**1. Java 篇**

**(1). Java基础知识**

* java中==和equals和hashCode的区别
  + ==：对于8种基本的数据类型short(2字节),char(2),int(4),float(4),byte（1）,double(8),long(8),Boolean,==运算符比较的是它们的值是否相等。对于引用类型（类、接口和数组），==比较的则是它们在内存中所存放的地址是否相等（就是判断是否为同一个对象）。**对象是放在堆中的，栈中存放的是对象的引用（地址）。由此可见'=='是对栈中的值进行比较的**。如果要比较堆中对象的内容是否相同，那么就要重写equals方法了。
  + equals()：Object的equals方法与==的功能相同，但是该方法被很多类重写了，如String类的equals方法功能是判断字符串的内容是否相等。
  + hashCode()：根据对象实例的内容，计算出一个int类型，用于在集合（如HashMap）中分配元素位置。如果两个对象equals，则其hashCode必定相等，反之则不然。
* int与Integer的区别
  + **Integer是int的包装类，int则是java的一种基本数据类型**
  + Integer变量必须实例化后才能使用，而int变量不需要
  + Integer实际是对象的引用，当new一个Integer时，实际上是生成一个指针指向此对象；而int则是直接存储数据值
  + Integer的默认值是null，int的默认值是0
  + 通过new生成的Integer永远是不相等的，但是形如Integer a = 10这样的Integer对象，如果值在区间-128到127之间，则==的结果是true，否则为false，超出这个范围时，判断的时数值加地址。
* 抽象类的意义、接口和抽象类的区别
  + **抽象类**：除了你不能实例化抽象类之外，它和普通Java类没有任何区别，即它只能作为子类的超类，其作用是用来创建继承层级里子类的模板，如提供一些方法的默认实现（也可以像接口一样存在一些抽象方法让子类自己去实现）。
  + 一个类只能继承一个抽象类，但是可以实现很多接口。因此接口用于解决多重继承问题。
  + 抽象类当中可以存在非抽象的方法，可接口不能，且它里面的方法只是一个声明必须用public来修饰没有具体实现的方法
  + 抽象类中的成员变量可以被不同的修饰符来修饰，可接口中的成员变量默认的都是静态常量（static final）。
  + **JDK8中**：接口的新特性：可以在接口中编写**静态方法**（只能通过接口名调用，不可以通过实现类的类名或者实现类的对象调用）**和default方法**（只能通过接口实现类的对象来调用）。JDK8中强调的区别是**数据成员和方法的可访问性**:抽象类允许非static和非final字段，允许方法是public、private或protected，而接口的字段类型本质上是public的、static的和final的，所有接口方法本质上都是公共的。
* 能否创建一个包含可变对象的不可变对象?
  + 不可变对象(Immutable Objects)即对象一旦被创建它的状态（对象的数据，也即对象属性值）就不能改变，任何对它的改变都应该产生一个新的对象。不可变对象的类即为不可变类。JAVA平台类库中包含许多不可变类，如**String、基本类型的包装类、BigInteger和BigDecimal**等。
  + 我们是**可以**创建一个包含可变对象的不可变对象的，你只需要谨慎一点，不要共享可变对象的引用就可以了，如果需要变化时，就返回原对象的一个拷贝。最常见的例子就是对象中包含一个日期对象的引用。
* 谈谈对java多态的理解： [对多态的理解](https://www.cnblogs.com/liujinhong/p/6003144.html)
  + 当超类对象引用变量引用子类对象（Father father = new Son();）时，被**引用对象的类型(即Son)**而不是引用变量的类型(Father)决定了调用谁的成员方法，但是这个被调用的方法**必须是在超类中定义过的**，也就是说被子类覆盖的方法 （但是如果强制把超类转换成子类的话，就可以调用子类中新添加而超类没有的方法了）。优先级：实际上这里涉及方法调用的优先问题 ，优先级由高到低依次为：**this.show(O)、super.show(O)、this.show((super)O)、super.show((super)O)**
  + 实现多态的技术称为：**动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法**。
  + 多态的作用：**消除类型之间的耦合关系**。
* String、StringBuffer、StringBuilder区别
  + String是不可变类，因此当对字符串内容进行合成操作时JVM会自动new一个StringBuilder对象进行操作，操作完成后再转成String。如果循环体中不主动使用StringBuilder对象而是直接string1 = string1 + string2，会在每次循环中自动生成一个StringBuilder对象，会造成内存的浪费。而StringBuffer比StringBuilder多了一个线程安全特性。
* 泛型中extends和super的区别
  + <? extends T>和<? super T>分别是上界通配符和下界通配符，用于对泛型类或集合所涉及的类型加以限制。
  + 通配符的副作用： 使用<? extends T>的泛型集合，取出的时候将会是T类型的元素，因此可能会丢失子类特有的信息。使用<? super T>的泛型集合，取出的时候必须为**Object**，因为T的粒度是最大的，无法将大粒度转化为小粒度。
* final，finally，finalize的区别
  + final：**修饰类则表示该类不可以被继承；修饰方法则该类的子类无法对该方法进行重写；修饰变量则该变量的值不可以被修改**。
  + finally：用于异常处理try-catch-finally。注意，如果try，catch和finally中都有return语句，则finally中的return会屏蔽掉其他块中的return，而且此时会吞掉异常情况，因此要避免在finally中使用return。
  + finalize:每个对象都会有该方法，该方法在gc启动、对象被回收的时候被调用，**可以帮助对象不被回收，**一般不需要重写。
* 序列化的方式 java.io.ObjectOutputStream以及ObjectInputStream
  + 把对象转换为字节序列的过程称为对象的序列化。在很多应用中，需要对某些对象进行序列化，让它们离开内存空间，入住物理硬盘，以便长期保存。比如最常见的是Web服务器中的Session对象，当有 10万用户并发访问，就有可能出现10万个Session对象，内存可能吃不消，于是Web容器就会把一些seesion先序列化到硬盘中，等要用了，再把保存在硬盘中的对象还原到内存中；当两个进程在进行远程通信时，彼此可以发送各种类型的数据。无论是何种类型的数据，都会以二进制序列的形式在网络上传送。发送方需要把这个Java对象转换为字节序列，才能在网络上传送；接收方则需要把字节序列再恢复为Java对象。
* String 转换成 Integer的方式及原理
  + Integer.parseInt(String s)
  + parseInt(String s)--内部调用parseInt(s,10)（默认为10进制）
  + 正常判断null，进制范围，length等
  + 判断第一个字符是否是符号位
  + 循环遍历确定每个字符的十进制值
  + 通过\*= 和-= 进行计算拼接
  + 判断是否为负值 返回结果。
* 静态属性和静态方法是否可以被继承？是否可以被重写？以及原因？
  + 可以被继承，但是不会被重写（overwrite），而是被隐藏。
  + 重写overwrite：重写方法后，子类中的方法的优先级要高于超类中的该方法。而隐藏是没有这个优先级之分的。
  + 由于多态是靠重载、**继承、重写实现的（主要是后两者），所以静态属性和静态方法不能实现多态。**
* 成员内部类、静态内部类、局部内部类和匿名内部类的理解，以及项目中的应用
  + 使用内部类的好处：一是可以实现多重继承；二是可以很好地实现隐藏——一般的非内部类是不允许有protected和private访问权限的，但是内部类却可以访问。
  + 成员内部类：Outer.Inner inner = outer.new Inner();成员内部类持有该外部类的引用（即可以通过this引用外部类的成员变量）；成员内部类中不可以定义static变量和方法。
  + 静态内部类：不会持有外部类的引用，可以访问外部类的静态成员变量，但是只能通过外部类的实例来访问外部类的非静态成员变量。静态内部类与外部类仅在逻辑上有从属关系，前者在逻辑上是可以独立存在的。
  + 局部内部类：在外部类的方法中进行定义，其作用范围仅限于该方法中，且局部内部类不能使用private,protected,public修饰符，不能有静态成员。
  + 匿名内部类：典型例子是Thread类和Runnable接口。只要一个类是抽象的或是一个接口，那么其子类中的方法都可以使用匿名内部类来实现。
* Java的异常体系
  + [Java异常体系简析](http://www.cnblogs.com/yiwangzhibujian/p/7141390.html)
* 什么是异常链
  + 当异常中抛出异常时，防止先前的异常信息丢失就用到异常链。
  + A异常抛出B异常前，通过B异常对象调用initCause方法传入A异常对象，再抛出B异常。接收B异常的catch通过调用getCause（）能获取A异常信息（即引发B异常的异常）。这样A异常在上抛过程就不会丢失。
* throw和throws的区别
  + throw()方法用于抛出异常，throw是在定义方法时表示该方法可能会抛出某些异常。
  + 一个方法**throws声明抛出异常后，那么调用该方法的时候必须处理声明的所有异常**，可以使用try，catch逐一捕获或者继续使用throws进行声明。
* 反射的原理，反射创建类实例的三种方式 [反射相关-这一篇全了解](https://blog.csdn.net/w372426096/article/details/84616662)
  + 反射机制指的是**程序在运行时能够获取自身的信息**。在java中，只要给定类的名字，那么就可以通过反射机制来获得类的所有属性和方法。
  + 使用场景：**维护方便，使用类名（字符串）就可以获得这个类的实例，扩展性高，扩展一些插件等，开始时还不知道类名。**
  + 获取Class类的第一种方法：调用该类对象的getClass()方法
  + 第二种方法：调用Class类的静态方法forName()
  + 第三种方法：调用该类的.class属性，如Random.class
* Java当中的四种引用

<https://blog.csdn.net/u014532217/article/details/79184412>

* + 强引用：创建一个对象并把这个对象赋给一个引用变量。当有引用变量指向该强引用时，**对象永远不会被GC回收，JVM宁愿抛出OOM错误也不会回收这种对象。一般程序中我们创建的对象都是强引用。**
  + 软引用：具有软引用的对象，当**内存空间足够时不会被回收**，内存空间不足时会被回收。可用来实现内存敏感的高速缓存如**网页缓存、图片缓存**等。
  + 弱引用：用于描述非必须对象。当JVM进行gc时，无论内存是否充足都会回收被弱引用关联的对象。Java源码中的**java.util.WeakHashMap中的key就是使用弱引用**，我的理解就是，一旦我不需要某个引用，JVM会自动帮我处理它，这样我就不需要做其它操作。本身WeakHashMap还在使用但是它内部的某个key已经不再使用，就可以将key回收。
  + 虚引用：与软引用、弱引用不同，它不影响对象的生命周期。当垃圾回收器准备回收一个对象时，如果发现它有虚引用，就会把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否加入了该虚引用，来了解被引用的对象是否将要被垃圾回收，从而可以在该对象被回收前采取必要的措施。
* 深拷贝和浅拷贝的区别是什么?
  + 深拷贝和浅拷贝不是Java中特有的方法，主要取决于对实现cloneable接口的类中定义的clone方法的实现。
  + 浅拷贝：能够复制变量，但是如果对象A内还有对象B，则只能复制该内部对象B的地址，也就是说，复制出的新对象A\_new中包含的对象与A中包含的对象是同一个，B，因此在A\_new中对B可以进行改动。
  + 深拷贝，不仅可以复制变量还可以复制原对象中的对象，即A\_new和B\_new，这样在新对象中对其内部对象进行操作不会影响到原来的对象及其内部对象。因此要深拷贝，那么被拷贝的对象的内部对象也必须实现cloneable接口。
* 什么是编译期常量？使用它有什么风险？ <https://blog.csdn.net/qq_32575047/article/details/78846264?utm_source=blogxgwz4>
  + 编译期常量就是所谓的public final static 常量。 由于在编译时就确定了值，在使用的场合会直接写成值。而不是直接到原来的类中读取。这样会有一个问题。 如果类A提供了常量类B使用了常量，并都进行了编译。然后，修改了类A的源码，调用系统进行编译。系统发现类A是新的代码，编译了，类B是旧的代码，就不进行编译，使用旧的类。所以导致类A的修改无法反映到类B中。这样**造成了读取变量的值不同的风险**。
* 你对String对象的intern()熟悉么? 先看完下面的再看这篇博客： [intern()——结合JVM内存区域看](https://blog.csdn.net/wjzhang5514/article/details/70209403)
  + 简而言之，通过intern()生成的字符串的位置在**~~方法区~~（JDK8以后改为）的字符串常量池**，而不是存放对象的堆。
  + 我们知道，一个Java程序运行后，String类会在内存的**~~方法区~~（JDK8中字符串常量池不再在方法区而是在Java堆中）**中维护一个字符串常量池。但是对一个字符串调用intern()方法后，会先检查池内是否有该字符串，若有则返回池中该字符串；若没有没有则先创建再返回，确保返回的字符串已经以字面量的形式存在于池中。
  + 因此如果使用new String()方法生成的字符串，其在内存中**是处于堆中**的，对该字符串调用intern()方法返回的新字符串的位置却是在**字符串常量池**，因此二者通过==运算符的结果将为false；如果是使用String abc = “abc”+”bcd”，那么abc==abc.intern为true，因为这种方法生成的字符串abc它本身就存放在字符串常量池。需要注意，形如String abc = str1 + str2 （str1、str2都是字符串对象），这样生成的abc是存放在堆中的，因为它等同于String abc = new String(str1 + str2)，而不是直接字面量的拼接，因此此时intern生成的字符串与原字符串的地址也是不同的。
* String为什么要设计成不可变的？
  + 第一是**字符串常量池的需要**。如果两个字符串都引用字符串常量池中的字符串，如果String是可变的话，那么改动一个字符串，将会影响到另一个，因此必须为不可变。
  + 第二是**String对象的HashCode**被频繁使用，字符串不变性保证了哈希码的唯一性，意味着不必每次都去计算新的哈希码。
  + 第三是**安全性的考虑**，String被许多的Java类（库）用来当做参数，例如URL、文件路径path以及反射机制所需要的String参数等，保证String不可变可以避免很多安全隐患。
* a=a+b与a+=b有什么区别吗?
  + 当ab是相同类型时没有区别。
  + 当a比b的类型范围小的时候（如a int，b float），a=a+b无法编译通过，必须类型强制转换——a=(int)(a+b).而a+=b在编译的时候编译器自动加上了强制类型转换，会丢失精度。
* 静态代理和动态代理的区别，什么场景使用？ [代理](https://blog.csdn.net/ikownyou/article/details/53081426) 
  + 区别：**代理类的生成时间不同**。
  + 代理的书面概念：为某个对象提供一个代理，以控制对这个对象的访问。 代理类和委托类有共同的父类或父接口，这样在任何使用委托类对象的地方都可以用代理对象替代。代理类负责请求的预处理、过滤、将请求分派给委托类处理、以及委托类执行完请求后的后续处理。
  + 静态代理：对于实现了某接口的目标对象，如果想要在调用该对象的某方法前后做一些别的事情（比如记录该对象的该方法的耗时），但是不想对该对象的类文件进行改动，就可以写一个静态代理类，通过代理对象去调用目标对象中的方法，此时就可以在代理对象中写我们希望的一些操作。  
    如：
* package proxy;  
    
  public class UserDaoProxy implements IUserDao {  
   private IUserDao **target**;  
   public UserDaoProxy(IUserDao *target*) {  
   this.**target** = *target*;  
   }  
    
