时间及负载

uname 显示操作系统相关信息的 命令

hostname 显示或者设置当前系统的主机名

stat 显示文件或文件系统的状态

du 计算磁盘空间的使用情况

df 报告文件系统磁盘空间的使用情况

top 实时显示系统资源使用情况

free 查看系统内存

date 和 cal 显示时间和日历

7、搜索文件命令

which 查找二进制命令，按环境变量 PATH 路径查找

find 从磁盘遍历查找文件或目录

8、系统权限与用户授权相关命令

chmod　改变文件或目录权限

ls -l　　查看文件权限

9、文件压缩与解压缩

tar 打包压缩

unzip 解压文件

gzip　gzip 压缩工具

zip 压缩工具

find * | grep “config”

在所有文件中找到文件名中含有 config 的文件

4、Linux 日志查看乱码的问题

5、Unix 中的 IO 模型

阻塞式 IO

应用进程受阻于内核提供的系统调用，该调用直到数据成功返回或者出错才返回（其他情况下不返回），这时阻塞结束。

非租塞式 IO

在非阻塞式中当应用进程调用系统接口时，如果数据没有准备好，则会返回一个标志来标识这种情况，这时系统应用知道数据没有准备好则不会一直阻塞，而是通过隔一段时间轮询一次，在两次轮询的间隙之间应用进程可以做其他的事情。

IO 复用

在之前的两个模式中，应用进程都是直接调用真正的 IO 系统接口，这个接口是面向应用进行直接读取硬件上的数据的。但是在 IO 复用模型中，应用进程直接调用的是一个选择器 select/poll，相当于在应用进程和直接 IO 系统调用之间添加了一个代理。

之前的阻塞和非阻塞模型由于是直接面向 IO 系统调用的，可以看成为其中有一个隐形的代理，但是只能代理一个 IO 通道；但是在 IO 复用模型中，该代理可以代理多个 IO 通道，所以复用的其实是 IO 通道 。当有一个 IO 通道可以进行读写时，则 select/poll 返回告诉应用进程，此时应用进程开始执行对应的读写操作，这里需要注意的是 select/poll 上的通道是需要应用进程自己去注册的，通道可以是读操作，也可以是写操作。

信号驱动式 IO

对应用进程的通知不是通过轮询实现的，而是使用信号机制来实现，这就使得在第一阶段，等待数据准备的时候，应用进程确确实实的不阻塞

异步 IO

上述 4 种 IO 模型，其不同点在于当接收缓冲区没有数据时，如何判断数据已经到来：阻塞式 IO 中 recvfrom 会阻塞直到接收缓冲区有数据；非阻塞式 IO 通过轮询 recvfrom 以判断接收缓冲区是否有数据；IO 复用中使用 select 或者

poll 以判断接收缓冲区是否有数据；信号驱动 IO 通过信号通知接收缓冲区是否有数据。

其相同点在于，IO 操作的第二个阶段，即从内核接收缓冲区向应用缓冲区复制数据时，调用 recvfrom 的进程会阻塞。

异步 IO 将数据复制过程也考虑进来，顺着信号驱动 IO 模型，将信号通知的时机放到数据复制完成之后，就是异步 IO 模型

6、进程切换时操作系统都会发生什么

1、（中断／异常等触发）正向模式切换并压入 PSW／PC 。（Program Status Word 程序状态字。program counter 程序计数器。指向下一条要执行的指令）

2、保存被中断进程的现场信息。

3、处理具体中断、异常。

4、把被中断进程的系统堆栈指针 SP 值保存到 PCB。（Stack Pointer 栈指针。Process Control Block 进程控制块。）

5、调整被中断进程的 PCB 信息，如进程状态）。

6、把被中断进程的 PCB 加入相关队列。

7、选择下一个占用 CPU 运行的进程。

8、修改被选中进程的 PCB 信息，如进程状态。

9、设置被选中进程的地址空间，恢复存储管理信息。

10、恢复被选中进程的 SP 值到处理器寄存器 SP。

11、恢复被选中进程的现场信息进入处理器。

12、（中断返回指令触发）逆向模式转换并弹出 PSW／PC。

数据结构与算法

数据结构

栈

队列

链表

哈希表

排序二叉树

红黑树

（1）每个节点或者是黑色，或者是红色。

（2）根节点是黑色。

（3）每个叶子节点（NIL）是黑色。[注意：这里叶子节点，是指为空(NIL 或 NULL)的叶子节点！]

