Java面试手册

书栈(BookStack.CN)

目　录

致谢

当前文档 《Java面试手册》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-10-05。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：http://www.bookstack.cn/books/java_interview_manual

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

介绍

Java面试手册

《Java面试手册》整理了从业到现在看到的、经历过的一些Java面试题。

这些面试题的主要来源是一些网站还有github上的内容，由于平常在收藏到“印象笔记”中的时候没有保留来源出处，如果有介意版权的可以联系我。

github地址为： `https://github.com/guanzhenxing/java_interview_manual`

主要分为以下部分：

    - 计算机网络
    - Nginx
    - Tomcat
    - Netty
- 软件工程
    - UML
- 业务
- 操作系统

主要参考：

- Java面试通关要点汇总集【终极版】
- 《后端架构师技术图谱》
- `https://github.com/hadyang/interview`
- `https://github.com/crossoverJie/Java-Interview`
- 后台开发常问面试题集锦

Java基础

Java基础

面向对象

面向对象

什么是对象

对象是系统中用来描述客观事物的一个实体，它是构成系统的一个基本单位。一个对象由一组属性和对这组属性进行操作的一组服务组成。

类的实例化可生成对象，一个对象的生命周期包括三个阶段：生成、使用、消除。

当不存在对一个对象的引用时，该对象成为一个无用对象。Java的垃圾收集器自动扫描对象的动态内存区，把没有引用的对象作为垃圾收集起来并释放。当系统内存用尽或调用System.gc()要求垃圾回收时，垃圾回收程与系统同步运行。

面向对象的特征

封装，继承和多态。

- 封装：面向对象最基础的一个特性，封装性，是指隐藏对象的属性和现实细节，仅对外提供公共访问方式。

> 封装的原则：将不需要对外提供的内容都隐藏（设置访问修饰符为*“private”*）起来。把属性都隐藏，仅提供公共方法对其访问，可以在访问方式中加入逻辑判断等语句。

- 继承：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。
- 多态：多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。

> 多态性分为编译时的多态性和运行时的多态性。
>
> 运行时的多态是面向对象最精髓的东西，要实现多态需要做两件事：
>
> *1).* 方法重写（子类继承父类并重写父类中已有的或抽象的方法）
>
> *2).* 对象造型（用父类型引用引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

什么是类

类是具有相同属性和方法的一组对象的集合，它为属于该类的所有对象提供了统一的抽象描述，其内部包括属性和方法两个主要部分。在面向对象的编程语言中，类是一个独立的程序单位，它应该有一个类名并包括属性和方法两个主要部分。

Java中的类实现包括两个部分：类声明和类体。

多态的好处

多态的定义：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。

主要有以下优点：

- 可替换性:多态对已存在代码具有可替换性.
- 可扩充性:增加新的子类不影响已经存在的类结构.
- 接口性:多态是超类通过方法签名,向子类提供一个公共接口,由子类来完善或者重写它来实现的.
- 灵活性:它在应用中体现了灵活多样的操作，提高了使用效率
- 简化性:多态简化对应用软件的代码编写和修改过程，尤其在处理大量对象的运算和操作时，这个特点尤为突出和重要

代码中如何实现多态

实现多态主要有以下三种方式：

- 接口实现
- 继承父类重写方法
- 同一类中进行方法重载

虚拟机是如何实现多态的

动态绑定技术(dynamic binding),执行期间判断所引用对象的实际类型,根据实际类型调用对应的方法.

重载（Overload）和重写（Override）的区别。重载的方法能否根据返回类型进行区分？

方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态性，而后者实现的是运行时的多态性。

- 重载发生在一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同或者二者都不同）则视为重载；

- 重写发生在子类与父类之间，重写要求子类被重写方法与父类被重写方法有相同的返回类型，比父类被重写方法更好访问，不能比父类被重写方法声明更多的异常（里氏代换原则）。重载对返回类型没有特殊的要求。

构造器不能被继承，因此不能被重写，但可以被重载。

父类的静态方法不能被子类重写。重写只适用于实例方法，不能用于静态方法，而子类当中含有和父类相同签名的静态方法，我们一般称之为隐藏，调用的方法为定义的类所有的静态方法。

构造器（constructor）是否可被重写（override）？

构造器不能被继承，因此不能被重写，但可以被重载。

接口的意义

接口的意义用四个词就可以概括：规范，扩展，回调和安全。

抽象类的意义

抽象类的意义可以用三句话来概括：

- 为其他子类提供一个公共的类型
- 封装子类中重复定义的内容
- 定义抽象方法,子类虽然有不同的实现,但是定义是一致的

抽象类和接口有什么区别

抽象类和接口都不能够实例化，但可以定义抽象类和接口类型的引用。一个类如果继承了某个抽象类或者实现了某个接口都需要对其中的抽象方法全部进行实现，否则该类仍然需要被声明为抽象类。接口比抽象类更加抽象，因为抽象类中可以定义构造器，可以有抽象方法和具体方法，而接口中不能定义构造器而且其中的方法全部都是抽象方法。抽象类中的成员可以是private、默认、protected、public的，而接口中的成员全都是public的。抽象类中可以定义成员变量，而接口中定义的成员变量实际上都是常量。有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法。

访问修饰符public,private,protected,以及不写（默认）时的区别

修饰符	当前类	同包	子类	其他包
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

类的成员不写访问修饰时默认为default。默认对于同一个包中的其他类相当于公开（public），对于不是同一个包中的其他类相当于私有（private）。受保护（protected）对子类相当于公开，对不是同一包中的没有父子关系的类相当于私有。Java中，外部类的修饰符只能是public或默认，类的成员（包括内部类）的修饰符可以是以上四种。

简述一下面向对象的"六原则一法则"。

- 单一职责原则：一个类只做它该做的事情。

单一职责原则想表达的就是"高内聚"，写代码最终极的原则只有六个字"高内聚、低耦合"。所谓的高内聚就是一个代码模块只完成一项功能，在面向对象中，如果只让一个类完成它该做的事，而不涉及与它无关的领域就是践行了高内聚的原则，这个类就只有单一职责。我们都知道一句话叫"因为专注，所以专业"，一个对象如果承担太多的职责，那么注定它什么都做不好。一个好的软件系统，它里面的每个功能模块也应该是可以轻易的拿到其他系统中使用的，这

样才能实现软件复用的目标。

- 开闭原则：软件实体应当对扩展开放，对修改关闭。

在理想的状态下，当我们需要为一个软件系统增加新功能时，只需要从原来的系统派生出一些新类就可以，不需要修改原来的任何一行代码。要做到开闭有两个要点：

1）抽象是关键，一个系统中如果没有抽象类或接口系统就没有扩展点；

2）封装可变性，将系统中的各种可变因素封装到一个继承结构中，如果多个可变因素混杂在一起，系统将变得复杂而换乱，如果不清楚如何封装可变性，可以参考《设计模式精解》一书中对桥梁模式的讲解的章节。

- 依赖倒转原则：面向接口编程。

该原则说得直白和具体一些就是声明方法的参数类型、方法的返回类型、变量的引用类型时，尽可能使用抽象类型而不用具体类型，因为抽象类型可以被它的任何一个子类型所替代，请参考下面的里氏替换原则。

- 里氏替换原则：任何时候都可以用子类型替换掉父类型。

关于里氏替换原则的描述，Barbara Liskov女士的描述比这个要复杂得多，但简单的说就是能用父类型的地方就一定能使用子类型。里氏替换原则可以检查继承关系是否合理，如果一个继承关系违背了里氏替换原则，那么这个继承关系一定是错误的，需要对代码进行重构。例如让猫继承狗，或者狗继承猫，又或者让正方形继承长方形都是错误的继承关系，因为你很容易找到违反里氏替换原则的场景。需要注意的是：子类一定是增加父类的能力而不是减少父类的能力，因为子类比父类的能力更多，把能力多的对象当成能力少的对象来用当然没有任何问题。

- 接口隔离原则：接口要小而专，绝不能大而全。

臃肿的接口是对接口的污染，既然接口表示能力，那么一个接口只应该描述一种能力，接口也应该是高度内聚的。例如，琴棋书画就应该分别设计为四个接口，而不应设计成一个接口中的四个方法，因为如果设计成一个接口中的四个方法，那么这个接口很难用，毕竟琴棋书画四样都精通的人还是少数，而如果设计成四个接口，会几项就实现几个接口，这样的话每个接口被复用的可能性是很高的。Java中的接口代表能力、代表约定、代表角色，能否正确的使用接口一定是编程水平高低的重要标识。