   @Override  
   public void save() {  
   System.***out***.println("开始事务...");  
   **target**.save();   
   System.***out***.println("提交事务...");  
   }  
  }
  + 静态代理的缺点：因为代理对象需要与目标对象实现一样的接口,所以会有很多代理类,类太多.同时,一旦接口增加方法,目标对象与代理对象都要维护.
  + 动态代理：是利用Java的反射机制，动态地在内存中构建代理对象(需要我们指定创建代理对象/目标对象实现的接口的类型)，使用java.lang.reflect.Proxy类。动态代理的代理对象**不需要实现接口**。

**App.java**

//目标对象  
IUserDao *target* = new UserDao();  
//给目标对象创建代理对象,需要将Object对象进行强转  
IUserDao *proxy* = (IUserDao) new ProxyFactory(*target*).getProxyInstance();

**ProxyFactory.getProxyInstance():**

public Object getProxyInstance() {  
 return Proxy.*newProxyInstance*(  
 **target**.getClass().getClassLoader(),  
 **target**.getClass().getInterfaces(),  
 new InvocationHandler() {  
 @Override  
 public Object invoke(Object *proxy*, Method *method*, Object[] *args*) throws Throwable {  
 System.***out***.println("开始事务2");  
 //执行目标对象方法  
 Object *returnValue* = *method*.invoke(**target**, *args*);  
 System.***out***.println("提交事务2");  
 return *returnValue*;  
 }  
 }  
 );  
}

* Java中实现多态的机制是什么？
  + 重载，**继承，重写**（主要是后二者）。
* 说一下泛型原理，并举例说明 [泛型——csdn](https://blog.csdn.net/MustangJy/article/details/86634807)
  + 泛型程序设计的作用就是让编写的代码可以被不同类型的对象所重用。
  + 类型擦除：类型擦除之所以产生，是因为对于虚拟机来讲不存在泛型类型对象，而只有普通的类和方法。虚拟机处理时要把泛型类型处理为对应的原始类型，这个过程就叫做类型擦除。
* Object类的equal和hashCode方法重写，为什么？   
  <https://blog.csdn.net/qq352773277/article/details/41675407>
  + 为什么要重写equals():因为Object中默认的equals方法是对两个对象引用的比较，与==相同，比较的是对象的地址；一般我们需要通过对象内容比较是否相等时，就要重写equals()方法。
  + 为什么同时要重写hashCode()：要结合HashTable、HashMap等哈希集合来分析，hashCode()用来计算元素在集合中的位置；所以，**如果两个对象是equals()的，那么这两个对象调用hashCode()必须产生同样的结果**，因此要同步重写hashCode()。
* **IO和NIO区别，NIO的具体实现**<https://blog.csdn.net/suifeng3051/article/details/48160753>  
  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。因此NIO适用于同时打开很多连接但是每个连接每次只是发送少量数据的场景，比如聊天服务器。  
  IO适用于少量连接使用大量带宽的情境，一个线程处理一个连接。
  + **IO是面向流的，NIO是面向缓冲区。**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。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。 Java NIO的缓冲导向方法略有不同。数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性。但是，还需要检查是否该缓冲区中包含所有需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。
  + **IO的流是阻塞的，而NIO有同步非阻塞模式和异步阻塞模式。**一个线程调用IO的read和write方法时，该线程被阻塞，直到读取到数据或者数据完全写入，在此期间线程不能做别的事。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现在可以管理多个输入和输出通道（channel）。
  + **NIO具有选择器，允许一个单独的线程监视多个通道。**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、在NIO中有几个比较关键的概念：Channel（通道），Buffer（缓冲区），Selector（选择器）。

2、Channel类似于同传统IO中的Stream，Channel是双向的，stream是单向的分为输入和输出流，对channel的读写必须是buffer对象。

* FileChannel：从文件读取数据的
* DatagramChannel：读写UDP网络协议数据
* SocketChannel：读写TCP网络协议数据
* ServerSocketChannel：可以监听TCP连接

3、一个Buffer实质上是一个容器对象，发给Channel的所有对象都必须先放到Buffer中；同样的，从Channel中读取的任何数据都要读到Buffer中。

使用 Buffer 读写数据一般遵循以下四个步骤：

1. 写入数据到 Buffer；
2. 调用 flip() 方法，将buffer从写模式切换到读模式
3. 从 Buffer 中读取数据；
4. 调用 clear() 方法或者 compact() 方法。

4、NIO是属于**同步IO，可以选择非阻塞方式**，没有阻塞进行读写，NIO通过Selector注册特定IO事件，可以监听任何数量的通道上的事件，不用轮询（阻塞直到可以进行read或wirte操作），也不用额外的线程

**JVM**

<https://blog.csdn.net/dd864140130/article/details/49515403>

Java**程序运行启动底层的流程**

首先编**译器编译出class文件**，即字节码文件，接着通过**类加载器完成类的加载**过程，如果使用双亲委派过程的话，自底向上检查类是否完成加载，如果没有加载自上而下尝试加载类，完成类的加载之后进行**类的执行**，线程创建后，都会产生程序计数器（PC）和栈（Stack），程序计数器存放下一条要执行的指令在方法内的偏移量，栈中存放一个个栈帧，**每个栈帧对应着每个方法的每次调用**，栈帧是用于支持虚拟机进行方法调用和方法执行的数据结构。每个栈帧都包括了一下几部分：**局部变量表、操作数栈、动态连接、方法的返回地址（返回地址是存放调用调用该方法的pc计数器的值（正常返回））** 和一些额外的附加信息，从Main方法开始执行，方法的执行对应上面的指令的顺序读取执行，栈中的帧栈的出入栈

* 谈谈你对解析与分派的认识。
* 内存模型及分区

<https://www.cnblogs.com/itar/p/7424311.html>

JVM 分为堆区和栈区，还有方法区，初始化的对象放在**堆**里面，引用放在栈里面，class类信息常量池（static常量和static变量）等放在**方法区**

**方法区**：主要是存储类信息，**常量池（static常量和static变量）还有符号引用**，编译后的代码（字节码）等数据，运行时候也可以将新的常量放入池中，如String.intern()方法，方法区的垃圾回收主要回收常量池和对类型的卸载。

**堆**：初始化的对象，成员变量 （那种非static的变量），所有的对象实例和数组都要在堆上分配，堆可以分为**新生代和老年代**，新生代中又可以分为Edon和FromSurvivor和ToSurvivor区，堆只要求是逻辑上连续的即可，其大小可以通过-Xmx和-Xms来扩展，当堆中没有内存完成实例分配且**无法扩展时候**，会抛出OutOfMenooryError异常。

**栈**：栈的结构是栈帧组成的，调用一个方法就压入一帧，帧上面存储局部变量表，操作数栈，方法出口等信息，局部变量表存放的是8大基础类型加上一个应用类型，所以还是一个指向地址的指针

* 对象创建的过程和访问定位

虚拟机遇到new指令时，首先检查这个指令的参数是否在常量池中能定位导到一个符号引用，并检查其是否被加载、解析和初始化过，如果没有则先执行相应的类加载过程。接下来为对象分配内存，两种方式：**指针碰撞**（堆中内存空闲内存与不可用内存界限明显），**空闲列表（**维护一个内存记录表，对应内存交错**），分配内存时保证线程安全方法：采用CAS**配上失败重试的方式保证更新操作的原子性**；**把内存分配的动作按照线程划分在不同的空间中执行，**即预先在堆中分配一块内存，为本地线程分配缓冲（TLAB），当TLAB不够时候，再同步锁定。**

访问定位两种方式：reference指向维护句柄池，句柄池在堆中，其中包含到对象实例和对象类型数据的指针；reference直接指向对象实例地址（HotSpot虚拟机使用直接指针访问）。

* 你知道哪些或者你们线上使⽤什么**GC策略**？它有什么优势，适⽤于什么场景？

**老年代**中对象存活率高，没有额外空间进行担保，采用**标记-整理，标记-清除**

**新生代**中对象存活率低，**复制**算法，付出少量复制成本完成回收

**标记清除**：会产生不连续的空间碎片，如果没有足够连续的会触发新GC

**标记整理**：标记完毕之后，让所有存活的对象向一端移动，没有碎片

**复制算法**：分为8：1的Eden区和survivor区，当存活对象较多时，进行分配担保这些对象直接进入老年代

* GC收集器有哪些？CMS收集器与G1收集器的特点。

串行收集器：串行收集器使用单独的线程进行收集，GC时服务有停顿时间

并行收集器：次要回收中使用多线程来执行

CMS（concurrent Mark sweep）收集器是基于**“标记—清除”**算法实现的，经过多次标记才会被清除，以**获取最短停顿时间**为目标，触发FullGC之前可以进行碎片整理，**无法处理浮动垃圾**，在CMS过程中用户线程产生的垃圾只能留到下一次GC时候回收，因此要预留空间，而不是等到几乎填满了才启动

G1从**整体来看是基于“标记—整理”**算法实现的收集器，从**局部（两个Region之间）上来看是基于“复制”**算法实现的，实现最大吞吐量，G1中保留了新生代和老年代的概念，但是内部不再是物理隔离的。

* SafePoint和saferegion

GC时候必须等到java线程全部进入safepoint，完成根节点枚举才可以

Saferegion则是一段安全的代码区域，无论线程状态，可能是sleep或blocked，没有分配cpu时间，保证安全GC

* 在JVM中，如何判断一个对象是否死亡（可达性分析）？

引用计数法：指的是如果某个地方引用了这个对象就+1，如果失效了就-1，当为0就会回收但是JVM没有用这种方式，因为无法判定相互循环引用（A**引用链法（可达性分析）**： 通过一种GC ROOT的对象（方法区中静态变量引用的对象等-static变量）来判断，如果有一条链能够到达GC ROOT就说明，不能到达GC ROOT就说明可以回收用B,B引用A）的情况

* 什么时候**触发minorGC**,什么时候**触发fullGC**

堆中年轻代的回收称为minorGC，**当Edon区域满时，会触发MinorGC**，Edon和S0中的存活对象会被复制到S1中，当S1空间不够时候，这些对象会直接移动到老年代中。存活的对象每经过一次minorGC年龄会加1，当达到某个值时这些对象会进入老年代。

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(**大对象会直接分配在老年代时**)**

（3）**方法区空间**不足（这里是永久代，java 8之后改为metaspace）

（4）通过Minor GC后进入老年代的平均大小（**年轻代中的对象年龄足够大后将进入老年代**）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**

* 内存分配策略

大对象直接进入老年代

长期存活的对象进入老年代

动态年龄判断，当Survivor中相同年龄的所有对象大小总和大于Survivor空间一半时候，年龄大于等于该对象年龄的对象进入老年代。

空间分配担保，保证minorGC是安全的，即老年代内存大小大于新生代所有对象总和时候才会进行，当不允许担保失败时候将进行FullGC。

* 类加载过程：**加载**（获取二进制字节流，生成方法区数据）、**验证**、**准备**（设置变量初始值但不等同代码中执行的初始化）、**解析**（将常量池中的符号引用替换为直接引用）、**初始化（**执行代码中的初始化**）**
* Java类加载器包括⼏种？它们之间的⽗⼦关系是怎么样的？双亲委派机制是什么意思？有什么好处？