（4）如果一个节点是红色的，则它的子节点必须是黑色的。

（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

B 树和 B+树

B 树：

1.根结点至少有两个子女。

2.每个中间节点都包含 k-1 个元素和 k 个孩子，其中 ceil（m/2） ≤ k ≤ m

3.每一个叶子节点都包含 k-1 个元素，其中 ceil（m/2） ≤ k ≤ m

4.所有的叶子结点都位于同一层。

5.每个节点中的元素从小到大排列，节点当中 k-1 个元素正好是 k 个孩子包含的元素的值域划分

6.每个结点的结构为：（n，A0，K1，A1，K2，A2，… ，Kn，An）

其中，Ki(1≤i≤n)为关键字，且 Ki<Ki+1(1≤i≤n-1)。

Ai(0≤i≤n)为指向子树根结点的指针。且 Ai 所指子树所有结点中的关键字均小于 Ki+1。

n 为结点中关键字的个数，满足 ceil(m/2)-1≤n≤m-1。

B+树：

1.有 k 个子树的中间节点包含有 k 个元素（B 树中是 k-1 个元素），每个元素不保存数据，只用来索引，所有数据

都保存在叶子节点。

2.所有的叶子结点中包含了全部元素的信息，及指向含这些元素记录的指针，且叶子结点本身依关键字的大小

自小而大顺序链接。

3.所有的中间节点元素都同时存在于子节点，在子节点元素中是最大（或最小）元素。

B 和 B+树的区别在于，B+树的非叶子结点只包含导航信息，不包含实际的值，所有的叶子结点和相连的节点使用链表相连，便于区间查找和遍历。

B+ 树的优点在于：

由于 B+树在内部节点上不包含数据信息，因此在内存页中能够存放更多的 key。 数据存放的更加紧密，具有更好的空间局部性。因此访问叶子节点上关联的数据也具有更好的缓存命中率。

B+树的叶子结点都是相链的，因此对整棵树的便利只需要一次线性遍历叶子结点即可。而且由于数据顺序排列并且相连，所以便于区间查找和搜索。而 B 树则需要进行每一层的递归遍历。相邻的元素可能在内存中不相邻，所以缓存命中性没有 B+树好。

但是 B 树也有优点，其优点在于，由于 B 树的每一个节点都包含 key 和 value，因此经常访问的元素可能离根节点更近，因此访问也更迅速。

LRU cache

布隆过滤器

跳表怎么实现的?

Skip List 构造步骤：

1、给定一个有序的链表；

2、选择连表中最大和最小的元素，然后从其他元素中按照一定算法（随机）随即选出一些元素，将这些元素组成有序链表。这个新的链表称为一层，原链表称为其下一层；

3、为刚选出的每个元素添加一个指针域，这个指针指向下一层中值同自己相等的元素。Top 指针指向该层首元素；

4、重复 2、3 步，直到不再能选择出除最大最小元素以外的元素。

哈夫曼编码是怎么回事？

依据字符出现概率来构造异字头的平均长度最短的码字，主要目的是根据使用频率来最大化节省字符（编码）的存储空间

非递归且不用额外空间（不用栈），如何遍历二叉树

算法

1、外部排序

待排序的文件很大，计算机内存不能容纳整个文件，这时候对文件就不能使用内部排序了。外部排序是指待排序的内容不能在内存中一下子完成，它需要做内外存的内容交换。

1、采用适当的内部排序方法对输入文件的每个片段进行排序，将排好序的片段（成为归并段）写到外部存储器中（通常由一个可用的磁盘作为临时缓冲区），这样临时缓冲区中的每个归并段的内容是有序的。

2、利用归并算法，归并第一阶段生成的归并段，直到只剩下一个归并段为止。

2、哪些排序是不稳定的，稳定意味着什么

排序稳定性：保证排序前两个相等的数其在序列的前后位置顺序和排序后它们两个的前后位置顺序相同。在简单形式化一下，如果 Ai = Aj，Ai 原来在位置前，排序后 Ai 还是要在 Aj 位置前。