- 合成聚合复用原则：优先使用聚合或合成关系复用代码。

通过继承来复用代码是面向对象程序设计中被滥用得最多的东西，因为所有的教科书都无一例外的对继承进行了鼓吹从而误导了初学者，类与类之间简单的说有三种关系，Is-A关系、Has-A关系、Use-A关系，分别代表继承、关联和依赖。其中，关联关系根据其关联的强度又可以进一步划分为关联、聚合和合成，但说白了都是Has-A关系，合成聚合复用原则想表达的是优先考虑Has-A关系而不是Is-A关系复用代码，原因嘛可以自己从百度上找到一万个理由，需要说明的是，即使在Java的API中也有不少滥用继承的例子，例如Properties类继承了Hashtable类，Stack类继承了Vector类，这些继承明显就是错误的，更好的做法是在Properties类中放置一个Hashtable类型的成员并且将其键和值都设置为字符串来存储数据，而Stack类的设计也应该是在Stack类中放一个Vector对象来存储数据。记住：任何时候都不要继承工具类，工具是可以拥有并可以使用的，而不是拿来继承的。

- 迪米特法则：迪米特法则又叫最少知识原则，一个对象应当对其他对象有尽可能少的了解。

迪米特法则简单的说就是如何做到”低耦合”，门面模式和调停者模式就是对迪米特法则的践行。对于门面模式可以举一个简单的例子，你去一家公司洽谈业务，你不需要了解这个公司内部是如何运作的，你甚至可以对这个公司一无所知，去的时候只需要找到公司入口处的前台美女，告诉她们你要做什么，她们会找到合适的人跟你接洽，前台的美女就是公司这个系统的门面。再复杂的系统都可以为用户提供一个简单的门面，Java Web开发中作为前端控制器的

Servlet或Filter不就是一个门面吗，浏览器对服务器的运作方式一无所知，但是通过前端控制器就能够根据你的请求得到相应的服务。调停者模式也可以举一个简单的例子来说明，例如一台计算机，CPU、内存、硬盘、显卡、声卡各种设备需要相互配合才能很好的工作，但是如果这些东西都直接连接到一起，计算机的布线将异常复杂，在这种情况下，主板作为一个调停者的身份出现，它将各个设备连接在一起而不需要每个设备之间直接交换数据，这样就减小了系统的耦合度和复杂度。

基础

JAVA基本

String 是最基本的数据类型吗？

不是。Java中的基本数据类型只有8个：byte、short、int、long、float、double、char、boolean；除了基本类型（primitive type）和枚举类型（enumeration type），剩下的都是引用类型（reference type）。

float f=3.4;是否正确？

不正确。3.4是双精度数，将双精度型（double）赋值给浮点型（float）属于下转型（down-casting，也称为窄化）会造成精度损失，因此需要强制类型转换float f =(float)3.4; 或者写成float f =3.4F;。

short s1 = 1; s1 = s1 + 1;有错吗?short s1 = 1; s1 += 1;有错吗？

对于short s1 = 1; s1 = s1 + 1;由于1是int类型，因此s1+1运算结果也是int 型，需要强制转换类型才能赋值给short型。而short s1 = 1; s1 += 1;可以正确编译，因为s1+= 1;相当于s1 = (short)(s1 + 1);其中有隐含的强制类型转换。

int和Integer有什么区别？

Java是一个近乎纯洁的面向对象编程语言，但是为了编程的方便还是引入了基本数据类型，但是为了能够将这些基本数据类型当成对象操作，Java为每一个基本数据类型都引入了对应的包装类型（wrapper class），int的包装类就是Integer，从Java 5开始引入了自动装箱/拆箱机制，使得二者可以相互转换。

Java 为每个原始类型提供了包装类型：

- 原始类型: boolean，char，byte，short，int，long，float，double
- 包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double

&和&&的区别？

&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。逻辑与跟短路与的差别是非常巨大的，虽然二者都要求运算符左右两端的布尔值都是true整个表达式的值才是true。&&之所以称为短路运算是因为，如果&&左边的表达式的值是false，右边的表达式会被直接短路掉，不会进行运算。很多时候我们可能都需要用&&而不是&，例如在验证用户登录时判定用户名不是null而且不是空字符串，应当写为：username != null &&!username.equals(“”)，二者的顺序不能交换，更不能用&运算符，因为第一个条件如果不成立，根本不能进行字符串的equals比较，否则会产生NullPointerException异常。注意：逻辑或运算符（|）和短路或运算符（||）的差别也是如此。

Math.round(11.5) 等于多少？Math.round(-11.5)等于多少？

Math.round(11.5)的返回值是12，Math.round(-11.5)的返回值是-11。四舍五入的原理是在参数上加0.5然后进行下取整。

switch 是否能作用在byte 上，是否能作用在long,float上，是否能作用在String上？

在Java 5以前，switch(expr)中，expr只能是byte、short、char、int。从Java 5开始，Java中引入了枚举类型，expr也可以是enum类型，从Java 7开始，expr还可以是字符串（String），但是长整型（long），浮点数（float）在目前所有的版本中都是不可以的。

两个对象值相同(x.equals(y) == true)，但却可有不同的hash code，这句话对不对？

不对，如果两个对象x和y满足x.equals(y) == true，它们的哈希码（hash code）应当相同。

Java对于eqauls方法和hashCode方法是这样规定的:

(1)如果两个对象相同（equals方法返回true），那么它们的hashCode值一定要相同;

(2)如果两个对象的hashCode相同，它们并不一定相同。当然，你未必要按照要求去做，但是如果你违背了上述原则就会发现在使用容器时，相同的对象可以出现在Set集合中，同时增加新元素的效率会大大下降（对于使用哈希存储的系统，如果哈希码频繁的冲突将会造成存取性能急剧下降）。

补充：关于equals和hashCode方法，很多Java程序都知道，但很多人也就是仅仅知道而已，在Joshua Bloch的大作《Effective Java》（很多软件公司，《Effective Java》、《Java编程思想》以及《重构：改善既有代码质量》是Java程序员必看书籍，如果你还没看过，那就赶紧去亚马逊买一本吧）中是这样介绍equals方法的：首先equals方法必须满足自反性（x.equals(x)必须返回true）、对称性（x.equals(y)返回true时，y.equals(x)也必须返回true）、传递性（x.equals(y)和y.equals(z)都返回true时，x.equals(z)也必须返回true）和一致性（当x和y引用的对象信息没有被修改时，多次调用x.equals(y)应该得到同样的返回值），而且对于任何非null值的引用x，x.equals(null)必须返回false。

实现高质量的equals方法的诀窍包括:

1. 使用==操作符检查”参数是否为这个对象的引用”;
2. 使用instanceof操作符检查”参数是否为正确的类型”;
3. 对于类中的关键属性，检查参数传入对象的属性是否与之相匹配;
4. 编写完equals方法后，问自己它是否满足对称性、传递性、一致性;
5. 重写equals时总是要重写hashCode;
6. 不要将equals方法参数中的Object对象替换为其他的类型，在重写时不要忘掉@Override注解。

当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递

还是引用传递？

是值传递。Java语言的方法调用只支持参数的值传递。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的属性可以在被调用过程中被改变，但对对象引用的改变是不会影响到调用者的。C++和C#中可以通过传引用或传输出参数来改变传入的参数的值。

String和StringBuilder、StringBuffer的区别？

Java平台提供了两种类型的字符串：String和StringBuffer/StringBuilder，它们可以储存和操作字符串。其中String是只读字符串，也就意味着String引用的字符串内容是不能被改变的。而StringBuffer/StringBuilder类表示的字符串对象可以直接进行修改。StringBuilder是Java 5中引入的，它和StringBuffer的方法完全相同，区别在于它是在单线程环境下使用的，因为它的所有方面都没有被synchronized修饰，因此它的效率也比StringBuffer要高。

抽象的（abstract）方法是否可同时是静态的（static），是否可同时是本地方法（native），是否可同时被synchronized修饰？

都不能。抽象方法需要子类重写，而静态的方法是无法被重写的，因此二者是矛盾的。本地方法是由本地代码（如C代码）实现的方法，而抽象方法是没有实现的，也是矛盾的。synchronized和方法的实现细节有关，抽象方法不涉及实现细节，因此也是相互矛盾的。

阐述静态变量和实例变量的区别。

静态变量是被static修饰符修饰的变量，也称为类变量，它属于类，不属于类的任何一个对象，一个类不管创建多少个对象，静态变量在内存中有且仅有一个拷贝；实例变量必须依存于某一实例，需要先创建对象然后通过对象才能访问到它。静态变量可以实现让多个对象共享内存。

补充：在Java开发中，上下文类和工具类中通常会有大量的静态成员。

Object中有哪些公共方法？

- equals()
- clone()
- getClass()
- notify(),notifyAll(),wait()
- toString()

是否可以从一个静态（static）方法内部发出对非静态（non-static）方法的调用？

不可以，静态方法只能访问静态成员，因为非静态方法的调用要先创建对象，在调用静态方法时可能对象并没有被初

始化。

深拷贝和浅拷贝的区别是什么？

浅拷贝：被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用仍然指向原来的对象。换言之，浅拷贝仅仅复制所考虑的对象，而不复制它所引用的对象。