类加载器：“通过一个类的全限定名获取描述此类的二进制字节流”的动作在jvm外部，实现这个动作的代码模块称为“类加载器”。

**类与类加载器共同确立了其在java虚拟机中的唯一性**

**启动类**加载器、**扩展类**加载器、**应用程序类**加载器、**自定义类**加载器

双亲委派模型是某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。-----例如类java.lang.Object，它存在在rt.jar中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的Bootstrap ClassLoader进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。相反，如果没有双亲委派模型而是由各个类加载器自行加载的话，如果用户编写了一个java.lang.Object的同名类并放在ClassPath中，那系统中将会出现多个不同的Object类，程序将混乱

* 如何⾃定义⼀个类加载器？你使⽤过哪些或者你在什么场景下需要⼀个⾃定义的类加载器吗？

<https://www.cnblogs.com/xrq730/p/4847337.html>

自定义类加载器（不打破双亲委派机制），重写findClass()方法，方法内部实现即找到.class文件，然后读取字节流，使用FileInputStream方法等。

从网络输入流中获取类的字节流时候可能需要加密解密操作，实现类的热部署也需要自定义类加载器。

* 动态链接和静态链接（符号引用和直接引用）

在Class文件中的常量持中存有大量的符号引用。字节码中的方法调用指令就以常量池中指向方法的符号引用作为参数。这些符号引用一部分在**类的加载阶段（解析）或第一次使用的时候就转化为了直接引用**（指向数据所存地址的指针或句柄等），这种转化称**为静态链接**。而相反的，另一部分在**运行期间转化为直接引用**，就称为**动态链接**。

动态扩展就是在运行期可以动态修改字节码，也就是反射机制与cglib

* **分派：静态分派与动态分派。**

所有依赖静态类型来定位方法执行版本的分派动作称为静态分派。静态分派的典型应用是**方法重载**，即以=左侧的对象作为依据。

动态分派是根据实际类型决定方法的执行版本，典型应用是**方法重写。**

先给出**宗量**的定义：方法的接受者（亦即方法的调用者）与方法的参数统称为方法的宗量。单分派是根据一个宗量对目标方法进行选择，多分派是根据多于一个宗量对目标方法进行选择（根据方法调用者和方法参数共同决定）

* **堆内存设置的参数是什么？**

设定堆最小内存大小-Xms

1. -Xmx：堆内存最大限制。
2. 设定新生代大小。  
   新生代不宜太小，否则会有大量对象涌入老年代

-XX:NewSize：新生代大小

-XX:NewRatio 新生代和老生代占比

-XX:SurvivorRatio：伊甸园空间和幸存者空间的占比

1. 设定垃圾回收器

​ 年轻代用 -XX:+UseParNewGC （串行） 年老代用-XX:+UseConcMarkSweepGC （CMS）

* Perm Space中保存什么数据？会引起OutOfMemory吗？
* 做GC时，⼀个对象在内存各个Space中被移动的顺序是什么？
* 你有没有遇到过OutOfMemory问题？你是怎么来处理这个问题的？处理 过程中有哪些收获？
* StackOverflow异常有没有遇到过？⼀般你猜测会在什么情况下被触发？如何指定⼀个线程的堆栈⼤⼩？⼀般你们写多少？
* 虚拟机在运行时有哪些优化策略
* 请解释StackOverflowError和OutOfMemeryError的区别？

**MySQL**

* **InnoDB支持的四种事务隔离级别名称，以及逐级之间的区别？**
* [**https://blog.csdn.net/Abandon\_Sun/article/details/79462021**](https://blog.csdn.net/Abandon_Sun/article/details/79462021)
  + READ\_UNCOMMITTED：读未提交，即能够读取到没有被提交的数据，很明显这个隔离级别无法解决脏读、不可重复读、幻读等问题。
  + READ\_COMMITTED：读已提交，即能够读取到已经被提交的数据，可以防止脏读，但不能防止不可重复读和幻读（在一个事务中的操作中发现了未被操作的数据，如其他事务新增的记录）
  + REPEATABLE\_READ：重复读取，**MySQL的默认隔离级别**，即在数据读出来之后进行加锁，即在事务整个周期内的数据都未变，即使其他事务对该数据进行了修改并提交，且**MySQL重复读取时并未提供行级锁！，**当该事务结束的时候别的事务才可以对该记录进行修改，但是无法解决幻读问题。
  + SERIALIZABLE：串行化，最高的事务隔离级别，不管一共多少事务，只有在执行完一个事务的所有子事务之后才可以执行另一个事务的所有子事务，不会产生并发问题。
* **事务的ACID：**
  + 原子性Atomic：事务开始后的所有操作要么全部做完要么全都不做，不可能停滞在某中间环节。执行过程中若出错，则会回滚到事务开始前的状态。
  + 一致性Consistency：事务将系统状态的改变进行封装，从而始终能保持系统处于一致的状态，即使事务是并发多个，系统也必须如同串行事务一样操作。
  + 隔离性Isolation：同一时间只允许一个事物请求同一个数据，隔离状态执行事务，使得它们好像是系统在给定时间内执行的唯一操作，确保每一事务在系统中认为只有自己在使用系统。
  + **持久性Durability**：**一旦事务完成以后**，该事务对数据库做的更改就持久地保存在数据库中，**不会被回滚**。
* **索引是什么，如何实现的？（结合下一个问题 B+树）**[**http://blog.codinglabs.org/articles/theory-of-mysql-index.html**](http://blog.codinglabs.org/articles/theory-of-mysql-index.html)
  + 索引是一个排序的列表，在这个列表中存储着索引的值和包含这个值的数据所在行的物理地址，在数据十分庞大的时候，索引可以大大加快查询的速度，这是因为使用索引后可以不用扫描全表来定位某行的数据，而是先通过索引表找到该行数据对应的物理地址然后访问相应的数据。
  + MySQL的MyISAM和InnoDB引擎中的索引都是使用B+树实现的，而不是使用二叉查找树或其进化品种红黑树，这个与磁盘的存取原理有关，B+树的数据结构可以更好地结合I/O的局部性原理和磁盘预读，I/O操作较少。
  + **两个引擎索引的区别**：MyISAM的索引，B+树的叶子节点的data域存放的是数据记录所在的地址，查询时先使用B+树搜索算法搜索索引，如果对应的key存在，则取出其data域的值，以该值作为地址去读取对应的数据记录——也就是说，**MyISAM的索引文件和数据文件是分离的**；而**InnoDB的数据文件本身就是索引文件**，就是说表数据文件本身就是按照B+树组织的一个索引结构，这棵树的叶子节点的data域保存了完整的数据记录，这个索引的key是数据表的主键，因此InnoDB的表数据文件本身就是主索引。 另外，MyISAM的主索引和副索引在结构上没有任何区别，只是主索引要求key值是唯一的而辅助索引key可以重复；**而InnoDB的副索引都引用主键（即主索引）作为data域，在通过副索引查找时，会先检索副索引获得主键，然后用主键到主索引中检索获得数据记录，使用自增字段作为主键时B+树的调整更少，副索引需要检索两遍索引才能找到数据，因此InnoDB不建议以过长的字段或者非单调的字段作为主键，如果没有特别的需要，请永远使用一个与业务无关的自增字段作为主键。**
* 索引使用及优化策略

<https://www.cnblogs.com/wuchanming/p/6886020.html>

**联合索引**：联合索引可以为多个字段创建一个索引。它的原理也很简单，比如，我们在（a,b,c）字段上创建一个联合索引，则索引记录会首先按照A字段排序，然后再按照B字段排序然后再是C字段

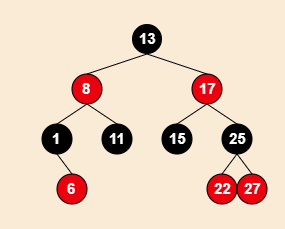
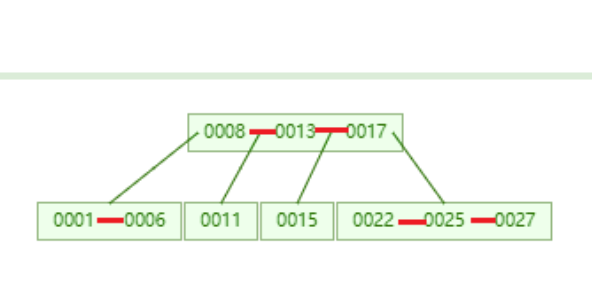
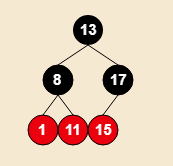
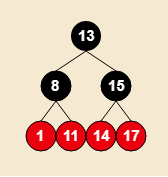
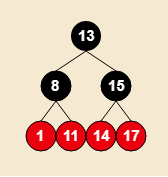
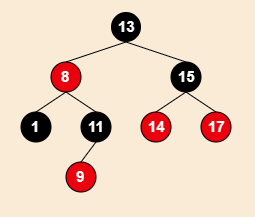
**最左前缀原理**：我们在（a,b,c）字段上建了一个联合索引，所以这个索引是先按a 再按b 再按c进行排列的，三个查询按照 （a ）, （a，b ）,（a，b，c ）的顺序都可以利用到该联合索引，这就是最左前缀匹配。即查询时必须出现最左边的a查询条件，无论和b、c出现的顺序。

**前缀索引**：前缀索引就是用列的前缀代替整个列作为索引key，当前缀长度合适时，可以做到既使得前缀索引的选择性接近全列索引，同时因为索引key变短而减少了索引文件的大小和维护开销。一般来说以下情况可以使用前缀索引：

·字符串列(varchar,char,text等)，需要进行全字段匹配或者前匹配。也就是=‘xxx’ 或者 like ‘xxx%’

·字符串本身可能比较长，而且前几个字符就开始不相同。

·前一半字符的索引选择性就已经接近于全字段的索引选择性。

* 最左前缀匹配原则，上面讲到了
* 主键外检一定要建索引
* 对 where,on,group by,order by 中出现的列使用索引
* 对**较小的数据列使用索引**,这样会使索引文件更小,同时**内存中也可以装载更多的索引键**
* 索引列不能参与计算，保持列“干净”
* 为较**长的字符串使用前缀索引**
* 尽量的扩展索引，不要新建索引。比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可
* 查询where条件数据类型不匹配也无法使用索引   
  字符串与数字比较不使用索引;
* 二叉查找树；二三树 **红黑树 B树 B+树**
  + 二叉查找树(有序二叉树)：最多有两个子节点，左子树所有节点值小于根节点值，右子树所有节点值大于根节点值。每个节点Node包含两个分别指向左右节点的Node、一个用于排序的Key、一个该节点包含的值Value以及一个记录该节点及所有子节点个数的Number。
  + <http://www.cnblogs.com/yangecnu/p/Introduce-2-3-Search-Tree.html>  
    **二三树是一种平衡的查找树，但不是二叉树。**二三树的每个节点保存1个或2个Key值。对于普通的2-Node，保存1个用于排序的Key和Value值以及左右两个子节点，左右子节点分别满足Key值小于和大于根节点Key值；对于3-Node，保存两个Key与对应Value值，以及左中右三个子节点，这三个子节点分别满足Key值小于根节点最小的Key值、介于两个Key之间、大于根节点最大的Key值；**二三树通过不同情况下选用不同的插入方式来保持查找树是平衡的**。
  + [**https://blog.csdn.net/qq\_26222859/article/details/80631121**](https://blog.csdn.net/qq_26222859/article/details/80631121)<http://lauandy.cc/2017/08/03/btree-and-bplustree/>  
    B树是**平衡多叉**树，可以看做是对二三树的一种扩展，即允许每个节点有最多M个子节点，其中M为B树的阶。每个节点的多个key按升序排列，且有 **节点所含key值的个数 = 节点的子树的个数 – 1**，就意味着某节点的每个子树所在的位置是由该节点的key值的分布决定的。**B树的另外一个特性就是所有的叶子节点都处于同一层，也就是说从根节点到任意节点的深度都相等（平衡）**。
  + B+树是B树的一种变形，索引每个节点一般是磁盘页的大小，即4kb，三层的b+树可以表示上百万的数据，二者的差异在于，非叶子节点的节点（就是中间节点）的**子树的个数 = 该节点的key的个数**，这是因为B+树中的**中间节点的key并不用于保存数据，而只用来索引**，而**叶子节点中包含了全部的key以及value**；所有的中间节点的key都同时存在于子节点，且在子节点的key中是最大或者最小的；**所有的子节点都按照升序以指针连接在一起**，**即增加顺序访问指针提高了区间访问的性能。**由于B+树的中间节点不再存储value，那么同样大小的磁盘页可以容纳更多的节点元素，**因此数据量相同的B树与B+树相比，后者更加“矮胖”，从而可以减少磁盘I/O**。B+树因为每次都要查找到叶子节点，因此**查找性能稳定。**范围查询时只需在叶子节点顺序遍历**，更简单。**
  + [**https://www.cnblogs.com/yangecnu/p/Introduce-Red-Black-Tree.html**](https://www.cnblogs.com/yangecnu/p/Introduce-Red-Black-Tree.html)**限定左旋的红黑树：是二三树的变体**[**HashMap-红黑树**](https://www.cnblogs.com/liwei2222/p/8013367.html)[**在线生成红黑树**](https://sandbox.runjs.cn/show/2nngvn8w) **红黑树**：Java HashMap的红黑树是普通的红黑树，既可以左旋也可以右旋的，与上面blog里的红节点只能作为左子树节点的规则是不一样的。红黑树从根到叶子节点的最长路径不会超过最短路径的二倍。**红黑树是黑色平衡的**，即从根节点到所有叶子节点的路径，经过的黑色的节点数是相等的。 本质上来说，红黑树这个数据结构的基本思想源自于阶数为4的B树(也就是2-3-4树)。一个这样的红黑树，本质上是一个这样的4阶B树：  
      