3、不同数据集，各种排序最好或者最差的情况

4、如何优化算法

前端

Ajax 实现原理

ajax 的全称是 Asynchronous JavaScript and XML，其中，Asynchronous 是异步的意思，它有别于传统 web 开发中采用的同步的方式。

1.使用 CSS 和 XHTML 来表示。

2. 使用 DOM 模型来交互和动态显示。

3.使用 XMLHttpRequest 来和服务器进行异步通信。

4.使用 javascript 来绑定和调用。

实现 AJAX 异步调用和局部刷新的步骤:

1、创建 XMLHttpRequest 对象,也就是创建一个异步调用对象；

2、创建一个新的 HTTP 请求,并指定该 HTTP 请求的方法、URL 及验证信息;

3、设置响应 HTTP 请求状态变化的函数;

4、发送 HTTP 请求;

5、获取异步调用返回的数据;

6、使用 JavaScript 和 DOM 实现局部刷新。

计算机网络

1、TCP 与 UDP

TCP 和 UDP 是 OSI 模型中的运输层中的协议。TCP 提供可靠的通信传输,而 UDP 则常被用于广播和细节控制交给应用的通信传输。

TCP 建立连接要进行 3 次握手

1、主机 A 通过向主机 B 发送一个含有同步序列号的标志位的数据段给主机 B ,向主机 B 请求建立连接,通过这个数据段,主机 A 告诉主机 B 两件事:我想要和你通信;你可以用哪个序列号作为起始数据段来回应我

2、主机 B 收到主机 A 的请求后,用一个带有确认应答(ACK)和同步序列号(SYN)标志位的数据段响应主机 A,也告诉主机 A 两件事:我已经收到你的请求了,你可以传输数据了;你要用序列号作为起始数据段来回应我

3、主机 A 收到这个数据段后,再发送一个确认应答,确认已收到主机 B 的数据段:"我已收到回复,我现在要开始传输实际数据了

TCP 断开连接要进行 4 次

1、当主机 A 完成数据传输后,将控制位 FIN 置 1,提出停止 TCP 连接的请求

2、主机 B 收到 FIN 后对其作出响应,确认这一方向上的 TCP 连接将关闭,将 ACK 置 1

3、由 B 端再提出反方向的关闭请求,将 FIN 置 1

4、主机 A 对主机 B 的请求进行确认,将 ACK 置 1,双方向的关闭结束.

为什么连接的时候是三次握手，关闭的时候却是四次握手？

答：因为当 Server 端收到 Client 端的 SYN 连接请求报文后，可以直接发送 SYN+ACK 报文。其中 ACK 报文是用来应答的，SYN 报文是用来同步的。但是关闭连接时，当 Server 端收到 FIN 报文时，很可能并不会立即关闭 SOCKET，所以只能先回复一个 ACK 报文，告诉 Client 端，"你发的 FIN 报文我收到了"。只有等到我 Server 端所有的报文都发送完了，我才能发送 FIN 报文，因此不能一起发送。故需要四步握手。

为什么需要三次握手和四次挥手

为了防止已失效的连接请求报文段突然又传送到了服务端，因而产生错误

TCP 与 UDP 的区别：

1.基于连接与无连接，TCP 连接只能是点对点的，UDP 支持一对一，一对多，多对一和多对多的交互通信；

2.对系统资源的要求（TCP 较多，UDP 少）；

3.UDP 程序结构较简单，具有较好的实时性，效率较高；

4.流模式与数据报模式；

5.TCP 保证数据正确性，UDP 可能丢包，TCP 通过校验和，重传控制，序号标识，滑动窗口、确认应答实现可靠传输，丢包时的重发控制保证数据顺序，UDP 不保证。