深拷贝：被复制对象的所有变量都含有与原来的对象相同的值，而那些引用其他对象的变量将指向被复制过的新对象，而不再是原有的那些被引用的对象。换言之，深拷贝把要复制的对象所引用的对象都复制了一遍。

如何实现对象克隆？

有两种方式：

1. 实现Cloneable接口并重写Object类中的clone()方法；
2. 实现Serializable接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆。

代码如下：

```
1. import java.io.ByteArrayInputStream;
2. import java.io.ByteArrayOutputStream;
3. import java.io.ObjectInputStream;
4. import java.io.ObjectOutputStream;
5.
6. public class MyUtil {
7.
8.     private MyUtil() {
9.         throw new AssertionError();
10.    }
11.
12.    public static <T> T clone(T obj) throws Exception {
13.        ByteArrayOutputStream bout = new ByteArrayOutputStream();
14.        ObjectOutputStream oos = new ObjectOutputStream(bout);
15.        oos.writeObject(obj);
16.
17.        ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray());
18.        ObjectInputStream ois = new ObjectInputStream(bin);
19.        return (T) ois.readObject();
20.
21.        // 说明：调用ByteArrayInputStream或ByteArrayOutputStream对象的close方法没有任何意义
22.        // 这两个基于内存的流只要垃圾回收器清理对象就能够释放资源，这一点不同于对外部资源（如文件流）的释放
23.    }
24. }
```