    因此对于**红黑树的插入**等操作，可以类比4阶B树的相关操作。  
    1. 要将一个节点A插入，首先要将该节点的key值与树中的节点Key值相比较，最后找到一个正确的null节点位置，然后替换此null节点。此时节点A的颜色为红色。  
    2. **if** A节点父节点为黑色，那么插入过程结束，因为红色的节点不会影响红黑树的平衡  
     **if** A节点父节点为红色，那么此时要看A节点的父节点的兄弟节点，也就是A节点的叔叔节点：  
     **if** A节点没有叔叔节点，则进行红黑树的**旋转操作（左旋或右旋）**，最后的情况是，A节点的爷爷节点的位置由A节点的父节点替换了，A节点原来的爷爷节点在旋转后成为了A节点的父节点的子节点，也就是**A节点的爷爷节点成为了A节点的兄弟节点**。  
    注意，旋转的过程中，不光A节点的父节点与爷爷节点的位置要变换，两个节点的颜色也要变：  
     插入14 →   
     if A节点有叔叔节点（可知A节点的叔叔节点颜色应该与A节点的父节点相同，即为红色） 则将A节点的父节点和叔叔节点颜色全部置为黑色，而将A节点的爷爷节点的颜色置为红色，以保证整个红黑树仍然是黑色平衡的。 **此时A节点的爷爷节点为红色，可能会导致不再满足红黑树的约束，因此要对变成红色的爷爷节点进行与插入A节点时相同的操作，来维持整个红黑树的平衡，这个过程可能是一个递归的过程**。  
     插入9 → 
  + 关于红黑树左旋和右旋：左旋右上位，右旋左上位。
* **什么时候设置了索引但无法使用？索引建立的原则？**[**https://www.jb51.net/article/37190.htm**](https://www.jb51.net/article/37190.htm)
  + 在JOIN操作中，MYSQL只有当主键和外键的数据类型相同时才会使用索引，否则即使建立了索引也无法使用；WHERE语句的查询条件中有 != 或者在查询条件中使用了函数，将无法使用索引；…
* 1. 在经常用作过滤器的字段上建立索引，经常作为where的字段；   
  2. 在SQL语句中经常进行GROUP BY、ORDER BY的字段上建立索引；

3. 用于联接的列（主健/外健）上建立索引；

3. 在经常存取的多个列上建立复合索引，但要注意复合索引的建立顺序要按照使用的频度来确定  
4. 在**不同值较少的字**段上不必要建立索引，如性别字段；   
5. 对于**经常存取**的列避免建立索引；

6. 避免选择大型数据类型作为索引

首先要确定表的大量操作是查询还是增删改，表的大小，小型表建立索引可能会影响性能。创建索引为了提高检索速度，索引的创建和维护需要耗费时间，随着数据量的增加而增加，索引表还需要占空间。 

* **解释MySQL外连接、内连接与自连接的区别**[**https://blog.csdn.net/aschulianwuyanzu/article/details/77879690**](https://blog.csdn.net/aschulianwuyanzu/article/details/77879690)
  + 自连接 （select a.id, b.score from student\_score a , student\_score b where a.id = b.scoreid）：其实是对同一张表进行的操作。
  + 内连接JOIN：显示内容为两个表内连接条件匹配的行。
  + 左外连接left join：左表为主表，显示内容为左表全部记录以及右表中与满足匹配条件的部分记录。
  + 右外连接right join：与左外连接相反。
* **乐观锁和悲观锁是什么以及在MySQL的实现？**[**https://blog.csdn.net/puhaiyang/article/details/72284702**](https://blog.csdn.net/puhaiyang/article/details/72284702)
  + 悲观锁：特点是先获取锁再执行业务操作，在MySQL中可以使用select .. for update实现悲观锁，会获取要操作的数据行的行级锁，在当前事务结束时自动释放，因此必须在事务中使用。
  + 乐观锁：不是数据库自带的，需要自己实现，特点是先执行业务操作，当需要更新数据库时再去请求锁。常用的实现方式是版本号机制，为数据库表增加一个数字类型的version字段。读取时将version字段值一起读出，当提交更新时，判断数据库表对应记录的当前版本信息与第一次提取的version值是否一致，若一致则进行更新，不一致则回滚。
* **MySQL表级锁、页级锁、行级锁？InnoDB的标准行级锁有哪2种，解释其含义。**

<https://www.cnblogs.com/rjzheng/p/9950951.html>

* + **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，可能出现死锁，锁定粒度最小，锁冲突概率最低，并发度最高；页级锁介于二者之间，可能会出现死锁。
  + InnoDB行锁是通过给**索引上的索引项**加锁来实现的只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁（实现时record lock + gap lock）。发生死锁时候，一般会检测到，然后让 事务回退释放锁，另一个事务获取锁完成事务。
  + MyIsam使用表级锁，不会产生死锁。
  + **InnoDB**的标准**行级锁是共享锁和排他锁**：select … for update就是排他锁，**排他锁申请前提：**当前没有线程对该结果集中的任何行使用排他锁或共享锁，否则申请会阻塞。select … lock in share mode为共享锁，**共享锁允许事物读一行记录**，不允许任何线程对该行记录进行修改。**排他锁允许当前事物删除或更新一行记录**，其他线程不能操作该记录。
* **常用检索**

**DISTINCT**：返回不同的值，重复值只返回一次，select distinct..

**LIMIT**：限制返回的行数，from products limit 5；limit 5,5；

**ORDER BY**：对选出的行进行排序 order by prod\_name（列名），可以按多个列排序，列名可以有多个，按照某个列降序的时候加DESC关键字，order by prod\_name DESC,prod\_price

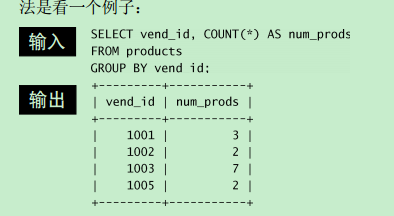
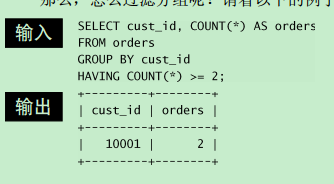
**WHERE**：可以对结果进行过滤，筛选出某个列等于、不等于或在某个范围的行。WHERE可以使用AND、OR、IN（范围）、NOT操作符进一步过滤。WHERE操作中判断的条件都是已知的。

**LIKE**：使用通配符，如where prod\_name LIKE ‘%jet’表示任意以jet结尾的产品名

**Concat（）函数**：select Concat（prod\_name，prod\_country）as prod\_title…，可以使用别名，拼接后作为一个列显示

**汇总数据**：使用**聚集函数**，avg()、count()、max()、min()、sum()等直接返回某列的均值、总行数等，select AVG(prod\_price) from ..

**GROUP BY**：分组把数据分为多个逻辑组，对每个组进行聚集计算（汇总）**HAVING**：过滤分组。

**使用子查询**：select的结果作为where的判断条件

**联结和关系表：外键（foreign key）** 外键为某个表中的一列，它包含另一个表**，vendors表的主键又叫作products的外键，它将vendors表与products表关联，利用供应商ID能从vendors表中找出相应供应商的详细信息。的主键值，定义了两个表之间的关系。**利用外键建立联结。

内部联结进行等值联结，外部联结会包含那些没有关联的行，指定left或right来包含outer join 左侧或右侧的所有行。

**UNION**：组合两个以上的select语句，select语句必须返回具有相同的行数

**视图：VIEW，创建标的一部分。**

* **数据库的横向、纵向分割**[**数据库优化 分字诀 分表（纵向拆分，横向分区）**](https://www.cnblogs.com/ideacore/p/6423718.html)
  + **纵向拆分：根据列的属性进行拆分，比如在业务中被访问的情况，或者列更新的频率。**
  + **横向拆分：根据行的属性进行拆分，比如将帖子的数据库按照发帖时间进行拆分。**
* **Mysql 的存储引擎是什么？MyISAM和InnoDB的区**别？  
  [**https://www.cnblogs.com/andy6/p/5789248.html**](https://www.cnblogs.com/andy6/p/5789248.html)
  + MySQL中的数据用不同的技术存储在文件（或者内存）中，这些技术中的每一种都使用不同的存储机制、索引技巧、锁定水平，从而提供不同方面的功能、性能。这些不同的技术以及配套的相关功能就称作MySQL的存储引擎（也称作表类型）。MyISAM和InnoDB是两个MYSQL常用的存储引擎。
  + InnoDB是MySQL的默认存储引擎，提供了对数据库ACID**事务的支持**，实现了SQL标准的四种隔离级别；还提供了行级锁，并且**是唯一提供外键约束的引擎**，其设计目标是**处理大容量数据库系统**。当需要使用数据库事务时，该引擎当然是首选，并且可以根据事务日志进行自动灾难恢复。由于锁的粒度更小，写操作不会锁定全表，所以在并发较高时，使用Innodb引擎会提升效率。**更新密集，支持事务，外键，auto\_increment，使用聚集索引，data域存储数据**
  + **MyISAM没有提供对数据库事务的支持**，也不支持行级锁和外键，因此当INSERT(插入)或UPDATE(更新)数据时即写操作需要锁定整个表，效率便会低一些。如果表的**读操作**远远多于写操作且不需要数据库事务的支持，那么MyIASM也是很好的选择。**读操作更多，不支持事务，使用表级锁，并发度低，使用非聚集索引**
  + **MEMORY引擎** 使用系统内存存储，速度更快，表数据是长度不变的格式。适用于：

**目标数据小，被频繁访问**

**数据丢失不会对服务产生影响。**

* **操作系统**
* 并发与并行：
  + 并发是并行的子集。系统中有多个任务同时存在叫并发，系统中有多个任务同时执行叫并行。并发是逻辑上的同时发生，并行更多是侧重于物理上的同时发生。并发往往是指程序代码的结构支持并发，并发的程序在多cpu上运行起来才有可能达到并行，并行往往是描述运行时的状态。并行——双十一 , 零时零分零秒,三十万用户同时登录淘宝网；并发——双十一 , 零时零分零秒 至 两点零分零秒 共有一百万用户登录淘宝网.
  + 在操作系统中，并发是指一个时间段中**有几个程序都处于已启动运行到运行完毕之间，且这几个程序都是在同一个处理机上运行**，但**任一个时刻点上只有一个程序在处理机上**运行。只有一个CPU的话，根本不可能真正同时进行一个以上的线程，它只能把CPU运行时间划分成若干个时间段,再将时间段分配给各个线程执行，在一个时间段的线程代码运行时，其它线程处于挂起状。
  + 当系统有一个以上CPU时,则线程的操作有可能非并发。当一个CPU执行一个线程时，另一个CPU可以执行另一个线程，两个线程互不抢占CPU资源，可以同时进行，这种方式我们称之为并行(Parallel)。
* **进程和线程的区别**
  + 进程:具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.
  + 线程:进程的一个实体,是**CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位**.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.
  + 区别：**进程有独立的地址空间**，一个进程崩溃后，在保护模式下不会对其它进程产生影响；线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间。**一个程序至少有一个进程,一个进程至少有一个线程.**
* 线程的状态：

<https://blog.csdn.net/pange1991/article/details/53860651>

* + 1. **初始(NEW)：**新创建了一个线程对象，但还没有调用start()方法。
  + 2. **运行(RUNNABLE)**：Java线程中将就绪（ready）和运行中（running）两种状态笼统的称为“运行”。
  + 线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取CPU的使用权，此时处于就绪状态（ready）。就绪状态的线程在获得CPU时间片后变为运行中状态（running）。
  + 3. **阻塞(BLOCKED)**：表示线程阻塞于锁，（需要获取某个条件或资源时候不能立即获得就会阻塞）
  + 4. **等待(WAITING)**：进入该状态的线程需要等待其他线程做出一些特定动作（通知或中断）。
  + 5. **超时等待(TIMED\_WAITING)**：该状态不同于WAITING，它可以在指定的时间后自行返回。
  + 6. **终止(TERMINATED)**：表示该线程已经执行完毕。
* **如何开启线程**

1. 继承Thread类，重写run()方法
2. 实现Runnable接口，重写run()方法
3. 通过Callable和Future创建线程，call()是有返回值的run

启动线程的方式都是start()方法

* Sleep、wait、yield方法

Sleep()方法会出让cpu时间片，但不会释放锁，当是指定时间到达之后会继续执行该线程，wait()方法是是当前线程释放锁和CPU时间片。Sleep()是Thread类的方法，wait()方法是Object类的方法**Thread.yield()**，一定是当前线程调用此方法，当前线程放弃获取的CPU时间片，但不释放锁资源，由**运行状态变为就绪状态**，让OS再次选择线程。作用：让相同优先级的线程轮流执行，但并不保证一定会轮流执行。