2、TCP 滑窗

RTT 和 RTO

RTT：发送一个数据包到收到对应的 ACK 所花的时间；

RTO：重传时间间隔

保证 TCP 的可靠性

保证 TCP 的流控特性

3、HTTP 特点，HTTP 与 HTTPS 的区别

特点

支持客户/服务端模式

简单快速

灵活

无连接

无状态

Http （超文本传输协议）

Https 以安全为目标的 HTTP 通道

区别

1、HTTP 的 URL 以 HTTP:// 开头，而 HTTPS 的 URL 以 HTTPs:// 开头；

2、HTTP 是不安全的，而 HTTPS 是安全的，HTTPS 需要经过加密过程，过程相比于 HTTP 要繁琐一点，效率低一些；

3、HTTP 无需证书，而 HTTPS 必需要认证证书；

4、HTTPS 需要证书

4、浏览器中输入 url 地址到显示主页的过程,整个过程会使用哪些协议

过程	协议
1、浏览器查找域名的 IP 地址 （DNS 查找过程：浏览器缓存，路由器缓存，DNS 缓存）	DNS：获取域名对应 IP
2、浏览器向 web 服务器发送一个 HTTP 请求 （cookie 会随着请求发送给服务器）	TCP：与服务器建立 TCP 链接 IP：建立 TCP 协议时，需要发送数据，发送数据在网络层使用 IP 协议 OPSF：IP 数据包在路由器之间，路由选择使用 OPSF 协议 ARP：路由器与服务器通信时，需要将 IP 地址转换为 MAC 地址
3、服务器处理请求 （请求，处理请求&它的参数、cookie，生成一个 HTML 响应）	
4、服务器发回一个 HTML 响应	
5、浏览器开始显示 HTML	

	HTTP：TCP 建立后，使用 HTTP 协议访问网页

5、IP 地址与 MAC 地址的区别

IP 地址是指互联网协议地址（Internet Protocol Address）IP Address 的缩写。IP 地址是 IP 协议提供的一种统一的地址格式，它为互联网上的每一个网络和每一台主机分配一个逻辑地址，以此来屏蔽物理地址的差异。

MAC 地址又称为物理地址、硬件地址，用来定义网络设备的位置。网卡的物理地址通常是由网卡生产厂家写入网卡的，具有全球唯一性。MAC 地址用于在网络中唯一标示一个网卡，一台电脑会有一或多个网卡，每个网卡都需要有一个唯一的 MAC 地址。

6、HTTP 请求,响应报文格式

HTTP 请求报文主要由请求行、请求头部、请求正文 3 部分组成

HTTP 响应报文主要由状态行、响应头部、响应正文 3 部分组成

7、Http 有没有状态

Http 无状态

无状态是指，当浏览器发送请求给 server 的时候，server 响应，可是同一个浏览器再发送请求给 server 的时候，他会响应，可是他不知道你就是刚才那个浏览器，简单地说，就是 server 不会去记得你，所以是无状态协议。

如何解决 Http 无状态

URL 重写

URL 重写是截取传入 Web 请求并自动将请求重定向到其他 URL 的过程。比如浏览器发来请求 hostname/101.html，服务器自动将这个请求中定向为 http://hostname/list.aspx?id=101。

url 重写的优点在于：

1、缩短 url，隐藏实际路径提高安全性。

2、易于用户记忆和键入。

3、易于被搜索引擎收录。

隐藏表单域

Cookie

Cookie 实际上是一小段的文本信息。客户端请求服务器，如果服务器需要记录该用户状态，就使用 response 向客户端浏览器颁发一个 Cookie。客户端浏览器会把 Cookie 保存起来。当浏览器再请求该网站时，*浏览器把请求的网址连同该 Cookie 一同提交给服务器*。服务器检查该 Cookie，以此来辨认用户状态。服务器还可以根据需要修改 Cookie 的内容。

Session

程序需要为某个客户端的请求创建一个 session 的时候，服务器首先检查这个客户端的请求里是否已包含了一个 session 标识 - 称为 session id；

如果已包含一个 session id 则说明以前已经为此客户端创建过 session，服务器就按照 session id 把这个 session 检索出来使用（如果检索不到，可能会新建一个）；

如果客户端请求不包含 session id，则为此客户端创建一个 session 并且生成一个与此 session 相关联的 session id，session id 的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个 session id 将被在本次响应中通过 Cookie 返回给客户端保存。

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。

8、HTTPS 加密过程，是怎么保证信息可靠的？

HTTP 协议（HyperText Transfer Protocol，超文本传输协议）：是客户端浏览器或其他程序与 Web 服务器之间的应用层通信协议 。

HTTPS 协议（HyperText Transfer Protocol over Secure Socket Layer）：可以理解为 HTTP+SSL/TLS， 即 HTTP 下加入 SSL 层，HTTPS 的安全基础是 SSL，因此加密的详细内容就需要 SSL，用于安全的 HTTP 数据传输。