下面是测试代码：

```
1. import java.io.Serializable;
2.
3. /**
```

```
 * 人类
 * @author 骆昊
 *
 */
class Person implements Serializable {
    private static final long serialVersionUID = -9102017020286042305L;

    private String name;    // 姓名
    private int age;        // 年龄
    private Car car;        // 座驾

    public Person(String name, int age, Car car) {
        this.name = name;
        this.age = age;
        this.car = car;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public Car getCar() {
        return car;
    }

    public void setCar(Car car) {
        this.car = car;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + ", car=" + car + "]";
    }

}
```

```
/**
 * 小汽车类
 * @author 骆昊
 *
```

```
 */
class Car implements Serializable {
    private static final long serialVersionUID = -5713945027627603702L;

    private String brand;       // 品牌
    private int maxSpeed;       // 最高时速

    public Car(String brand, int maxSpeed) {
        this.brand = brand;
        this.maxSpeed = maxSpeed;
    }

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    public int getMaxSpeed() {
        return maxSpeed;
    }

    public void setMaxSpeed(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }

    @Override
    public String toString() {
        return "Car [brand=" + brand + ", maxSpeed=" + maxSpeed + "]";
    }

}
```

```
class CloneTest {

    public static void main(String[] args) {
        try {
            Person p1 = new Person("Hao LUO", 33, new Car("Benz", 300));
            Person p2 = MyUtil.clone(p1);   // 深度克隆
            p2.getCar().setBrand("BYD");
            // 修改克隆的Person对象p2关联的汽车对象的品牌属性
            // 原来的Person对象p1关联的汽车不会受到任何影响
            // 因为在克隆Person对象时其关联的汽车对象也被克隆了
            System.out.println(p1);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

注意：基于序列化和反序列化实现的克隆不仅仅是深度克隆，更重要的是通过泛型限定，可以检查出要克隆的对象是否支持序列化，这项检查是编译器完成的，不是在运行时抛出异常，这种是方案明显优于使用Object类的clone方法克隆对象。让问题在编译的时候暴露出来总是优于把问题留到运行时。

String s = new String(“xyz”);创建了几个字符串对象？

两个对象，一个是静态区的”xyz”，一个是用new创建在堆上的对象。

java中==和eqauls()的区别,equals()和`hashcode的区别

==是运算符,用于比较两个变量是否相等,而equals是Object类的方法,用于比较两个对象是否相等.默认Object类的equals方法是比较两个对象的地址,此时和==的结果一样.换句话说:基本类型比较用==,比较的是他们的值.默认下,对象用==比较时,比较的是内存地址,如果需要比较对象内容,需要重写equal方法

a==b与a.equals(b)有什么区别

如果a 和b 都是对象，则 a==b 是比较两个对象的引用，只有当 a 和 b 指向的是堆中的同一个对象才会返回 true，而 a.equals(b) 是进行逻辑比较，所以通常需要重写该方法来提供逻辑一致性的比较。例如，String 类重写 equals() 方法，所以可以用于两个不同对象，但是包含的字母相同的比较。

接口是否可继承（extends）接口？抽象类是否可实现（implements）接口？抽象类是否可继承具体类（concrete class）？

接口可以继承接口，而且支持多重继承。抽象类可以实现(implements)接口，抽象类可继承具体类也可以继承抽象类。

Java 中的final关键字有哪些用法？

(1)修饰类：表示该类不能被继承；

(2)修饰方法：表示方法不能被重写；

(3)修饰变量：表示变量只能一次赋值以后值不能被修改（常量）。

throw和throws的区别

throw用于主动抛出java.lang.Throwable 类的一个实例化对象，意思是说你可以通过关键字 throw 抛出一个 Error 或者 一个Exception，如：throw new IllegalArgumentException(“size must be multiple of 2")。

throws 的作用是作为方法声明和签名的一部分，方法被抛出相应的异常以便调用者能处理。Java 中，任何未处理的受检查异常强制在 throws 子句中声明。

Error和Exception有什么区别？

Error表示系统级的错误和程序不必处理的异常，是恢复不是不可能但很困难的情况下的一种严重问题；比如内存溢出，不可能指望程序能处理这样的情况；

Exception表示需要捕捉或者需要程序进行处理的异常，是一种设计或实现问题；也就是说，它表示如果程序运行正常，从不会发生的情况。

Java语言如何进行异常处理，关键字：throws、throw、try、catch、finally分别如何使用？

Java通过面向对象的方法进行异常处理，把各种不同的异常进行分类，并提供了良好的接口。在Java中，每个异常都是一个对象，它是Throwable类或其子类的实例。当一个方法出现异常后便抛出一个异常对象，该对象中包含有异常信息，调用这个对象的方法可以捕获到这个异常并可以对其进行处理。

Java的异常处理是通过5个关键词来实现的：try、catch、throw、throws和finally。

一般情况下是用try来执行一段程序，如果系统会抛出（throw）一个异常对象，可以通过它的类型来捕获（catch）它，或通过总是执行代码块（finally）来处理；
try用来指定一块预防所有异常的程序；
catch子句紧跟在try块后面，用来指定你想要捕获的异常的类型；
throw语句用来明确地抛出一个异常；
throws用来声明一个方法可能抛出的各种异常（当然声明异常时允许无病呻吟）；
finally为确保一段代码不管发生什么异常状况都要被执行；

try语句可以嵌套，每当遇到一个try语句，异常的结构就会被放入异常栈中，直到所有的try语句都完成。如果下一级的try语句没有对某种异常进行处理，异常栈就会执行出栈操作，直到遇到有处理这种异常的try语句或者最终将异常抛给JVM。

运行时异常与受检异常有何异同？

异常表示程序运行过程中可能出现的非正常状态，运行时异常表示虚拟机的通常操作中可能遇到的异常，是一种常见运行错误，只要程序设计得没有问题通常就不会发生。受检异常跟程序运行的上下文环境有关，即使程序设计无误，仍然可能因使用的问题而引发。Java编译器要求方法必须声明抛出可能发生的受检异常，但是并不要求必须声明抛出未被捕获的运行时异常。

异常和继承一样，是面向对象程序设计中经常被滥用的东西，在Effective Java中对异常的使用给出了以下指导原则：

- 不要将异常处理用于正常的控制流（设计良好的API不应该强迫它的调用者为了正常的控制流而使用异常）
- 对可以恢复的情况使用受检异常，对编程错误使用运行时异常
- 避免不必要的使用受检异常（可以通过一些状态检测手段来避免异常的发生）
- 优先使用标准的异常

- 每个方法抛出的异常都要有文档
- 保持异常的原子性
- 不要在catch中忽略掉捕获到的异常

列出一些你常见的运行时异常？

- ArithmeticException（算术异常）
- ClassCastException （类转换异常）
- IllegalArgumentException （非法参数异常）
- IndexOutOfBoundsException （下标越界异常）
- NullPointerException （空指针异常）
- SecurityException （安全异常）

阐述final、finally、finalize的区别

- final：修饰符（关键字）有三种用法：如果一个类被声明为final，意味着它不能再派生出新的子类，即不能被继承，因此它和abstract是反义词。将变量声明为final，可以保证它们在使用中不被改变，被声明为final的变量必须在声明时给定初值，而在以后的引用中只能读取不可修改。被声明为final的方法也同样只能使用，不能在子类中被重写。
- finally：通常放在try…catch…的后面构造总是执行代码块，这就意味着程序无论正常执行还是发生异常，这里的代码只要JVM不关闭都能执行，可以将释放外部资源的代码写在finally块中。
- finalize：Object类中定义的方法，Java中允许使用finalize()方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在销毁对象时调用的，通过重写finalize()方法可以整理系统资源或者执行其他清理工作。

java当中的四种引用

强引用,软引用,弱引用,虚引用.不同的引用类型主要体现在GC上：

强引用：如果一个对象具有强引用，它就不会被垃圾回收器回收。即使当前内存空间不足，JVM也不会回收它，而是抛出 OutOfMemoryError 错误，使程序异常终止。如果想中断强引用和某个对象之间的关联，可以显式地将引用赋值为null，这样一来的话，JVM在合适的时间就会回收该对象

软引用：在使用软引用时，如果内存的空间足够，软引用就能继续被使用，而不会被垃圾回收器回收，只有在内存不足时，软引用才会被垃圾回收器回收。

弱引用：具有弱引用的对象拥有的生命周期更短暂。因为当 JVM 进行垃圾回收，一旦发现弱引用对象，无论当前内存空间是否充足，都会将弱引用回收。不过由于垃圾回收器是一个优先级较低的线程，所以并不一定能迅速发现弱引用对象

虚引用：顾名思义，就是形同虚设，如果一个对象仅持有虚引用，那么它相当于没有引用，在任何时候都可能被垃圾回收器回收。

更多了解参见深入对象引用： `http://blog.csdn.net/dd864140130/article/details/49885811`

为什么要有不同的引用类型

不像C语言,我们可以控制内存的申请和释放,在Java中有时候我们需要适当的控制对象被回收的时机,因此就诞生了不同的引用类型,可以说不同的引用类型实则是对GC回收时机不可控的妥协.有以下几个使用场景可以充分的说明:

利用软引用和弱引用解决OOM问题:用一个HashMap来保存图片的路径和相应图片对象关联的软引用之间的映射关系,在内存不足时,JVM会自动回收这些缓存图片对象所占用的空间,从而有效地避免了OOM的问题.

通过软引用实现Java对象的高速缓存:比如我们创建了一Person的类,如果每次需要查询一个人的信息,哪怕是几秒中之前刚刚查询过的,都要重新构建一个实例,这将引起大量Person对象的消耗,并且由于这些对象的生命周期相对较短,会引起多次GC影响性能。此时,通过软引用和 HashMap 的结合可以构建高速缓存,提供性能.

内部类的作用

内部类可以有多个实例,每个实例都有自己的状态信息,并且与其他外围对象的信息相互独立.在单个外围类当中,可以让多个内部类以不同的方式实现同一接口,或者继承同一个类.创建内部类对象的时刻不依赖于外部类对象的创建.内部类并没有令人疑惑的"is-a"关系,它就像是一个独立的实体.

内部类提供了更好的封装,除了该外围类,其他类都不能访问

SimpleDateFormat是线程安全的吗？

非常不幸,DateFormat 的所有实现,包括 SimpleDateFormat 都不是线程安全的,因此你不应该在多线程序中使用,除非是在对外线程安全的环境中使用,如 将 SimpleDateFormat 限制在 ThreadLocal 中。如果你不这么做,在解析或者格式化日期的时候,可能会获取到一个不正确的结果。因此,从日期、时间处理的所有实践来说,我强力推荐 joda-time 库。

如何格式化日期？

Java 中,可以使用 SimpleDateFormat 类或者 joda-time 库来格式日期。DateFormat 类允许你使用多种流行的格式来格式化日期。参见答案中的示例代码,代码中演示了将日期格式化成不同的格式,如 dd-MM-yyyy 或 ddMMyyyy。

说出几条 Java 中方法重载的最佳实践？

下面有几条可以遵循的方法重载的最佳实践来避免造成自动装箱的混乱。

- 不要重载这样的方法:一个方法接收 int 参数,而另个方法接收 Integer 参数。
- 不要重载参数数量一致,而只是参数顺序不同的方法。
- 如果重载的方法参数个数多于 5 个,采用可变参数。

说说反射的用途及实现

反射机制是Java语言中一个非常重要的特性,它允许程序在运行时进行自我检查,同时也允许对其内部成员进行操作。

反射机制提供的功能主要有：得到一个对象所属的类；获取一个类的所有成员变量和方法；在运行时创建对象；在运行时调用对象的方法；

说说自定义注解的场景及实现

登陆、权限拦截、日志处理，以及各种 Java 框架，如 Spring，Hibernate，JUnit 提到注解就不能不说反射，Java 自定义注解是通过运行时靠反射获取注解。

实际开发中，例如我们要获取某个方法的调用日志，可以通过 AOP（动态代理机制）给方法添加切面，通过反射来获取方法包含的注解，如果包含日志注解，就进行日志记录。
反射的实现在 Java 应用层面上讲，是通过对 Class 对象的操作实现的，Class 对象为我们提供了一系列方法对类进行操作。在 JVM 这个角度来说，Class 文件是一组以 8 位字节为基础单位的二进制流，各个数据项目按严格的顺序紧凑的排列在 Class 文件中，里面包含了类、方法、字段等等相关数据。

通过对 Class 数据流的处理我们即可得到字段、方法等数据。

什么要重写hashcode()和equals()以及他们之间的区别与关系？

可以参考：

- 为什么要重写hashCode()方法和equals()方法以及如何进行重写
- Java hashCode() 和 equals()的若干问题解答
- Java中equals()与hashCode()方法详解

集合

集合

Java中的集合及其继承关系

关于集合的体系是每个人都应该烂熟于心的,尤其是对我们经常使用的List,Map的原理更该如此.这里我们看这张图即可:

List、Set、Map是否继承自Collection接口？

List、Set 是，Map 不是。Map是键值对映射容器，与List和Set有明显的区别，而Set存储的零散的元素且不允许有重复元素（数学中的集合也是如此），List是线性结构的容器，适用于按数值索引访问元素的情形。

阐述ArrayList、Vector、LinkedList的存储性能和特性。

ArrayList 和Vector都是使用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，它们都允许直接按序号索引元素，但是插入元素要涉及数组元素移动等内存操作，所以索引数据快而插入数据慢。Vector中的方法由于添加了synchronized修饰，因此Vector是线程安全的容器，但性能上较ArrayList差，因此已经是Java中的遗留容器。

LinkedList使用双向链表实现存储（将内存中零散的内存单元通过附加的引用关联起来，形成一个可以按序号索引的线性结构，这种链式存储方式与数组的连续存储方式相比，内存的利用率更高），按序号索引数据需要进行前向或后向遍历，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快。

Vector属于遗留容器（Java早期的版本中提供的容器，除此之外，Hashtable、Dictionary、BitSet、Stack、Properties都是遗留容器），已经不推荐使用，但是由于ArrayList和LinkedListed都是非线程安全的，如果遇到多个线程操作同一个容器的场景，则可以通过工具类Collections中的synchronizedList方法将其转换成线程安全的容器后再使用（这是对装潢模式的应用，将已有对象传入另一个类的构造器中创建新的对象来增强实现）。

Collection和Collections的区别？

Collection是一个接口，它是Set、List等容器的父接口；Collections是个一个工具类，提供了一系列的静态方法来辅助容器操作，这些方法包括对容器的搜索、排序、线程安全化等等。

List、Map、Set三个接口存取元素时，各有什么特点？

List以特定索引来存取元素，可以有重复元素。

Set不能存放重复元素（用对象的equals()方法来区分元素是否重复）。

Map保存键值对（key-value pair）映射，映射关系可以是一对一或多对一。

Set和Map容器都有基于哈希存储和排序树的两种实现版本，基于哈希存储的版本理论存取时间复杂度为O(1)，而基于排序树版本的实现在插入或删除元素时会按照元素或元素的键（key）构成排序树从而达到排序和去重的效果。

List和Set区别

Set是最简单的一种集合。集合中的对象不按特定的方式排序，并且没有重复对象。

- HashSet： HashSet类按照哈希算法来存取集合中的对象，存取速度比较快
- TreeSet ：TreeSet类实现了SortedSet接口，能够对集合中的对象进行排序。

List的特征是其元素以线性方式存储，集合中可以存放重复对象。

- ArrayList() ： 代表长度可以改变得数组。可以对元素进行随机的访问，向ArrayList()中插入与删除元素的速度慢。
- LinkedList(): 在实现中采用链表数据结构。插入和删除速度快，访问速度慢。

LinkedHashMap和PriorityQueue的区别

PriorityQueue 是一个优先级队列,保证最高或者最低优先级的的元素总是在队列头部，但是 LinkedHashMap 维持的顺序是元素插入的顺序。当遍历一个 PriorityQueue 时，没有任何顺序保证，但是 LinkedHashMap 课保证遍历顺序是元素插入的顺序。

WeakHashMap与HashMap的区别是什么？

WeakHashMap 的工作与正常的 HashMap 类似，但是使用弱引用作为 key，意思就是当 key 对象没有任何引用时，key/value 将会被回收。

ArrayList和LinkedList的区别？

最明显的区别是 ArrrayList底层的数据结构是数组，支持随机访问，而 LinkedList 的底层数据结构是双向循环链表，不支持随机访问。使用下标访问一个元素，ArrayList 的时间复杂度是 O(1)，而 LinkedList 是 O(n)。

相对于ArrayList，LinkedList的插入，添加，删除操作速度更快，因为当元素被添加到集合任意位置的时候，不需要像数组那样重新计算大小或者是更新索引。

LinkedList比ArrayList更占内存，因为LinkedList为每一个节点存储了两个引用，一个指向前一个元素，一个指向下一个元素。

ArrayList和Array有什么区别？

Array可以容纳基本类型和对象，而ArrayList只能容纳对象。

Array是指定大小的，而ArrayList大小是固定的

ArrayList与Vector区别

ArrayList和Vector在很多时候都很类似。

- 两者都是基于索引的，内部由一个数组支持。
- 两者维护插入的顺序，我们可以根据插入顺序来获取元素。
- ArrayList和Vector的迭代器实现都是fail-fast的。
- ArrayList和Vector两者允许null值，也可以使用索引值对元素进行随机访问。

以下是ArrayList和Vector的不同点。

- Vector是同步的，而ArrayList不是。然而，如果你寻求在迭代的时候对列表进行改变，你应该使用CopyOnWriteArrayList。
- ArrayList比Vector快，它因为有同步，不会过载。
- ArrayList更加通用，因为我们可以使用Collections工具类轻易地获取同步列表和只读列表。

HashMap和Hashtable的区别

HashMap和Hashtable都实现了Map接口，因此很多特性非常相似。但是，他们有以下不同点：

- HashMap允许键和值是null，而Hashtable不允许键或者值是null。
- Hashtable是同步的，而HashMap不是。因此，HashMap更适合于单线程环境，而Hashtable适合于多线程环境。
- HashMap提供了可供应用迭代的键的集合，因此，HashMap是快速失败的。另一方面，Hashtable提供了对键的列举(Enumeration)。
- 一般认为Hashtable是一个遗留的类。

HashSet和HashMap区别

- HashSet实现了Set接口，它不允许集合中有重复的值。它存储的是对象
- HashMap实现了Map接口，Map接口对键值对进行映射。Map中不允许重复的键。Map接口有两个基本的实现，HashMap和TreeMap。

HashMap和ConcurrentHashMap的区别

- ConcurrentHashMap对整个桶数组进行了分段，而HashMap则没有。
- ConcurrentHashMap在每一个分段上都用锁进行保护，从而让锁的粒度更精细一些，并发性能更好，而HashMap没有锁机制，不是线程安全的。

引入ConcurrentHashMap是为了在同步集合HashTable之间有更好的选择，HashTable与HashMap、ConcurrentHashMap主要的区别在于HashMap不是同步的、线程不安全的和不适合应用于多线程并发环境下，而ConcurrentHashMap是线程安全的集合容器，特别是在多线程和并发环境中，通常作为Map的主要实现。

Comparator和Comparable的区别？

Comparable 接口用于定义对象的自然顺序，而 comparator 通常用于定义用户定制的顺序。Comparable 总是只有一个，但是可以有多个 comparator 来定义对象的顺序。

poll()方法和remove()方法区别？

poll() 和 remove() 都是从队列中取出一个元素，但是 poll() 在获取元素失败的时候会返回空，但是 remove() 失败的时候会抛出异常。

ArrayList、HashMa和LinkedList的默认空间是多少？扩容机制是什么

- ArrayList 的默认大小是 10 个元素。扩容点规则是，新增的时候发现容量不够用了，就去扩容；扩容大小规则是：扩容后的大小= 原始大小+原始大小/2 + 1。
- HashMap 的默认大小是16个元素（必须是2的幂）。扩容因子默认0.75，扩容机制.(当前大小 和 当前容量的比例超过了 扩容因子，就会扩容，扩容后大小为 一倍。例如：初始大小为 16 ，扩容因子 0.75 ，当容量为12的时候，比例已经是0.75 。触发扩容，扩容后的大小为 32.)
- LinkedList 是一个双向链表，没有初始化大小，也没有扩容的机制，就是一直在前面或者后面新增就好。