* 进程，线程，协程之间的区别  
  <https://www.cnblogs.com/work115/p/5620272.html>
  + 进程与线程
    - 调度：线程作为**任务调度和执行**的基本单位，进程作为**资源分配**的基本单位；
    - 并发性：不仅进程之间可以并发执行，同一个进程的多个线程之间也可并发执行；
    - 拥有资源：**进程是拥有资源的一个独立单位**，**线程不拥有系统资源**，但可以访问隶属于进程的资源；
    - 系统开销：在创建或撤消进程时，由于系统都要为之分配和回收资源，导致系统的开销明显大于创建或撤消线程时的开销
  + 线程与协程：协程与线程类似，一个程序可以包含多个协程，区别在于协程的切换是由当前的协程主动切换到其他协程。
* 什么是Daemon线程？它有什么意义？与用户线程的区别？
  + 所谓守护线程，是指在程序运行的时候在后台提供一种通用服务的线程，比如垃圾回收线程就是一个很称职的守护者，并且这种线程并不属于程序中不可或缺的部分。因此，**当所有的非守护线程结束时，程序也就终止了，同时会杀死进程中的所有守护线程**。反过来说，只要任何非守护线程还在运行，程序就不会终止。用户线程和守护线程两者几乎没有区别，唯一的不同之处就在于虚拟机的离开：如果用户线程已经全部退出运行了，只剩下守护线程存在了，虚拟机也就退出了。 因为没有了被守护者，守护线程也就没有工作可做了，也就没有继续运行程序的必要了。
* 线程之间是如何通信的？
  + **synchronized**：同步，当多个线程访问共享变量时，谁拿到锁谁就执行。
  + **共享变量，volatile关键字（内部实现原理）**
  + **Lock/Condition机制**
  + **wait-notify**：当条件未满足时，线程就用wait()方法主动放弃cpu资源，进入阻塞态，等待其他线程达成条件后notify()该线程。缺点是如果在调用wait()方法之前，其他线程已经使用了notify(),那么在调用wait()方法之后该线程永远不会被唤醒。
  + **管道通信**：java.io.PipedInputStream和PipedOutputStream
* **进程之间**是如何通信的？
  + Socket通信，
  + 无名管道、有名管道，管道是固定大小的缓冲区，通过缓冲区进行数据的读取写入操作，数据读取是一次性操作。
  + 信号（通知某一进程发生什么事了）
  + 信号量，作为计数器，控制多个进程对共享资源的访问。
  + **共享内存，每个进程都绑定到共享内存，完成通信后脱离。**
  + 消息队列
* 为什么要有线程，而不是仅仅用进程？
  + **切换进程的开销比线程的大。**
  + 线程具有更高的并发层次：进程属于在处理器这一层上提供的抽象；线程则属于在进程这个层次上再提供了一层并发的抽象。
  + 线程可以**更有效地利用cpu资源**：如果将一个进程分解为若干个线程，则可以让不同的线程运行在不同的核上，从而提高了进程的执行速度。
* 什么是可重入锁（ReentrantLock）与不可重入锁？  
  <https://www.cnblogs.com/dj3839/p/6580765.html>
  + 二者的区别在于能否递归地使用，可重入锁不会产生下面不可重入锁的那种死锁情况。
  + 首先锁是通过wait()和notify()方法实现的，当wait()的时候线程就会陷入阻塞态，等待被其他线程调用notify()唤醒。
  + **不可重入锁的死锁情况**：如果当前线程的一个方法已经获得了锁，那么当该方法又调用一个需要获得同一个锁的方法时，这个方法由于获得不到锁，就会陷入阻塞态。而由于这个线程陷入阻塞态，这个已经获得的锁也不会被释放，因为陷入阻塞态的线程已经没有能力调用notify()方法，而是永远等待着被自己外层方法获得的锁。
* 什么是线程组，为什么在Java中不推荐使用？
  + 线程组是用来管理线程的一个线程集合，由于它不是线程安全的，已经被弃用。
* **乐观锁和悲观锁的理解及如何实现，有哪些实现方式**？

<https://juejin.im/post/5b4977ae5188251b146b2fc8>

* + 悲观锁：总是假设最坏的情况，每次取数据时都认为其他线程会修改，因此都会加锁，当其他线程想要访问时，都会进入阻塞态。Java **synchronized**就是悲观锁。
  + 乐观锁：总是假设在自己读写数据时，不会有其他线程对数据进行修改，因此只是在**更新时判断一下数据是不是被其他线程修改**了，一般用**版本号机制或者CAS操作**（Compare-And-Swap，该操作是具有原子性的赋值操作）实现。如java.util.concurrent.atomic原子变量类就使用了CAS操作。
* Java中用到的线程调度算法是什么？
  + 抢占式。一个线程用完CPU之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。
* 同步方法和同步块，哪个是更好的选择？  
  <http://www.importnew.com/21866.html>
  + 一般来说同步块更好。因为同步将会锁住整个对象（java的每个对象都有一把锁），一般来说并不是整个方法都需要参与同步，而同步块的粒度小，性能会高一些。
* 如何控制某个方法允许并发访问线程的个数？
  + 使用信号量Semaphore <https://www.jianshu.com/p/561cc5148f65>
* 在Java中wait和sleep方法的不同？
  + **wait()方法调用后，线程将会放弃CPU资源**，等待其他线程调用notify()方法将其唤醒。
  + sleep()调用后线程不会放弃CPU资源，相当于空转。
* Thread类中的yield方法有什么作用？
  + 当线程调用了yield()方法，**将会放弃当前持有的cpu资源，而是重新从运行态进入就绪态**，因此获得cpu资源的可能是其他线程，也可能又是该线程。
* Object类中的方法：clone, getClass, toString, finalize, equals, hashCode, wait, notify, notifyAll
* 谈谈wait/notify关键字的理解
* **为什么wait, notify 和 notifyAll这些方法不在thread类里面而是在Object类中**？
  + 这个涉及到**wait什么的问题。等的是某个对象上的锁，大家（多个线程）竞争这个锁**，是吧，wait和notify方法都放到目标对象上，那这个对象上可以维护线程的队列，可以对相关线程进行调度（方法和方法所操纵的数据要在一起）.**如果将wait方法和线程队列都放到Thread中，那么就必然要求某个Thread知道其他所有Thread的信息，（大家都要相互知道），这合理吗？很容易出问题的。**
  + 在java的内置锁机制中，每个对象都可以成为锁，也就是说每个对象都可以去调用wait，notify方法，而Object类是所有类的一个父类，把这些方法放在Object中，则java中的所有对象都可以去调用这些方法了。
  + 一个线程可以拥有多个对象锁，wait，notify，notifyAll跟对象锁之间是有一个绑定关系的，比如你用对象锁aObject调用的wait()方法，那么你只能通过aObject.notify()或者aObject.notifyAll()来唤醒这个线程，这样jvm很容易就知道应该从哪个对象锁的等待池中去唤醒线程，假如Thread.wait()，Thread.notify()，Thread.notifyAll()来调用，虚拟机根本就不知道需要操作的对象锁是哪一个。
* 什么导致线程阻塞？
  + 当线程想要访问一些共享变量时，要获取访问它的权限即锁，当锁被别的线程持有时，该线程进入阻塞态。
* Synchronized内部实现

锁的实现是在jvm层面，synchronized映射成字节码指令就是增加来两个指令：monitorenter和monitorexit。当一条线程进行执行的遇到monitorenter指令的时候，它会去尝试获得锁，如果获得锁那么锁计数+1（为什么会加一呢，因为它是一个可重入锁，所以需要用这个锁计数判断锁的情况），如果没有获得锁，那么阻塞。当它遇到monitorexit的时候，锁计数器-1，当计数器为0，那么就释放锁。

**自旋：**java线程其实是映射在内核之上的，线程的挂起和恢复会极大的影响开销。并且jdk官方人员发现，很多线程在等待锁的时候，在很短的一段时间就获得了锁，所以它们在线程等待的时候，并不需要把线程挂起，而是让他无目的的循环，一般设置10次。这样就避免了线程切换的开销，极大的提升了性能

* synchronized和lock的区别，读写锁ReentrantReadWriteLock

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

synchronized是java语言层面的实现，虚拟机层面，lock的实现是一个类。

Synchronized的锁在同步代码块执行完毕或者发生异常时会释放锁，lock的锁需要在finally中去手动释放

Synchronized锁的获取其他线程在等待时会阻塞，lock可以去尝试获取不用一直等待

Synchronized无法获取锁的状态，lock可以知道线程有没有成功获得锁

Synchronized是不可中断锁，Lock是可中断锁，如果某一线程A正在执行锁中的代码，另一线程B正在等待获取该锁，可能由于等待时间过长，线程B不想等待了，想先处理其他事情，我们可以让它中断自己或者在别的线程中中断它，这种就是可中断锁。

Synchronized是非公平锁，lock可以实现公平锁，即按照线程请求锁的顺序来给予锁。

ReentrantReadWriteLock里面提供了很多丰富的方法，不过最主要的有两个方法：readLock()和writeLock()用来获取读锁和写锁。可以同时进行读操作，但是不能同时进行读写或同时写操作。如果有一个线程已经占用了读锁，则此时其他线程如果要申请写锁，则申请写锁的线程会一直等待释放读锁。如果有一个线程已经占用了写锁，则此时其他线程如果申请写锁或者读锁，则申请的线程会一直等待释放写锁。

* 谈谈**volatile关键字的作用和内部实现原理**
  + 保证了可见性和禁止指令重排**，对变量的操作use之前必须先load进工作内存，修改后即assign和store、write连续执行，保证修改后立即同步到主存中。** 禁止指令重排在赋值语句后多执行lock addl 操作，设置内存屏障，保障不会把后面的指令重排序到内存屏障之前执行。
  + 不能保证写的线程安全，因为如+1操作，先从主存中取值，再+1，同步回主存，volatile只能保证从主存中取值时候是最新的值，但是在操作过程中可能已经被别的线程修改。
  + 对变量的写操作不依赖于当前值，或能确保只有单一的线程修改变量的值
  + 变量不需要与其他的状态变量共同参与不变
* 原子性、可见性、有序性

Synchronized保证原子性，synchronized、volatile、final保证可见性，有序性：volatile和synchronized保证线程之间操作的有序性

* Java线程实现

一对一的线程模型，一条java线程就**映射到一条轻量级进程**中，用户线程完全建立在用户空间中，**轻量级进程**作为用户线程和内核线程的桥梁，可以使用内核提供的线程调度功能和处理器映射。

* 死锁的四个必要条件？

（1） 互斥条件：一个资源每次只能被一个进程使用。

（2） 占有且等待：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3）不可强行占有:进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） 循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

* 谈谈对Synchronized关键字，类锁，方法锁，重入锁的理解
* static synchronized 方法的多线程访问和作用
* 同一个类里面两个synchronized方法，两个线程同时访问的问题
* 你如何确保main()方法所在的线程是Java程序最后结束的线程？
  + 可以在main()函数中调用所创建的线程的join()方法。join（）方法的作用是将cpu资源转让给一个线程，等待该线程完成后，原线程（此处即是main()函数的线程，它也是个用户线程）才能继续用下运行。
* 谈谈ThreadLocal关键字的作用
* IO中同步与异步，阻塞与非阻塞区别

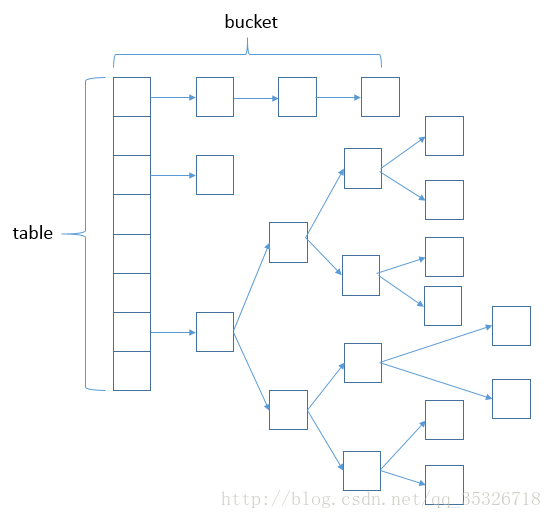
<https://www.yuque.com/page/luan.ma/io-model>

* 什么是Callable和Future?
* ThreadLocal、synchronized 和volatile 关键字的区别
* synchronized与Lock的区别
* ReentrantLock 、synchronized和volatile比较
* 在Java中CycliBarriar和CountdownLatch有什么区别？
* CopyOnWriteArrayList可以用于什么应用场景？
* ReentrantLock的内部实现
* lock原理
* Java中Semaphore是什么？
* Java中invokeAndWait 和 invokeLater有什么区别？
* 多线程中的忙循环是什么?
* 怎么检测一个线程是否拥有锁？
* 对象锁和类锁是否会互相影响？
* 什么是线程池，如何使用?
* Java线程池中submit() 和 execute()方法有什么区别？
* Java中interrupted 和 isInterruptedd方法的区别？
* 用Java实现阻塞队列
* BlockingQueue介绍：
* 多线程有什么要注意的问题？
* 如何保证多线程读写文件的安全？
* 多线程断点续传原理
* 断点续传的实现
* 实现生产者消费者模式
* 用Java写一个会导致死锁的程序，你将怎么解决？
* SimpleDateFormat是线程安全的吗?
* Java中的同步集合与并发集合有什么区别？
* Java中ConcurrentHashMap的并发度是什么？
* 什么是Java Timer类？如何创建一个有特定时间间隔的任务？

**集合+数据结构**

* Collection 和Collections 的区别？
* 修改对象A的equals方法的签名，那么使用HashMap存放这个对象实例的时候，会调用哪个equals方法？
* List,Set,Map的区别
* List和Map的实现方式以及存储方式
* **HashMap的实现原理、从源码层级描述put方法、HashMap的扩容如何实现**