一些问题

1、Tomcat 类加载器之为何违背双亲委派模型

异步写系统

一致性 Hash 算法，它解决了什么问题？

一致性 Hash 算法将整个哈希值空间组织成一个虚拟的圆环；

整个空间按顺时针方向组织，圆环的正上方的点代表 0，0 点右侧的第一个点代表 1，以此类推，2、3、4、5、6……直到 2^32-1，也就是说 0 点左侧的第一个点代表 2^32-1， 0 和 2^32-1 在零点中方向重合，我们把这个由 2^32 个点组成的圆环称为 Hash 环。

下一步将各个服务器使用 Hash 进行一个哈希，具体可以选择服务器的 IP 或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置；

接下来使用如下算法定位数据访问到相应服务器：将数据 key 使用相同的函数 Hash 计算出哈希值，并确定此数据在环上的位置，从此位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器！

一致性哈希算法解决了普通余数 Hash 算法伸缩性差的问题，可以保证在上线、下线服务器的情况下尽量有多的请求命中原来路由到的服务器。

6 大设计原则，什么是高内聚低耦合

单一职责原则(Single Responsibility Principle, SRP)：一个类只负责一个功能领域中的相应职责，或者可以定义为：就一个类而言，应该只有一个引起它变化的原因。

开闭原则(Open-Closed Principle, OCP)：一个软件实体应当对扩展开放，对修改关闭。即软件实体应尽量在不修改原有代码的情况下进行扩展。

里氏代换原则(Liskov Substitution Principle, LSP)：所有引用基类（父类）的地方必须能透明地使用其子类的对象。

依赖倒转原则(Dependency Inversion Principle, DIP)：抽象不应该依赖于细节，细节应当依赖于抽象。换言之，要针对接口编程，而不是针对实现编程。

接口隔离原则(Interface Segregation Principle, ISP)：使用多个专门的接口，而不使用单一的总接口，即客户端不应该依赖那些它不需要的接口。

迪米特法则(Law of Demeter, LoD)：一个软件实体应当尽可能少地与其他实体发生相互作用。

Netty

Netty 封装了 JDK 的 NIO，是一个异步事件驱动的网络应用框架，用于快速开发可维护的高性能服务器和客户端

JDK 原生也有一套网络应用程序 API，但是存在一系列问题，主要如下：

NIO 的类库和 API 繁杂，使用麻烦。你需要熟练掌握 Selector、ServerSocketChannel、SocketChannel、ByteBuffer 等。

需要具备其他的额外技能做铺垫。例如熟悉 Java 多线程编程，因为 NIO 编程涉及到 Reactor 模式，你必须对多线程和网路编程非常熟悉，才能编写出高质量的 NIO 程序。

可靠性能力补齐，开发工作量和难度都非常大。例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处理等等。 NIO 编程的特点是功能开发相对容易，但是可靠性能力补齐工作量和难度都非常大。

JDK NIO 的 Bug。例如臭名昭著的 Epoll Bug，它会导致 Selector 空轮询，最终导致 CPU 100%。

Netty 对 JDK 自带的 NIO 的 API 进行封装，解决上述问题，主要特点有：

设计优雅，适用于各种传输类型的统一 API 阻塞和非阻塞 Socket；基于灵活且可扩展的事件模型，可以清晰地分离关注点；高度可定制的线程模型 - 单线程，一个或多个线程池；真正的无连接数据报套接字支持（自 3.1 起）。

使用方便，详细记录的 Javadoc，用户指南和示例；没有其他依赖项，JDK 5（Netty 3.x）或 6（Netty 4.x）就足够了。

高性能，吞吐量更高，延迟更低；减少资源消耗；最小化不必要的内存复制。

安全，完整的 SSL/TLS 和 StartTLS 支持。

社区活跃，不断更新，社区活跃，版本迭代周期短，发现的 Bug 可以被及时修复，同时，更多的新功能会被加入。

应用场景：

在分布式系统中，各个节点之间需要远程服务调用，高性能的 RPC 框架必不可少， Netty 作为异步高性能的通信框架，往往作为基础通信组件被这些 RPC 框架使用；