```
1. private static final int DEFAULT_CAPACITY = 10;
2.
3. //from HashMap.java JDK 7
4. static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

如何实现集合排序？

你可以使用有序集合，如 TreeSet 或 TreeMap，你也可以使用有顺序的的集合，如 list，然后通过 Collections.sort() 来排序。

如何打印数组内容

你可以使用 Arrays.toString() 和 Arrays.deepToString() 方法来打印数组。由于数组没有实现 toString() 方法，所以如果将数组传递给 System.out.println() 方法，将无法打印出数组的内容，但是 Arrays.toString() 可以打印每个元素。

LinkedList的是单向链表还是双向？

双向循环列表,具体实现自行查阅源码.

TreeMap是实现原理

采用红黑树实现,具体实现自行查阅源码.

遍历ArrayList时如何正确移除一个元素

该问题的关键在于面试者使用的是 ArrayList 的 remove() 还是 Iterator 的 remove()方法。这有一段示例代码，是使用正确的方式来实现在遍历的过程中移除元素，而不会出现 ConcurrentModificationException 异常的示例代码。

什么是ArrayMap？它和HashMap有什么区别？

ArrayMap是Android SDK中提供的,非Android开发者可以略过。

ArrayMap是用两个数组来模拟map,更少的内存占用空间,更高的效率。

具体参考这篇文章:ArrayMap VS HashMap: `http://lvable.com/?p=217%5D`

如何决定选用HashMap还是TreeMap？

对于在Map中插入、删除和定位元素这类操作，HashMap是最好的选择。然而，假如你需要对一个有序的key集合进行遍历，TreeMap是更好的选择。基于你的collection的大小，也许向HashMap中添加元素会更快，将map换为TreeMap进行有序key的遍历。

HashMap的实现原理

1. HashMap概述： HashMap是基于哈希表的Map接口的非同步实现。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。
2. HashMap的数据结构： 在java编程语言中，最基本的结构就是两种，一个是数组，另外一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap也不例外。HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。

当我们往Hashmap中put元素时,首先根据key的hashcode重新计算hash值,根绝hash值得到这个元素在数组中的位置(下标),如果该数组在该位置上已经存放了其他元素,那么在这个位置上的元素将以链表的形式存放,新加入的放在链头,最先加入的放入链尾.如果数组中该位置没有元素,就直接将该元素放到数组的该位置上.

需要注意Jdk 1.8中对HashMap的实现做了优化,当链表中的节点数据超过八个之后,该链表会转为红黑树来提高查询效率,从原来的O(n)到O(logn)

也可以参考:

- 深入Java集合学习系列：HashMap的实现原理
- 深入理解HashMap

解决Hash冲突的方法有哪些

开放地址法、链地址法、再哈希法、建立公共溢出区等

参考:

- java 解决Hash(散列)冲突的四种方法–开放定址法(线性探测,二次探测,伪随机探测)、链地址法、再哈希、建立公共溢出区
- Java 8中HashMap冲突解决

ConcurrentHashMap 的工作原理及代码实现

ConcurrentHashMap具体是怎么实现线程安全的呢，肯定不可能是每个方法加synchronized，那样就变成了HashTable。

从ConcurrentHashMap代码中可以看出，它引入了一个“分段锁”的概念，具体可以理解为把一个大的Map拆分成N个小的HashTable，根据key.hashCode()来决定把key放到哪个HashTable中。

在ConcurrentHashMap中，就是把Map分成了N个Segment，put和get的时候，都是现根据key.hashCode()算出放到哪个Segment中。

你了解Fail-Fast机制吗

Fail-Fast即我们常说的快速失败,

更多内容参看fail-fast机制: `http://blog.csdn.net/chenssy/article/details/38151189`

Fail-fast和Fail-safe有什么区别

Iterator的fail-fast属性与当前的集合共同起作用，因此它不会受到集合中任何改动的影响。Java.util包中的所有集合类都被设计为fail->fast的，而java.util.concurrent中的集合类都为fail-safe的。当检测到正在遍历的集合的结构被改变时，Fail-fast迭代器抛出ConcurrentModificationException，而fail-safe迭代器从不抛出ConcurrentModificationException。

说出几点 Java 中使用 Collections 的最佳实践

这是我在使用 Java 中 Collectionc 类的一些最佳实践:

- 使用正确的集合类，例如，如果不需要同步列表，使用 ArrayList 而不是 Vector。
- 优先使用并发集合，而不是对集合进行同步。并发集合提供更好的可扩展性。
- 使用接口代表和访问集合，如使用List存储 ArrayList，使用 Map 存储 HashMap 等等。
- 使用迭代器来循环集合。
- 使用集合的时候使用泛型。

BlockingQueue是什么？

Java.util.concurrent.BlockingQueue是一个队列，在进行检索或移除一个元素的时候，它会等待队列变为非空；当在添加一个元素时，它会等待队列中的可用空间。BlockingQueue接口是Java集合框架的一部分，主要用于实现生产者-消费者模式。我们不需要担心等待生产者有可用的空间，或消费者有可用的对象，因为它都在