**HashMap扩容后rehash很巧妙，扩容使原来容量括为两倍，即高位多1，rehash之后，原来的hash值在高位随机为0或1，为1，则扩容后的位置是原来的位置+扩容的长度，为0则保持不变，还保持均匀分布**<https://www.cnblogs.com/yangming1996/p/7997468.html?tdsourcetag=s_pctim_aiomsg>

* + JDK8之前，HashMap的内部是由数组+链表实现的，JDK8开始对于链表长度超过8的链表将转存为红黑树。  
    
* HashMap、HashTable、ConcurrentHashMap原理、区别

<https://www.cnblogs.com/yangming1996/p/8031199.html>

<https://www.cnblogs.com/snowater/p/7742287.html>

* + HashTable是线程安全的，但是每次进行操作都要把整个表锁住，容易并发冲突，并发效率低。 ConcurrentHashMap相当于对HashTable进行的改进。 JDK1.8之前的ConcurrentHashMap使用的是分段锁技术，这样只需要获取对应段的锁就可以了。JDK1.8开始不再使用分段锁，而是对每一行数据（也就是Node级别）进行加锁，粒度更小，并发冲突概率更低。
  + ConcurrentHashmap中baseCount、Node、table等都是volatile修饰
  + sizeCtl来表示阈值或是否有线程在操作扩容，0：默认值；-1：代表哈希表正在进行初始化大于0：相当于 HashMap 中的 threshold，表示阈值；小于-1：代表有多个线程正在进行扩容
  + 初始化table时候判断sizeCtl是否小与0，不小于0则直接进行扩容，并设置threshold，否则当前线程yield
  + Put操作时候，如果table没有初始化则执行初始化，如果当前hash值的位置为空，则以cas操作添加一个新节点至该位置，如果当前节点是MOVED类型，则执行协助扩容操作，如果当前节点是普通节点，则将当前桶的头节点锁住，进行遍历执行修改或添加到链表末尾，最后会判断是否需要裂变为红黑树。
  + 扩容将旧节点转移到新节点时候，每个线程处理的桶首节点的数量最少是16，会根据CPU数量进行计算，每个线程对所处理的桶进行转移到新table中，当前线程任务完成时候，table = nextTab，进行转移时用synchronized锁住这个桶的头结点执行链表或红黑树的迁移。
  + HashMap(HM)与HashTable(HT)：HT的key与value都不可以为null，而HM可以存储null键和null值；HM不是线程安全的，而HT是，实现线程安全的方式是在修改数据时锁住整个HashTable，HashTable内部的方法都使用了synchronized方法，重量级锁，效率很低
  + HashTable内部hash冲突使用链表，hashTable插入新节点是在链表头结点，hashmap和concurrentHashMap是插入链表尾部
* HashMap为什么线程不安全

增加或删除节点时，会先得到现在的链表的头结点，之后在进行操作，可能另一个线程操作时候，该节点已被修改过

resize后的newTable也可能是不同步的

* HashMap与HashSet的区别
  + 主要区别是：HashMap实现了Map接口而HashSet实现了Set接口；HashMap存储的是键值对而HashSet仅仅存储对象。
* **LinkedList**

**内部节点有双向指针**，指向前一个或后一个节点，保存有头结点和尾节点，当遍历查找某个节点时候会先判断其位于前半部还是后半部，拷贝是浅拷贝，拷贝出来的结果是两个链表都引用的同一个元素内容。

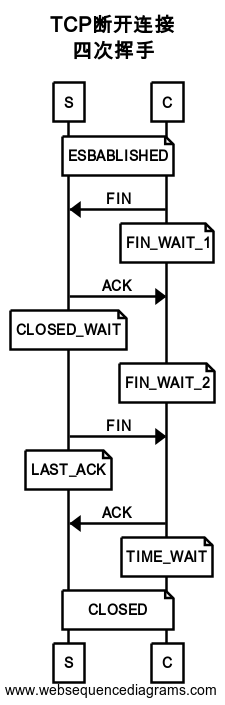
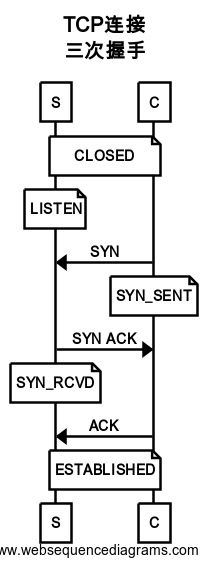
* Array和ArrayList有何区别？什么时候更适合用Array
  + ArrayList内部是一个数组，进行插入删除操作时候，其后的所有元素要移动后面的所有的元素，末尾插入删除叫容易，适合读操作比较多的场景。
  + 区别：数组可以包含**基本数据类型和引用类型**，ArrayList只能包含**引用类型**，因此需要对基本数据类型进行装箱，降低效率；ArrayList可以通过内部方法自动调整容量，ArrayList调整容量时候，一般在原先长度上扩充一半，而数组不能调整大小；ArrayList是List接口的实现类，相比数组支持更多的方法和特性如addAll()、iterator()等。
  + 场景：集合长度固定则选择数组，否则选择ArrayList——但是如果长度增长频繁，应该考虑预设ArrayList的长度或者使用LinkedList代替，因为ArrayList每次扩容都要进行数组拷贝；对于使用基本数据类型的情况尽量使用数组；当需要对集合进行比较频繁的插入和移动，使用ArrayList。
* Comparable和Comparator接口有何区别？
* Java集合的快速失败机制 “fail-fast”？
* fail-fast 与 fail-safe 之间的区别？
* BlockingQueue是什么？
* Iterator类有什么作用
* poll()方法和remove()方法区别？

**计算机网络**

* 从网络加载一个10M的图片，说下注意事项
  + 浏览器缓存 -> DNS域名解析 -> TCP连接 -> HTTP请求与相应
* TCP状态机、状态转换图

<https://www.cnblogs.com/Toonter/p/5926094.html>

<https://blog.csdn.net/wenqian1991/article/details/40110703>

* + 
* TIME\_WAIT存在的理由：

可靠地实现TCP全双工连接的终止，**允许超时重传ACK，确保服务端能被关闭。**

允许**老的重复分节（数据报）**在网-----------------------------------------络中消逝。

* OSI网络体系结构与TCP/IP协议模型

TCP/IP体系结构：

应用层：*直接为用户的应用进程提供服务，如 HTTP、支持文件传输的 FTP 协议等*

运输层：向两个主机中进程之间的通信提供服务，复用和分用，协议有TCP/UDP

**网际**层：负责为分组交换网上的不同主机提供通信服务，传送IP数据报。网络层的另一个任务就是选择合适的路由。**路由器工作在网络层。**

**网络接口层：**定义如何使用实际网络（如Ethernet、Serial Line等）来传送数据。

**OSI七层网络：**

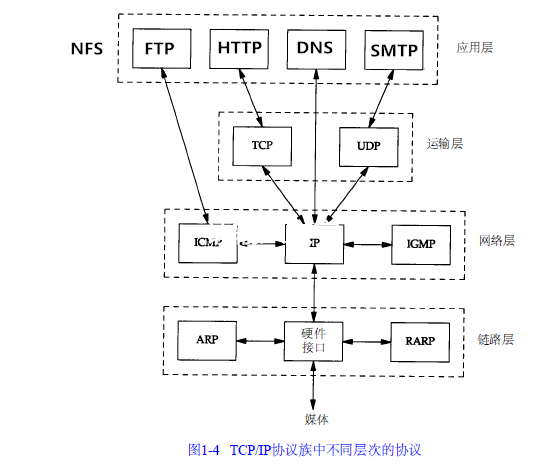
**表示层：对数据进行格式化，负责不同系统间的通信，对不同的编码方式进行转换。**

**会话层（Session Layer）**：负责在网络中的两节点之间建立、维持和终止通信。

**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，提供了错误检测和纠正等。**交换机工作在数据链路层，进行帧解码并传送。**

**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。**用来传输比特流。**

<https://www.cnblogs.com/gdayq/p/5797645.html>  
<https://blog.csdn.net/hguisu/article/details/38700899>

* + TCP/IP模型
* 三次握手（TCP建立连接）和四次挥手（断开连接）的具体过程，要说出状态的转换

第一次握手：建立连接时，客户端发送同步位SYN+序号到服务器，并进入**SYN\_SENT状态**，等待服务器确认；

第二次握手：服务器收到syn包，必须确认客户的SYN（ack=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入**SYN\_RECV状态**；

第三次握手： 客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=k+1），此包发送完毕，客户端和服务器**进入ESTABLISHED**（TCP连接成功）状态，完成三次握手

第一次挥手：主动关闭方发送一个FIN进入FIN\_WAIT\_1状态

第二次挥手：被动方发送ACK给对方，确认号为收到序号+1，进入close\_wait状态，主动方接收到ACK进入FIN\_WAIT\_2状态

第三次挥手：被动方发送FIN给主动发，进入last\_ack状态

第四次挥手：主动方发送ACK给被动方进入TIME\_WAIT状态，被动方收到ACK后进入CLOSED状态。

* 为什么TCP链接需要三次握手，两次不可以么，为什么？  
  <https://blog.csdn.net/xumin330774233/article/details/14448715>
  + 为了防止已失效的连接请求报文段（比如延误的连接请求报文）突然又传送到了服务端，因而产生错误。

• 客户端向服务器发送了FIN后服务器一直不发送FIN怎么办（过多的CLOSE\_WAIT怎么解决）

<https://blog.csdn.net/wwd0501/article/details/78674170>

CLOSE\_WAIT状态会持续7200s，之后会最多发送9个探针，每次间隔75s，探测客户端是否存活，若客户端没有活动则会关闭。修噶这几个参数可以减少CLOSE\_WAIT过多问题。（发送探针的行为也适用于TCP的保活，主要是服务端想知道客户端是否崩溃。）

* TCP协议如何来保证传输的可靠性
  + 主要是三个：**超时重传**（防止丢失，确认丢失和确认迟到相关）、**检测报文 段的校验和**（防止数据错误）、**序号机制**（防止乱序）
* TCP流量控制、拥塞控制
  + 以字节为单位的**滑动窗口**协议，自适应超时重传时间选择
  + 滑动窗口实现流量控制，（TCP发送方或接收方的缓冲区满了即窗口为0）使用**持续计时器** 到时间后发送探测报文段打破零窗口僵局（**对应发送方缓冲区满了或者接收方满了，即任一方的窗口为0**）
  + 拥塞控制算法：**慢开始（指数增加）和拥塞避免（线性增加，拥塞时，慢开始门限减半为发生拥塞时候的窗口值）**，**快重传（收到三个重复确认后立即重传）和快恢复（把慢开始门限减半，但是不执行慢开始而是执行拥塞避免）**，使用快恢复算法时，只有在TCP建立连接和网络超时时使用**慢开始**
* TCP客户端试图与一个**不存在的端口**建立连接

O 服务端的内核响应一个RST发送给客户端表示出错，即使服务端进程没有启动，这是由操作系统帮助完成的。（**TCP头中的RST关键字**）

* TCP客户端试图与一个**不存在的主机上的某端口**建立连接

TCP在等待6秒之后再发送SYN，之后再等24秒发送SYN，总共等待75秒之后仍未收到则返回ETIMEDOUT错误，最好在调用connect()时设置连接超时时间。

* Server进程被阻塞

客户端进程也可以通过这个连接给服务器端发送请求，服务器端TCP会应答ACK表示已经收到这个分节（这里的收到指的是数据已经在内核的缓冲区里准备好，由于进程被阻塞，无法将数据从内核的缓冲区复制到应用程序的缓冲区），但永远不会返回结果。

* TCP半连接队列和完全连接队列

<https://segmentfault.com/a/1190000008224853>

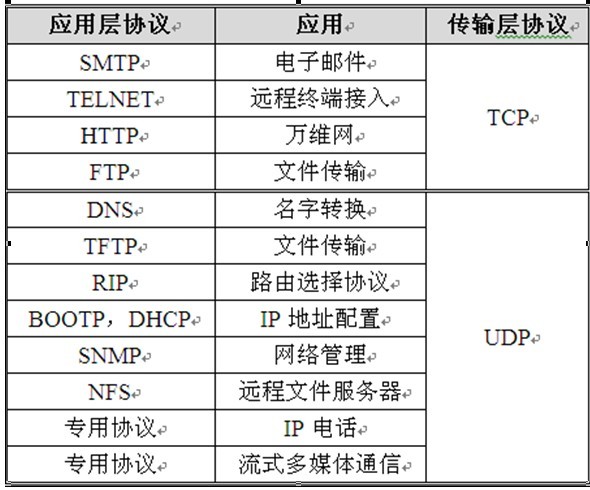
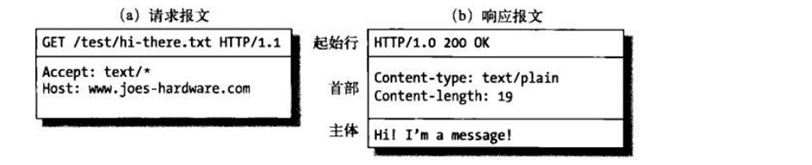
服务器完成套接字创建，绑定端口后，内核中维持了 SYN 和 ACCEPT两个队列。 建立连接时，客户端向服务器发送**SYN包，服务器将其加入到SYN队列中**，并返回 SYN+ACK 包给客户端，一段时间后，客户端发来针对服务器的SYN包的ACK网络分组时，内核从**SYN队列中取出连接，放到ACCEPT队列中**。 而调用accpet()函数时，实际就是从ACCEPT队列中拿出套接字而已。