游戏行业。无论是手游服务端还是大型的网络游戏，Java 语言得到了越来越广泛的应用。Netty 作为高性能的基础通信组件，它本身提供了 TCP/UDP 和 HTTP 协议栈。 非常方便定制和开发私有协议栈，账号登录服务器，地图服务器之间可以方便的通过 Netty 进行高性能的通信；

大数据领域。经典的 Hadoop 的高性能通信和序列化组件 Avro 的 RPC 框架，默认采用 Netty 进行跨界点通信，它的 Netty Service 基于 Netty 框架二次封装实现。

Netty 零拷贝

即所谓的 Zero-copy, 就是在操作数据时, 不需要将数据 buffer 从一个内存区域拷贝到另一个内存区域. 因为少了一次内存的拷贝, 因此 CPU 的效率就得到的提升。

Netty 提供了 CompositeByteBuf 类, 它可以将多个 ByteBuf 合并为一个逻辑上的 ByteBuf, 避免了各个 ByteBuf 之间的拷贝.

通过 wrap 操作, 我们可以将 byte[] 数组、ByteBuf、ByteBuffer 等包装成一个 Netty ByteBuf 对象, 进而避免了拷贝操作.

ByteBuf 支持 slice 操作, 因此可以将 ByteBuf 分解为多个共享同一个存储区域的 ByteBuf, 避免了内存的拷贝.

通过 FileRegion 包装的 FileChannel.tranferTo 实现文件传输, 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.

Netty 粘包与拆包

单点登录实现的过程

反向代理

Nginx 的基本配置，如何通过 lua 语言设置规则，如何设置 session 粘滞，

Nginx 底层

协议，集群设置，失效转移

分布式架构相关

1、为什么用微服务

2、为什么 zookeeper 能作为注册中心

zookeeper 是个分布式文件系统,当你注册服务就是创建了一个 znode 节点，节点存储了该服务的 IP、端口、调用方式。

第一次调用服务，定位服务位置，缓存到本地。再次调用通过负载均衡算法从 IP 列表中取一个服务提供者的服务器调用服务。

当服务器下线，这个服务也就下线了。再次上线就会缓存新服务 ip 等信息

3、使用分布式碰到的 bug

4、zookeeper 有集群吗？怎么实现的

5、zookeeper 宕机还能访问吗；服务失效踢出 zookeeper 中临时节点的原理

Zookeeper 本身也是集群，推荐配置不少于 3 个服务器（至少 2*N+1）。Zookeeper 自身也要保证当一个节点宕机时，其他节点会继续提供服务。

2、如果是一个 Follower 宕机，还有 2 台服务器提供访问，因为 Zookeeper 上的数据是有多个副本的，数据并不会丢失；

3、如果是一个 Leader 宕机，Zookeeper 会选举出新的 Leader。

1、dubbo

dubbo 原理，为什么要用 dubbo

Dubbo 是一个分布式服务框架，致力于提供高性能和透明化的远程服务调用方案，这容易和负载均衡弄混，负载均衡是对外提供一个公共地址，请求过来时通过轮询、随机等，路由到不同 server。目的分摊压力；

我们通过 dubbo 建立 service 这个服务，并且到 zookeeper 上面注册，填写对应的 zookeeper 服务所在 的 IP 及端口号。

1.服务接口层：该层与实际业务逻辑有关，根据服务消费方和服务提供方的业务设计，实现对应的接口。

2.配置层：对外配置接口，以 ServiceConfig 和 ReferenceConfig 为中心，可以直接 new 配置类，也可以根据 spring 解析配置生成配置类。

3.服务代理层：服务接口通明代理，生成服务的客户端 Stub 和服务端 Skeleton，以 ServiceProxy 为中心，扩展接口 ProxyFactory。

4.服务注册层：封装服务地址的注册和发现，以服务 URL 为中心，扩展接口为 RegistryFactory、Registry、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：封装多个提供者的路由和负载均衡，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，实现对服务消费通明。只需要与一个服务提供方进行交互。

6.监控层：RPC 调用时间和次数监控，以 Statistics 为中心，扩展接口 MonitorFactory、Monitor 和 MonitorService。