BlockingQueue的实现类中被处理了。Java提供了集中BlockingQueue的实现，比如ArrayBlockingQueue、LinkedBlockingQueue、PriorityBlockingQueue,、SynchronousQueue

队列和栈是什么，列出它们的区别？

栈和队列两者都被用来预存储数据。java.util.Queue是一个接口，它的实现类在Java并发包中。队列允许先进先出（FIFO）检索元素，但并非总是这样。Deque接口允许从两端检索元素。

栈与队列很相似，但它允许对元素进行后进先出（LIFO）进行检索。

Stack是一个扩展自Vector的类，而Queue是一个接口。

多线程情况下HashMap死循环的问题

可以参考：疫苗：JAVA HASHMAP的死循环

HashMap出现Hash DOS攻击的问题

可以参考：HASH COLLISION DOS 问题

Java Collections和Arrays的sort方法默认的排序方法是什么？

参考：Collections.sort()和Arrays.sort()排序算法选择

多线程

多线程

什么是进程，什么是线程，为什么需要多线程编程？

进程是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，是操作系统进行资源分配和调度的一个独立单位；

线程是进程的一个实体，是CPU调度和分派的基本单位，是比进程更小的能独立运行的基本单位。线程的划分尺度小于进程，这使得多线程程序的并发性高；进程在执行时通常拥有独立的内存单元，而线程之间可以共享内存。

使用多线程的编程通常能够带来更好的性能和用户体验，但是多线程的程序对于其他程序是不友好的，因为它可能占用了更多的CPU资源。当然，也不是线程越多，程序的性能就越好，因为线程之间的调度和切换也会浪费CPU时间。时下很时髦的Node.js就采用了单线程异步I/O的工作模式。

什么是线程安全

如果你的代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么你的代码就是线程安全的。

这个问题有值得一提的地方，就是线程安全也是有几个级别的：

- 不可变。像String、Integer、Long这些，都是final类型的类，任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用
- 绝对线程安全。不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java中也有，比方说CopyOnWriteArrayList、CopyOnWriteArraySet
- 相对线程安全。相对线程安全也就是我们通常意义上所说的线程安全，像Vector这种，add、remove方法都是原子操作，不会被打断，但也仅限于此，如果有个线程在遍历某个Vector、有个线程同时在add这个Vector，99%的情况下都会出现ConcurrentModificationException，也就是fail-fast机制。
- 线程非安全。这个就没什么好说的了，ArrayList、LinkedList、HashMap等都是线程非安全的类

编写多线程程序有几种实现方式？

Java 5以前实现多线程有两种实现方法：一种是继承Thread类；另一种是实现Runnable接口。
两种方式都要通过重写run()方法来定义线程的行为，推荐使用后者，因为Java中的继承是单继承，一个类有一个父类，如果继承了Thread类就无法再继承其他类了，显然使用Runnable接口更为灵活。

Java 5以后创建线程还有第三种方式：实现Callable接口，该接口中的call方法可以在线程执行结束时产生一个返回值。

synchronized关键字的用法？

synchronized关键字可以将对象或者方法标记为同步，以实现对对象和方法的互斥访问，可以用synchronized(对象) { … }定义同步代码块，或者在声明方法时将synchronized作为方法的修饰符。

简述synchronized 和java.util.concurrent.locks.Lock的异同？

Lock是Java 5以后引入的新的API，和关键字synchronized相比主要相同点：Lock 能完成synchronized所实现的所有功能；主要不同点：Lock有比synchronized更精确的线程语义和更好的性能，而且不强制性的要求一定要获得锁。synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且最好在finally 块中释放（这是释放外部资源的最好的地方）。

当一个线程进入一个对象的synchronized方法A之后，其它线程是否可进入此对象的synchronized方法B？

不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的synchronized修饰符要求执行方法时要获得对象的锁，如果已经进入A方法说明对象锁已经被取走，那么试图进入B方法的线程就只能在等锁池（注意不是等待池哦）中等待对象的锁。

synchronized和ReentrantLock的区别

synchronized是和if、else、for、while一样的关键字，ReentrantLock是类，这是二者的本质区别。既然ReentrantLock是类，那么它就提供了比synchronized更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，ReentrantLock比synchronized的扩展性体现在几点上：

1. ReentrantLock可以对获取锁的等待时间进行设置，这样就避免了死锁
2. ReentrantLock可以获取各种锁的信息
3. ReentrantLock可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的:ReentrantLock底层调用的是Unsafe的park方法加锁，synchronized操作的应该是对象头中mark word.

举例说明同步和异步。

如果系统中存在临界资源（资源数量少于竞争资源的线程数量的资源），例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就必须进行同步存取（数据库操作中的排他锁就是最好的例子）。当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。事实上，所谓的同步就是指阻塞式操作，而异步就是非阻塞式操作。

启动一个线程是调用run()还是start()方法？

启动一个线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由JVM 调度并执行，这并不意味着线程就会立即运行。run()方法是线程启动后要进行回调（callback）的方法。

为什么需要run()和start()方法，我们可以只用run()方法来完成任务吗？

我们需要run()&start()这两个方法是因为JVM创建一个单独的线程不同于普通方法的调用，所以这项工作由线程的start方法来完成，start由本地方法实现，需要显示地被调用，使用这俩个方法的另外一个好处是任何一个对象都可以作为线程运行，只要实现了Runnable接口，这就避免因继承了Thread类而造成的Java的多继承问题。

什么是线程池（thread pool）？

在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。

在Java中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这就是“池化资源”技术产生的原因。线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。

Java 5+中的Executor接口定义一个执行线程的工具。它的子类型即线程池接口是ExecutorService。要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，因此在工具类Executors面提供了一些静态工厂方法，生成一些常用的线程池，如下所示：

- newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。
- newFixedThreadPool：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。
- newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。
- newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。
- newSingleThreadExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

线程的基本状态以及状态之间的关系？

其中Running表示运行状态；Runnable表示就绪状态（万事俱备，只欠CPU）；Blocked表示阻塞状态；阻塞状态又有多种情况，可能是因为调用wait()方法进入等待池，也可能是执行同步方法或同步代码块进入等锁池，或者是调用了sleep()方法或join()方法等待休眠或其他线程结束，或是因为发生了I/O中断。

Java中如何实现序列化，有什么意义？

序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操

作，也可将流化后的对象传输于网络之间。序列化是为了解决对象流读写操作时可能引发的问题（如果不进行序列化可能会存在数据乱序的问题）。

要实现序列化，需要让一个类实现Serializable接口，该接口是一个标识性接口，标注该类对象是可被序列化的，然后使用一个输出流来构造一个对象输出流并通过writeObject(Object)方法就可以将实现对象写出（即保存其状态）；如果需要反序列化则可以用一个输入流建立对象输入流，然后通过readObject方法从流中读取对象。序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆。

产生死锁的条件

1. 互斥条件：一个资源每次只能被一个进程使用。
2. 请求与保持条件：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. 不剥夺条件:进程已获得的资源，在未使用完之前，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。

什么是线程饿死，什么是活锁？

线程饿死和活锁虽然不想是死锁一样的常见问题，但是对于并发编程的设计者来说就像一次邂逅一样。

当所有线程阻塞，或者由于需要的资源无效而不能处理，不存在非阻塞线程使资源可用。JavaAPI中线程活锁可能发生在以下情形：

- 当所有线程在程序中执行Object.wait(0)，参数为0的wait方法。程序将发生活锁直到在相应的对象上有线程调用Object.notify()或者Object.notifyAll()。
- 当所有线程卡在无限循环中。

什么导致线程阻塞

阻塞指的是暂停一个线程的执行以等待某个条件发生（如某资源就绪），学过操作系统的同学对它一定已经很熟悉了。Java 提供了大量方法来支持阻塞，下面让我们逐一分析。

方法	说明
sleep()	sleep() 允许 指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到CPU 时间，指定的时间一过，线程重新进入可执行状态。 典型地，sleep() 被用在等待某个资源就绪的情形：测试发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止
suspend() 和 resume()	两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的 resume() 被调用，才能使得线程重新进入可执行状态。典型地，suspend() 和 resume() 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume() 使其恢复。
yield()	yield() 使当前线程放弃当前已经分得的CPU 时间，但不使当前线程阻塞，即线程仍处于可执行状态，随时可能再次分得 CPU 时间。调用 yield() 的效果等价于调度程序认为该线程已执行了足够的时间从而转到另一个线程。
wait() 和 notify()	两个方法配套使用，wait() 使得线程进入阻塞状态，它有两种形式，一种允许 指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的 notify() 被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的 notify() 被调用.