当syn队列满了时候，如果不启用syncookies，则**服务端会丢弃新来的SYN包**，Client则因得不到响应返回 connection time out

当accept队列满了时候，通过tcp\_abort\_on\_overflow来决定响应，1表示发送RST让Client释放连接之后再重新建立连接，0表示直接丢弃该ACK

**全连接队列的大小取决于：min(backlog, somaxconn) . backlog是在socket创建的时候传入的，somaxconn是一个os级别的系统参数**

**半连接队列的大小取决于：max(64, /proc/sys/net/ipv4/tcp\_max\_syn\_backlog)。 不同版本的os会有些差异**

* TCP与**UDP（User Datagram Protocol）**的区别
  + 连接：TCP**面向连接**；UDP**无连接**，并不能知道对方是否收到、收到的信息是否准确无误
  + 传输形式：**TCP面向字节流**，会把从上层传输下来的数据当作是无结构的字节流，如果上层数据块太长，TCP就会将其划分为多个短块进行发送；**UDP面向报文**，无论应用层交给UDP多长的报文，UDP都照样发送，既不合并也不会拆分，而是保留这些报文的边界，因此使用UDP的应用程序自己必须选择合适大小的报文。
  + UDP支持一对一、一对多和多对一通信，TCP支持一对一通信
  + UDP的首部开销小，只有8个字节，UDP首部：**源端口和目的端口，数据报长度（最小值为8只含首部），校验和；**TCP首部20个字节，TCP首部字段：**源端口和目的端口，序号，确认号（确认号 之前的所有数据都已收到），紧急URG，确认ACK，复位RST，同步SYN，终止FIN，窗口，**
  + TCP有拥塞控制，UDP没有
* TCP与UDP的有哪些应用  
  
* HTTP报文结构 <https://www.cnblogs.com/zhuifeng/p/4072248.html?tdsourcetag=s_pctim_aiomsg>
  + HTTP报文分为请求报文和响应报文，都是由起始行、首部、主体构成的。  
    
  + http头部包括：

通用头部（既可以出现在请求报文中，也可以出现在响应报文中）常用的有：Date，Via，Cache-Control，Connection（逐跳首部、连接的管理）

请求头部：Accept（用户可处理的媒体类型）Host（请求资源所在服务器）Range（实体的字节请求范围）

响应头部：Location（令客户端重定向至指定URI）Age（推算资源创建经过时间）Server（http服务器的安装信息）

实体头部：Allow（资源支持的http方法）Content-Encoding、Length等实体属性，

* HTTP的长连接和短连接?  
  <https://www.cnblogs.com/echo1937/p/6507049.html>
  + HTTP的长连接、短连接本质上是TCP的长连接、短连接。
  + 短连接即客户端与服务端进行完一次读写后就会关闭连接，当又需要数据交互时就会重复三次握手、四次挥手等过程建立一个新的连接。此时存在的连接都是有用的，不需要额外的手段控制。当Web服务器的并发数很高时，使用短连接较好。
  + 长连接即客户端与服务端进行完一次读+++++++写后不会立刻关闭TCP连接，而是保持一段时间从而可以复用该连接，这样可以省去较多的TCP建立和关闭连接的操作，适合频繁请求资源的客户端。
* HTTP与HTTPS的区别以及如何实现安全性  
  <http://www.cnblogs.com/ghjbk/p/6738069.html>
  + HTTPS相当于包裹在SSL协议外壳里的HTTP协议。SSL协议位于TCP/IP协议与各种应用层协议之间，为数据通讯提供安全支持。主要用于解决HTTP由于明文传输消息而导致的第三方窃听问题。
  + **HTTPS连接过程：非对称加密 -> 对称加密**
* 客户端向服务端发送请求
* 服务端返回数字证书
* 客户端用自己的CA[主流的CA机构证书一般都内置在各个主流浏览器中]公钥去解密证书,如果证书有问题会提示风险
* 如果证书没问题客户端会生成一个对称加密的随机秘钥然后再和刚刚解密的服务器端的公钥对数据进行加密,然后发送给服务器端
* 服务器端收到以后会用自己的私钥对客户端发来的对称秘钥进行解密
* 之后双方就拿着这个对称加密秘钥来进行正常的通信如何验证证书的合法性
  + 服务器传来的数字证书是通过可信赖的数字证书签发机构（**CA**）的公钥加密的，浏览器和操作系统都会维护一个CA列表（包括它们的公钥），再通过数字证书上的CA与证书上写的生成**数字签名**编号的方法，去**验证服务器发来的数字签名是否与生成的数字签名相**同，从而确认合法性。
* HTTP1.0与2.0的区别
  + 二进制格式：HTTP2.0没有改变1.x的语义，只是在应用层使用二进制分帧方式传输。
  + 多路复用连接共享：为**每个**request设置一个ID，这样一个TCP连接上就可以有多个request，接收方根据连接的requestID将request归属到不同的服务端请求里进行处理。
  + 首部压缩：维护一个键值对的集合——首部表，新增的键值对会更新到表尾，从而不需要每次通信都必须携带用于描述资源属性的首部。
  + 服务端推送：服务端可以根据客户端的请求，提前返回多个响应，即推送额外的资源给客户端，这样当客户端需要这些资源时就可以在缓存中找到。
* **Get与POST的区别 Amazing**  
  <https://sunshinevvv.coding.me/blog/2017/02/09/HttpGETv.s.POST/>
  + 本质上来说，POST和GET方法是没有区别的，都是TCP连接，只有语义区别。GET方法是用来从服务端获取数据，语义是读。而POST是向服务器提交一些数据，语义是修改服务端的一些数据。
  + 所谓的POST与GET方法的区别，都是浏览器在实现GET与POST方法时的区别，都是现象上的区别。  
    
* TCP的流量控制与拥塞处理  
  <https://www.cnblogs.com/wxgblogs/p/5616829.html>
  + 流量控制是指发送方的速率不能过快，高于接收方的接受速率。拥塞处理是指防止过多的数据注入到网络中，使得路由器或链路过载。
  + 流量控制：滑动窗口流量控制 + 零窗口计时器避免死锁.
  + 拥塞控制**：慢启动、拥塞窗口、快重传与快恢复**
* IP地址的分类
  + ipv6：地址空间更大、路由表更小、更高的安全性（可以对网络层数据加密）
* 有了唯一的Mac地址为啥还需要IP地址？
  + 方便路由、避免维护极其庞大的MAC地址表
* 交换机、集线器与路由器、网桥、网关有什么区别？  
  <https://www.tianmaying.com/tutorial/NetWorkInstrument>

路由器与网关的区别：前者连接的是相似的网络，而后者连接的是不相似的网络，如连接公网与私网。

* URI和URL的区别
  + URL是URI的子集。所以不论是用定位的方式还是用编号的方式，我们都可以唯一确定一个人，都是URl的一种实现，而URL就是用定位的方式实现的URI。而URL则通过描述是哪个主机上哪个路径上的文件来唯一确定一个资源，也就是定位的方式来实现的URI。对于现在网址我更倾向于叫它URL，毕竟它提供了资源的位置信息，如果有一天网址通过号码来标识变成了http://741236985.html，那感觉叫成URI更为合适，不过这样子的话还得想办法找到这个资源咯…
* GET请求中URL编码的意义：避免服务端解析请求时出现歧义
* 常见状态码及原因短语  
  <https://blog.csdn.net/q1056843325/article/details/53147180>
  + 
  + 200 OK
  + 301 Moved Permanently（资源永久移动）、302 Found（资源临时移动）、304 Not Modified（没有满足条件的资源，并非重定向）
  + 400 Bad Request（请求报文语法或参数错误）、401 Unauthorized（需要认证或认证失败）、403 Forbidden（拒绝访问）、404 Not Found（找不到资源）
  + 500 Internal Server Error（Web应用Bug或临时故障）、503 Service Unavailable（服务器超载或停机维护）
* 说说Session、Cookie
  + 因为http是无状态的，服务器不知道用户上次坐了什么，为了辨别用户身份、购物车功能、记录浏览数据等作用而出现的
  + 共同点：作用都是为了保存、跟踪用户的会话。
  + Cookie是服务器发送给客户端的，保存在客户端。当浏览器再次请求服务端时，就会把请求的网址连同Cookie一起提交给服务器，服务器检查该cookie来确认用户状态，并且可以根据需要修改Cookie内容。
  + Session存储在服务端。服务器端为每个用户创建Session ID（唯一的、不容易找到规律的字符串），session ID在响应中返回到客户端保存，当浏览器禁止使用cookie的时候，可以使用**URL重写**，就是把Session id直接附加在URL路径的后面一种是作为URL路径的附加信息, 另一种是作为**查询字符串附加在URL后面。**
  + 不同点：cookie以文本格式存储在浏览器，储量有限；session存储在服务端，可以无限量存储多个变量且更加安全。

有效期不同：只需要设置Cookie的过期时间属性为一个很大很大的数字，Cookie就可以在浏览器保存很长时间。 由于Session依赖于名为JSESSIONID的Cookie，而Cookie JSESSIONID的过期时间默许为–1，只需关闭了浏览器（一次会话结束），该Session就会失效。

对服务器造成的压力不同

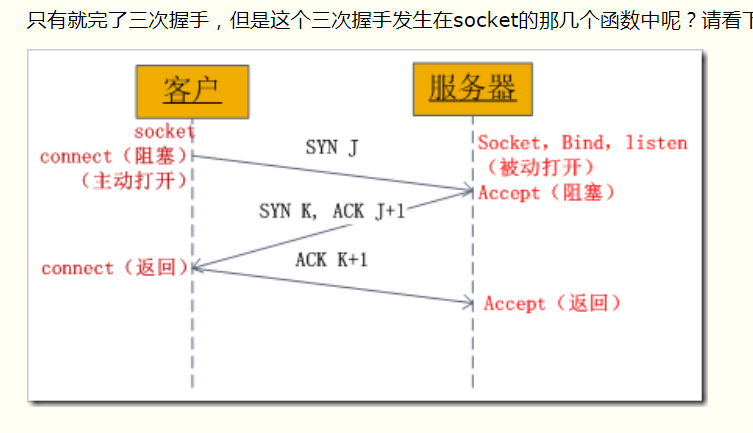
Session是保管在服务器端的，每个用户都会产生一个Session。假如并发访问的用户十分多，会产生十分多的Session，耗费大量的内存。而Cookie保管在客户端，不占用服务器资源。假如并发阅读的用户十分多，Cookie是很好的选择

* 如**何避免浏览器缓存**
  + 在服务端响应首部加入Expires设置浏览器的缓存有效性；
  + 在服务端响应首部中加入Cache-Control=no-cache避免缓存
* 什么是分块传送。
  + 首先响应首部中的Content-Length对客户端非常重要，客户端通过这个字段来判断哪个位置是响应消息的结束，因为对于持久连接TCP并不会在请求完立马结束，而是可以发送多次请求/响应。
  + 对于复杂页面来说，等到消息体完全创建好之后再计算Content-Length传给客户端的话，等待时间太久了。
  + 因此就提出了分块传输编码，把数据分解成一系列数据块，并以块的形式发送给客户端，这样服务端就不需要预先告诉客户端发送内容的总大小了。
* 谈谈SQL 注入
  + 所谓SQL注入，就是通过把SQL命令插入到Web表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。
  + 预防：对用户的输入进行校验，对引号等符号进行转换；为每个应用使用单独的权限有限的数据库连接，不要使用管理员权限的数据库连接；机密信息不要明文存放；异常信息中应该给出尽可能少的提示，最好使用自定义的错误信息对原始错误信息进行包装。
* DDos攻击有那些预防方法？
* 什么是XSS 攻击
  + 跨站脚本攻击，攻击者向Web页面里插入而已的Script代码，当用于浏览时，嵌入其中的Script代码被执行，从而达到恶意攻击的目的。 根源也是由于服务端没有过滤好客户端提交的数据。
* **Socket网络通信 java**

ServerSocket serverSocket = new ServerSocket(int port, int backlog);  ServerSocket作用是向操作系统注册相应协议服务，申请端口并监听这个端口是否有链接请求。其中port是端口号，backlog是服务器最多允许链接的客户端数。注册完成后，服务器分配此端口用于提供某一项进程服务

服务器端Socket:  Socket socket = serverSocket.accept();  服务器端创建一个socket对象用于等待客户端socket的链接（accept方法是创建一个阻塞队列，只有客户端socket申请链接到服务器后，服务器端socket才能收到消息） 。如果服务器端socket收到客户端的链接请求，那么经过"三次握手"过程，建立客户端与服务器端的连接。

客户端Socket： Socket socket = new Socket(String host, int port);  客户端创建按一个socket对象用于链接具体服务器host的具体服务端口port，用于获得服务器进程的相应服务