7.远程调用层：封装 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，其他模型都是向它靠拢，或转换成它，它代表一个可执行体，可向它发起 Invoker 调用，它有可能是一个本地实现，也有可能是一个远程实现，也有可能是一个集群实现。

8.信息交换层：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 和 ExchangeChannel，ExchangeClient 和 ExchangeServer。

9.网络传输层：抽象和 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：可复用的一些工具，扩展接口为 Serialization、ObjectInput，ObejctOutput 和 ThreadPool。

Dubbo 的远程调用怎么实现的

读取配置、拼装 url、创建 Invoker、服务导出、服务注册以及消费者通过动态代理、filter、获取 Invoker 列表、负载均衡

动态代理

dubbo 集群负载均衡策略

Random LoadBalance 随机，按权重设置随机概率。

RoundRobin LoadBalance 轮询，按公约后的权重设置轮询比率。

LeastActive LoadBalance 最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。

ConsistentHash LoadBalance 一致性 Hash，相同参数的请求总是发到同一提供者。

SPI 机制

系统设计的各个抽象，往往有很多不同的实现方案，在面向的对象的设计里，一般推荐模块之间基于接口编程，模块之间不对实现类进行硬编码。一旦代码里涉及具体的实现类，就违反了可拔插的原则，如果需要替换一种实现，

就需要修改代码。为了实现在模块装配的时候能不在程序里动态指明，这就需要一种服务发现机制。

Java SPI 就是提供这样的一个机制：为某个接口寻找服务实现的机制。有点类似 IOC 的思想，就是将装配的控制权移到程序之外，在模块化设计中这个机制尤其重要。所以 SPI 的核心思想就是解耦。

2、Dubbo 有哪些传输协议？

1、dubbo 协议，是默认的基于 TCP 传输协议的长连接，NIO 异步通信，适合于小数据量高并发的场景以及服务消费者机器数远大于服务提供者机器数的情况。

2、RMI 协议

RMI 协议采用 JDK 标准的 java.rmi.*实现，采用阻塞式短连接和 JDK 标准序列化方式

Java 标准的远程调用协议。

3、Hession 协议

Hessian 协议用于集成 Hessian 的服务，Hessian 底层采用 Http 通讯，采用 Servlet 暴露服务，Dubbo 缺省内嵌 Jetty 作为服务器实现

4、Http 协议

采用 Spring 的 HttpInvoker 实现

基于 http 表单的远程调用协议。

5、webservice

基于 CXF 的 frontend-simple 和 transports-http 实现

基于 WebService 的远程调用协议。

6、thrif

Thrift 是 Facebook 捐给 Apache 的一个 RPC 框架，当前 dubbo 支持的 thrift 协议是对 thrift 原生协议的扩展，在原生协议的基础上添加了一些额外的头信息，比如 service name，magic number 等。Thrift 不支持 null 值，不能在协议中传 null

7、redis 协议

8、memcached 协议

3、分布式锁

数据库实现

缓存实现

zookeeper 实现

4、死锁，分布式架构出现死锁如何解决

当且仅当以下四个条件同时成立时，死锁才会发生：

1) 互斥。同一个资源在同一时刻最多只能被一个进程占用。

2) 占有并等待。必然有一个进程至少占用了系统中的一个资源，同时在等待获取被其他进程占用的资源。

3) 不可剥夺。一个进程不能剥夺被其他进程占用的资源。

4) 循环等待。在等待图中有一个循环。

死锁预防 使引起死锁的必要条件不成立

- 资源排序，按资源序列申请
- 将所有并发事务排序，按标识符或者开始时间
- 有死锁危险的时候，事务退出已经占有的资源， 有两种方法：

等待-死亡： 重启较为年轻的事务，较为年老的事务等待已经持有资源但是较为年轻的事务

受伤-等待： 年轻的等待年老的，较年轻的重启，而重启事务并不一定是目前正在申请的事务 （有疑问）

死锁检测

- 检测死锁时循环等待的圈

5、分布式会话

保证 session 一致性的架构设计常见方法：

1、session 同步法：多台 web-server 相互同步数据

2、客户端存储法：一个用户只存储自己的数据

3、反向代理 hash 一致性：四层 hash 和七层 hash 都可以做，保证一个用户的请求落在一台 web-server 上

4、后端统一存储：web-server 重启和扩容，session 也不会丢失