怎么检测一个线程是否持有对象监视器

Thread类提供了一个holdsLock(Object obj)方法，当且仅当对象obj的监视器被某条线程持有的时候才会返回true，注意这是一个static方法，这意味着”某条线程”指的是当前线程。

请说出与线程同步以及线程调度相关的方法。

- wait(): 使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；
- sleep(): 使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException异常；
- notify(): 唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；
- notityAll(): 唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

sleep() 、join（）、yield（）有什么区别

- sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；
- 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；
- sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；
- sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

wait(),notify()和suspend(),resume()之间的区别

初看起来它们与 suspend() 和 resume() 方法对没有什么分别，但是事实上它们是截然不同的。区别的核心在于，前面叙述的所有方法，阻塞时都不会释放占用的锁（如果占用了的话），而这一对方法则相反。上述的核心区别导致了一系列的细节上的区别。

首先，前面叙述的所有方法都隶属于 Thread 类，但是这一对却直接隶属于 Object 类，也就是说，所有对象都拥有这一对方法。初看起来这十分不可思议，但是实际上却是很自然的，因为这一对方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用任意对象的 wait() 方法导致线程阻塞，并且该对象上的锁被释放。而调用 任意对象的notify()方法则导致从调用该对象的 wait() 方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真正可执行）。

其次，前面叙述的所有方法都可在任何位置调用，但是这一对方法却必须在 synchronized 方法或块中调用，理由也很简单，只有在synchronized 方法或块中当前线程才占有锁，才有锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此，这一对方法调用必须放置在这样的 synchronized 方法或块中，该方法或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现IllegalMonitorStateException 异常。

wait() 和 notify() 方法的上述特性决定了它们经常和synchronized关键字一起使用，将它们和操作系统进程间通信机制作一个比较就会发现它们的相似性：synchronized方法或块提供了类似于操作系统原语的功能，它们的执行不会受到多线程机制的干扰，而这一对方法则相当于 block 和wakeup 原语（这一对方法均声明为synchronized）。它们的结合使得我们可以实现操作系统上一系列精妙的进程间通信的算法（如信号量算法），并

用于解决各种复杂的线程间通信问题。

关于 wait() 和 notify() 方法最后再说明两点:

第一: 调用 notify() 方法导致解除阻塞的线程是从因调用该对象的 wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。

第二: 除了 notify()，还有一个方法 notifyAll() 也可起到类似作用，唯一的区别在于，调用 notifyAll() 方法将把因调用该对象的 wait() 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

谈到阻塞，就不能不谈一谈死锁，略一分析就能发现，suspend() 方法和不指定超时期限的 wait() 方法的调用都可能产生死锁。遗憾的是，Java 并不在语言级别上支持死锁的避免，我们在编程中必须小心地避免死锁。

以上我们对 Java 中实现线程阻塞的各种方法作了一番分析，我们重点分析了 wait() 和 notify() 方法，因为它们的功能最强大，使用也最灵活，但是这也导致了它们的效率较低，较容易出错。实际使用中我们应该灵活使用各种方法，以便更好地达到我们的目的。

为什么wait()方法和notify()/notifyAll()方法要在同步块中被调用

这是JDK强制的，wait()方法和notify()/notifyAll()方法在调用前都必须先获得对象的锁

wait()方法和notify()/notifyAll()方法在放弃对象监视器时有什么区别

wait()方法和notify()/notifyAll()方法在放弃对象监视器的时候的区别在于: wait()方法立即释放对象监视器，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象监视器。

Runnable和Callable的区别

Runnable接口中的run()方法的返回值是void，它做的事情只是纯粹地去执行run()方法中的代码而已；Callable接口中的call()方法是有返回值的，是一个泛型，和Future、FutureTask配合可以用来获取异步执行的结果。

这其实是很有用的一个特性，因为多线程相比单线程更难、更复杂的一个重要原因就是因为多线程充满着未知性，某条线程是否执行了？某条线程执行了多久？某条线程执行的时候我们期望的数据是否已经赋值完毕？无法得知，我们能做的只是等待这条多线程的任务执行完毕而已。而Callable+Future/FutureTask却可以方便获取多线程运行的结果，可以在等待时间太长没获取到需要的数据的情况下取消该线程的任务。

Thread类的sleep()方法和对象的wait()方法都可以让线程暂停执行，它们有什么区别？

sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复。

wait()是Object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

线程的sleep()方法和yield()方法有什么区别？

1. sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；
2. 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；
3. sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；
4. sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

为什么wait,nofity和nofityAll这些方法不放在Thread类当中

一个很明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。简单的说，由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。

怎么唤醒一个阻塞的线程

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

什么是多线程的上下文切换

多线程的上下文切换是指CPU控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取CPU执行权的线程的过程。

FutureTask是什么

这个其实前面有提到过，FutureTask表示一个异步运算的任务。FutureTask里面可以传入一个Callable的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然，由于FutureTask也是Runnable接口的实现类，所以FutureTask也可以放入线程池中。

一个线程如果出现了运行时异常怎么办？

如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：如果这个线程持有某个某个对象的监视器，那么这个对象监视器会被立即释放

Java当中有哪几种锁

自旋锁：自旋锁在JDK1.6之后就默认开启了。基于之前的观察，共享数据的锁定状态只会持续很短的时间，为了这一小段时间而去挂起和恢复线程有点浪费，所以这里就做了一个处理，让后面请求锁的那个线程在稍等一会，但是不放弃处理器的执行时间，看看持有锁的线程能否快速释放。为了让线程等待，所以需要让线程执行一个忙循环也就是自旋操作。在jdk6之后，引入了自适应的自旋锁，也就是等待的时间不再固定了，而是由上一次在同一个锁上的自旋时间及锁的拥有者状态来决定

偏向锁：在JDK1.之后引入的一项锁优化，目的是消除数据在无竞争情况下的同步原语。进一步提升程序的运行性能。 偏向锁就是偏心的偏，意思是这个锁会偏向第一个获得他的线程，如果接下来的执行过程中，改锁没有被其他线程获取，则持有偏向锁的线程将永远不需要再进行同步。偏向锁可以提高带有同步但无竞争的程序性能，也就是说他并不一定总是对程序运行有利，如果程序中大多数的锁都是被多个不同的线程访问，那偏向模式就是多余的，在具体问题具体分析的前提下，可以考虑是否使用偏向锁。

轻量级锁：为了减少获得锁和释放锁所带来的性能消耗，引入了“偏向锁”和“轻量级锁”，所以在Java SE1.6里锁一共有四种状态，无锁状态，偏向锁状态，轻量级锁状态和重量级锁状态，它会随着竞争情况逐渐升级。锁可以升级但不能降级，意味着偏向锁升级成轻量级锁后不能降级成偏向锁

如何在两个线程间共享数据

通过在线程之间共享对象就可以了，然后通过wait/notify/notifyAll、await/signal/signalAll进行唤起和等待，比方说阻塞队列BlockingQueue就是为线程之间共享数据而设计的

如何正确的使用wait()?使用if还是while？

wait() 方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没有满足，所以在处理前，循环检测条件是否满足会更好。下面是一段标准的使用 wait 和 notify 方法的代码：

```
synchronized (obj) {
   while (condition does not hold)
     obj.wait(); // (Releases lock, and reacquires on wakeup)
     ... // Perform action appropriate to condition
}
```

什么是线程局部变量ThreadLocal

线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java提供ThreadLocal类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放，Java 应用就存在内存泄露的风险。

ThreadLoal的作用是什么？

简单说ThreadLocal就是一种以空间换时间的做法在每个Thread里面维护了一个ThreadLocal.ThreadLocalMap

把数据进行隔离，数据不共享，自然就没有线程安全方面的问题了.

ThreadLocal 原理分析

ThreadLocal为解决多线程程序的并发问题提供了一种新的思路。ThreadLocal，顾名思义是线程的一个本地化对象，当工作于多线程中的对象使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程分配一个独立的变量副本，所以每一个线程都可以独立的改变自己的副本，而不影响其他线程所对应的副本。从线程的角度看，这个变量就像是线程的本地变量。

ThreadLocal类非常简单好用，只有四个方法，能用上的也就是下面三个方法：

- void set(T value)：设置当前线程的线程局部变量的值。
- T get()：获得当前线程所对应的线程局部变量的值。
- void remove()：删除当前线程中线程局部变量的值。

ThreadLocal是如何做到为每一个线程维护一份独立的变量副本的呢？在ThreadLocal类中有一个Map，键为线程对象，值是其线程对应的变量的副本，自己要模拟实现一个ThreadLocal类其实并不困难，代码如下所示：

```
1. import java.util.Collections;
2. import java.util.HashMap;
3. import java.util.Map;
4. public class MyThreadLocal<T> {
5.     private Map<Thread, T> map = Collections.synchronizedMap(new HashMap<Thread, T>());
6.     public void set(T newValue) {
7.         map.put(Thread.currentThread(), newValue);
8.     }
9.     public T get() {
10.         return map.get(Thread.currentThread());
11.     }
12.     public void remove() {
13.         map.remove(Thread.currentThread());
14.     }
15. }
```