**设计模式：**本质目的是解耦，和建立在此基础上的可扩展性和强健性

* **单例模式：**[**http://blog.jobbole.com/109449/**](http://blog.jobbole.com/109449/)
  + **作用：**保证系统中只有一个实例，避免全局使用的类被频繁地创建和销毁，从而节省系统资源，如I/O和数据库的连接等。
  + **构造函数私有、声明静态的单例对象**。 保证单例类不能被外部实例化，而只能静态引用(getInstance())。
  + **构造单例对象之前要加对象锁**（lock一个静态的object对象；也可以使用synchronized(Singleton.class)，关键是保证是引用类型即可）。锁住的必须是引用类型而不可以是值类型（如int类型变量），因为每个线程在声明值类型变量时的地址都是不一样的，上个线程锁住的东西下个线程进来会认为根本没锁。
  + **两次检查：**在lock前检查单例实例是否已经被构造、在lock后再次检查单例实例是否已经被构造。

public class Singleton {  
 private static Singleton ***instance***;  
 private Singleton() {// 初始化单例对象的一些信息 }  
  
 public Singleton getInstance() {  
 if (***instance*** == null) {  
 // 使用Singleton.class的对象锁  
 synchronized (Singleton.class) {  
 if (***instance*** == null) {  
 ***instance*** = new Singleton();  
 }}}  
 return ***instance***; }}

* **简单/静态工厂模式：抽象产品**(定义一个公共的接口)、**具体产品**(多个实现了该接口的类)和**工厂**(以及一个用来实例化具体产品的工厂类)**。**
  + **作用：**将对象的创建与对象的使用分离开，用户类只需要传入一个正确的参数就可以获得需要的对象，而不用关心对象的实现细节**。**
  + **缺点：**工厂类不够灵活，想要添加新的具体产品需要修改工厂类的判断逻辑代码，产品较多时工厂方法的代码就会特别复杂**。**

/\*\*  
 \* 定义一个公共的接口或抽象类  
 \*/  
abstract class Car{  
 abstract public void run();  
 abstract public void park();  
}  
/\*\*  
 \* 接口的具体实现  
 \*/  
class Benz extends Car{ // 具体的实现}  
  
class Ford extends Car{ // 具体的实现}  
  
public class StaticFactory {

// 工厂判断传入参数，生成对应类的实例  
 public static Car getCar(String *type*) {  
 if (*type*.equals("Benz")) return new Benz();  
 if (*type*.equals("Ford")) return new Ford();  
 return null;  
 }  
 // 使用工厂获得需要的类的实例，而不需要关心类的实例化细节  
 public static void main(String[] *args*) {  
 Car *aCar* = StaticFactory.*getCar*("BMW");  
 if (*aCar* == null) {  
 System.***out***.println("No such car");  
 } else {  
 *aCar*.run();  
 *aCar*.park();  
 }  
 }  
}

* **工厂方法模式：抽象工厂 + 具体工厂 + 抽象产品 + 具体产品。** 不再像简单工厂模式那样由一个工厂类实例化具体产品，而是使用抽象工厂的子类去实例化产品**。**
  + 抽象工厂定义一个公共的抽象方法，具体工厂来实现这些方法，以返回实现了某一个抽象产品接口的不同的具体产品。
  + **特点**：**具体产品和具体工厂一一对应**。
  + **优点：**相比于简单/静态工厂模式，工厂方法模式利用多态特性，在需要添加新的产品类的时候，不需要对现有的系统进行修改，而只需要增加产品类以及对应的具体工厂类的代码。
  + **缺点**在于由于产品和工厂是一一对应的，导致系统类的个数成对增加，增加了复杂性**。**

//抽象产品接口

interface Movable{ public void move(); }  
  
class Plane implements Movable { //具体产品的实现 }   
class Vehicle implements Movable { //具体产品的实现 }  
class Person implements Movable { //具体产品的实现 }  
// 抽象工厂接口  
abstract class MovableItemFactory{  
 abstract public Movable getMovableItem();  
}  
// 具体工厂的实现  
class PlaneFactory extends MovableItemFactory {  
 @Override  
 public Movable getMovableItem() {return new Plane();}  
}  
class VehicleFactory extends MovableItemFactory { //具体工厂的实现 }  
class PersonFactory extends MovableItemFactory { //具体工厂的实现 }  
  
public class FactoryMethodMode {  
 public static void main(String[] *args*) {

//需要具体产品时，需要先获得对应的具体工厂对象  
 **MovableItemFactory** *personFactory* = new PersonFactory();  
  **Movable** *p* = *personFactory*.getMovableItem();  
 *p*.move();  
 }}

* **抽象工厂模式：围绕一个超级工厂创建其他工厂，再创建具体产品**
  + **特点：**与工厂方法模式不同之处在于，抽象工厂模式的一个工厂生产**多个产品**（前者是一一对应的）。抽象工厂定义生成包含多个产品（即一个产品族）的多个方法，具体的工厂也根据自己的产品族去生成对应的产品。
  + 缺点：当要删除、修改或添加一个产品时，需要对抽象工厂和所有具体工厂类进行改动。

interface AbstractFactory {

// 抽象工厂接口定义  
 public Product1 getProduct1();  
 public Product2 getProduct2();  
}  
abstract class Product1 {  
 abstract void product1\_work();  
}  
abstract class Product2 {  
 abstract void product2\_work();  
}  
class LenovoFactory implements AbstractFactory {  
 @Override  
 public Product1 getProduct1() {  
 return new Lenovo\_Product1();  
 }  
 @Override  
 public Product2 getProduct2() {  
 return new Lenovo\_Product2();  
 }  
}  
class DellFactory implements AbstractFactory { // 具体的实现}  
class Dell\_Product1 extends Product1{ // 具体的实现}  
class Dell\_Product2 extends Product2{// 具体的实现}  
class Lenovo\_Product1 extends Product1{// 具体的实现}  
class Lenovo\_Product2 extends Product2{// 具体的实现}  
  
public class AbstractFactoryPattern{  
 public static void main(String[] *args*) {

// 获取某个品牌（产品族）的具体工厂，生成对应的产品族  
 AbstractFactory ***factory***= new LenovoFactory();  
 Product1 *p1* = ***factory***.getProduct1();  
 Product2 *p2* = ***factory***.getProduct2();  
 *p1*.product1\_work();  
 *p2*.product2\_work();}}

* **装饰器模式**
  + 装饰器模式又称为包装（Wrapper）模式。装饰器模式以对客户端透明的方式扩展对象的功能，是**继承关系的一个替代方案**。扩展对象的方式是，新建一个装饰器类，与被装饰类实现相同的接口，该类持有被装饰类的引用，就可以在调用被装饰类的方法前后增加一些自定义的方法。这样，在使用被装饰类的方法时，并不是调用被装饰类对象的方法，而是调用装饰器的方法。
* **代理模式&与装饰器模式的区别**
  + 代理类与被代理类同样是要实现相同的接口，代理对象也持有被代理对象的引用。
  + 代理与装饰器模式实现机制是一样的，但是就语义来说，二者的功能是相反的。**装饰器模式关注于扩展/增强对象的功能，然而代理模式关注于控制对对象的访问**。换句话说，用代理模式，代理类（proxy class）可以对它的客户隐藏一个对象的具体信息。因此，**当使用代理模式的时候，我们常常在一个代理类中创建一个对象的实例**。并且，**当我们使用装饰器模式的时候，我们通常的做法是将原始对象作为一个参数传给装饰者的构造器**。
  + 代理模式的应用比较广泛，如Spring的AoP就是代理和反射实现的。另外还有延迟加载，如果当前没有使用这个组件时，则不需要真正地去初始化它，而是用一个代理对象去替代它的原有位置。当真正需要使用的时候，才对它进行加载。装饰器的应用相对没有那么广泛，在Java IO包中大量应用了装饰器，用以实现不同输入输出类的功能的组合。

**开放性问题**

* [**你的系统如何支撑高并发？**](https://www.jianshu.com/p/121fa3cd1d15)
  + **保证业务逻辑的合理性，避免资源浪费**
  + **系统集群化部署&负载均衡**
  + **数据库分库分表+读写分离，但是读写分离与数据一致性相冲突**
  + **数据库缓存处理高并发读，保存容易被发起读请求的数据，先查缓存，缓存中没有再去读数据库**
  + **消息中间件 削峰填谷处理高并发写：如消息队列集群，把允许异步化的写请求放入消息队列，先把必须同步处理的请求处理掉**
* [**高并发秒杀系统如何设计与优化**](https://blog.csdn.net/zuochao_2013/article/details/82458496)

**Linux**

<https://mp.weixin.qq.com/s?__biz=Mzg2OTA0Njk0OA==&mid=2247484915&amp;idx=1&amp;sn=ee4ca75866b5b00ee68c31307ada59bd&source=41#wechat_redirect>

### spring

1. **spring概述**

* 使用Spring框架的好处是什么？
* **Spring的加载过程**：
  + **创建Spring容器（Spring上下文）**用来加载bean：轻量级BeanFactory接口或更常用更强大的ApplicationContext接口，即Spring的上下文。ApplicationContext接口的常用实现：FileSystemXmlApplicationContext（使用绝对路径）、  
    **ClassPathXmlApplicationContext**（相对路径）或者**WebXmlApplicationContext**（在一个web应用程序的范围内加载xml文件中定义的bean）
  + 容器创建后，会**自动加载**xml文件中描述的**JavaBean的类实例**。加载顺序：bean的构造器方法（按照xml中的<constructor-arg>来决定调用什么参数的构造器方法） Bean的setter方法(按照xml中的<property>) <init-method>相关——如果xml中设置了<bean init-method=”abc()”>，则会在调用init-method方法**前后**分别调用BeanPostProcessor的两个方法
  + **执行业务代码**：可以用Context.getBean()方法获得所需的JavaBean对象进行业务操作。
  + **应用结束，调用JavaBean的destroy-method（如果设置了的话）。**
* **Spring + Struts Web的加载顺序**：
  + 服务器（Tomcat）启动时，读取web.xml(包括Struts的过滤器配置、spring容器的配置、spring容器监听器)。
  + 按照xml中配置，先调用Spring容器监听器ContextLoaderListener的无参构造方法生成该对象，监听器会监听ServletContext的后续启动
  + tomcat容器创建一个ServletContext，该事件会触发监听器的initWebApplicationContext()方法，生成Spring的WebApplicationContext，即Spring的容器。然后进行Spring容器的一系列加载，如一些单例的JavaBean。然后，监听器会将WebApplicationContext注入到ServletContext中，作为其一个属性，从而这两个上下文对象都可以互相引用得到。
  + StrutsPrepareAndExecuteFilter.init()方法，初始化Struts的过滤器，对Http请求和相应进行处理。
* 解释AOP模块
  + 面向切面编程需要把程序的逻辑分解成为不同的部分，即关注点。跨一个应用程序多个点的功能成为横切关注点，**这些横切关注点在概念上独立于应用程序的业务逻辑**。AOP可以使对横切关注点进行解耦，如**日志记录**、审计、安全性和缓存等。在AOP中如果想对一个方面的内容进行修改，不需要在每一个相关的类的源文件中修改很多内容，而是在XML配置或者在JavaBean中进行相关的AOP声明。

**2. spring依赖注入**

* IOC的优点是什么：解耦，避免牵一发而动全身。
* 什么是Spring的依赖注入？
* 有哪些不同类型的IOC（依赖注入）方式？
  + 基于构造器：容器调用Bean带有多个参数的构造器方法，其传入参数就是容器提供的依赖注入。
  + 基于Setter方法：在调用无参构造器方法或无参静态工厂方法实例化bean之后，容器调用bean的setter方法，传入参数是容器提供的依赖注入。
* 什么是Spring beans?
  + 就是按照一定的约束/规范编写的Java类。例如，设计到业务的属性设计为private，然后编写对应的**setter**/getter方法去操作。
* 一个 Spring Bean 定义 包含什么？
* 解释Spring支持的几种bean的作用域。
* Spring框架中的单例bean是线程安全的吗?
* 解释Spring框架中bean的生命周期
* 哪些是重要的bean生命周期方法？ 你能重载它们吗？
* 什么是bean装配?
* 什么是bean的自动装配？
* 解释不同方式的自动装配 。
* 自动装配有哪些局限性 ?

**3. spring 注解**

* 怎样开启注解装配？
* 谈谈@Required、 @Autowired、 @Qualifier注解。

**4， spring 数据访问**

* 在Spring框架中如何更有效地使用JDBC?
* 使用Spring通过什么方式访问Hibernate?
* Spring框架的事务管理有哪些优点？

**5. Spring面向切面编程（AOP）**

* 解释AOP
* Aspect 切面
* 在Spring AOP 中，关注点和横切关注的区别是什么？
* 通知
* 有几种不同类型的自动代理？
* 什么是织入。什么是织入应用的不同点？

**6. springMVC**

* 什么是Spring的MVC框架？
* DispatcherServlet
* WebApplicationContext
* 什么是Spring MVC框架的控制器？
* @Controller 注解
* @RequestMapping 注解

### JavaWeb

**servlet与Tomcat**

* Servlet生命周期
* forward和redirect的区别
* tomcat容器是如何创建servlet类实例？用到了什么原理？
* 什么是cookie？Session和cookie有什么区别？
* Servlet安全性问题
* Tomcat 有哪几种Connector 运行模式(优化)？
* \*

**JSP**

* jsp静态包含和动态包含的区别
* jsp有哪些内置对象?作用分别是什么?
* jsp和servlet的区别、共同点、各自应用的范围？
* 写出5种JSTL常用标签
* JSP是如何被执行的？执行效率比SERVLET低吗？
* 说出Servlet和CGI的区别？
* 简述JSP的设计模式。