如果你提交任务时，线程池队列已满，这时会发生什么

如果你使用的LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为LinkedBlockingQueue可以近乎认为是一个无穷大的队列，可以无限存放任务；如果你使用的是有界队列比方说ArrayBlockingQueue的话，任务首先会被添加到ArrayBlockingQueue中，ArrayBlockingQueue满了，则会使用拒绝策略RejectedExecutionHandler处理满了的任务，默认是AbortPolicy。

为什么要使用线程池

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活地控制并发的数目。

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。

Thread.sleep(0)的作用是什么

由于Java采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到CPU控制权的情况，为了让某些优先级比较低的线程也能获取到CPU控制权，可以使用Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡CPU控制权的一种操作。

什么是CAS

CAS，全称为Compare and Swap，即比较-替换。假设有三个操作数：内存值V、旧的预期值A、要修改的值B，当且仅当预期值A和内存值V相同时，才会将内存值修改为B并返回true，否则什么都不做并返回false。当然CAS一定要volatile变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值A对某条线程来说，永远是一个不会变的值A，只要某次CAS操作失败，永远都不可能成功

什么是乐观锁和悲观锁

乐观锁：乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

悲观锁：悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像synchronized，不管三七二十一，直接上了锁就操作资源了。

ConcurrentHashMap的并发度是什么？

ConcurrentHashMap的并发度就是segment的大小，默认为16，这意味着最多同时可以有16条线程操作ConcurrentHashMap，这也是ConcurrentHashMap对Hashtable的最大优势，任何情况下，Hashtable能同时有两条线程获取Hashtable中的数据吗？

ConcurrentHashMap的工作原理

ConcurrentHashMap在jdk 1.6和jdk 1.8实现原理是不同的.

jdk 1.6:
ConcurrentHashMap是线程安全的，但是与Hashtablea相比，实现线程安全的方式不同。Hashtable是通过对hash表结构进行锁定，是阻塞式的，当一个线程占有这个锁时，其他线程必须阻塞等待其释放锁。
ConcurrentHashMap是采用分离锁的方式，它并没有对整个hash表进行锁定，而是局部锁定，也就是说当一个线程占有这个局部锁时，不影响其他线程对hash表其他地方的访问。
具体实现:ConcurrentHashMap内部有一个Segment

jdk 1.8
在jdk 8中，ConcurrentHashMap不再使用Segment分离锁，而是采用一种乐观锁CAS算法来实现同步问题，但其底层还是“数组+链表->红黑树”的实现。

CyclicBarrier和CountDownLatch区别

这两个类非常类似，都在java.util.concurrent下，都可以用来表示代码运行到某个点上，二者的区别在于：

CyclicBarrier的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；CountDownLatch则不是，某线程运行到某个点上之后，只是给某个数值-1而已，该线程继续运行

CyclicBarrier只能唤起一个任务，CountDownLatch可以唤起多个任务

CyclicBarrier可重用，CountDownLatch不可重用，计数值为0该CountDownLatch就不可再用了

java中的++操作符线程安全么？

不是线程安全的操作。它涉及到多个指令，如读取变量值，增加，然后存储回内存，这个过程可能会出现多个线程交差

有三个线程T1，T2，T3，怎么确保它们按顺序执行？

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。

如何在Java中创建Immutable对象？

这个问题看起来和多线程没什么关系， 但不变性有助于简化已经很复杂的并发程序。Immutable对象可以在没有同步的情况下共享，降低了对该对象进行并发访问时的同步化开销。可是Java没有@Immutable这个注解符，要创建不可变类，要实现下面几个步骤：通过构造方法初始化所有成员、对变量不要提供setter方法、将所有的成员声明为私有的，这样就不允许直接访问这些成员、在getter方法中，不要直接返回对象本身，而是克隆对象，并返回对象的拷贝。

你有哪些多线程开发良好的实践？

- 给线程命名
- 最小化同步范围
- 优先使用volatile
- 尽可能使用更高层次的并发工具而非wait和notify()来实现线程通信,如BlockingQueue,Semeaphore
- 优先使用并发容器而非同步容器.
- 考虑使用线程池

可以创建Volatile数组吗？

Java 中可以创建 volatile类型数组，不过只是一个指向数组的引用，而不是整个数组。如果改变引用指向的数组，将会受到volatile 的保护，但是如果多个线程同时改变数组的元素，volatile标示符就不能起到之前的保护作用了

Volatile关键字的作用

一个非常重要的问题，是每个学习、应用多线程的Java程序员都必须掌握的。理解volatile关键字的作用的前提是要理解Java内存模型，这里就不讲Java内存模型了，可以参见第31点，volatile关键字的作用主要有两个：

- 多线程主要围绕可见性和原子性两个特性而展开，使用volatile关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到volatile变量，一定是最新的数据
- 代码底层执行不像我们看到的高级语言——Java程序这么简单，它的执行是Java代码->字节码->根据字节码执行对应的C/C++代码->C/C++代码被编译成汇编语言->和硬件电路交互，现实中，为了获取更好的性能JVM可能会对指令进行重排序，多线程下可能会出现一些意想不到的问题。使用volatile则会对禁止语义重排序，当然这也一定程度上降低了代码执行效率

从实践角度而言，volatile的一个重要作用就是和CAS结合，保证了原子性，详细的可以参见java.util.concurrent.atomic包下的类，比如AtomicInteger。

volatile能使得一个非原子操作变成原子操作吗？

一个典型的例子是在类中有一个 long 类型的成员变量。如果你知道该成员变量会被多个线程访问，如计数器、价格等，你最好是将其设置为 volatile。为什么？因为 Java 中读取 long 类型变量不是原子的，需要分成两步，如果一个线程正在修改该 long 变量的值，另一个线程可能只能看到该值的一半（前 32 位）。但是对一个 volatile 型的 long 或 double 变量的读写是原子。

一种实践是用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写。double 和 long 都是64位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然后再读剩下的 32 位，这个过程不是原子的，但 Java 中 volatile 型的 long 或 double 变量的读写是原子的。volatile 修复符的另一个作用是提供内存屏障（memory barrier），例如在分布式框架中的应用。简单的说，就是当你写一个 volatile 变量之前，Java 内存模型会插入一个写屏障（write barrier），读一个 volatile 变量之前，会插入一个读屏障（read barrier）。意思就是说，在你写一个 volatile 域时，能保证任何线程都能看到你写的值，同时，在写之前，也能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新到缓存。

volatile类型变量提供什么保证？

volatile 主要有两方面的作用:1.避免指令重排2.可见性保证.例如，JVM 或者 JIT为了获得更好的性能会对语句重排序，但是 volatile 类型变量即使在没有同步块的情况下赋值也不会与其他语句重排序。 volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线程是可见的。某些情况下，volatile 还能提供原子性，如读 64 位数据类型，像 long 和 double 都不是原子的(低32位和高32位)，但 volatile 类型的 double 和 long 就是原子的.

Java 中，编写多线程程序的时候你会遵循哪些最佳实践？

这是我在写Java 并发程序的时候遵循的一些最佳实践：

- 给线程命名，这样可以帮助调试。
- 最小化同步的范围，而不是将整个方法同步，只对关键部分做同步。
- 如果可以，更偏向于使用 volatile 而不是 synchronized。

- 使用更高层次的并发工具，而不是使用 wait() 和 notify() 来实现线程间通信，如 BlockingQueue，CountDownLatch 及 Semeaphore。
- 优先使用并发集合，而不是对集合进行同步。并发集合提供更好的可扩展性。

说出至少 5 点在 Java 中使用线程的最佳实践。

这个问题与之前的问题类似，你可以使用上面的答案。对线程来说，你应该：

- 对线程命名
- 将线程和任务分离，使用线程池执行器来执行 Runnable 或 Callable。
- 使用线程池

Java中如何获取到线程dump文件

死循环、死锁、阻塞、页面打开慢等问题，打线程dump是最好的解决问题的途径。所谓线程dump也就是线程堆栈，获取到线程堆栈有两步：

- 获取到线程的pid，可以通过使用jps命令，在Linux环境下还可以使用ps -ef | grep java
- 打印线程堆栈，可以通过使用jstack pid命令，在Linux环境下还可以使用kill -3 pid

另外提一点，Thread类提供了一个getStackTrace()方法也可以用于获取线程堆栈。这是一个实例方法，因此此方法是和具体线程实例绑定的，每次获取获取到的是具体某个线程当前运行的堆栈。

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

这是我在并发编程网上看到的一个问题，把这个问题放在最后一个，希望每个人都能看到并且思考一下，因为这个问题非常好、非常实际、非常专业。关于这个问题，个人看法是：

1. 高并发、任务执行时间短的业务，线程池线程数可以设置为CPU核数+1，减少线程上下文的切换
2. 并发不高、任务执行时间长的业务要区分开看：
    - 假如是业务时间长集中在IO操作上，也就是IO密集型的任务，因为IO操作并不占用CPU，所以不要让所有的CPU闲下来，可以加大线程池中的线程数目，让CPU处理更多的业务
    - 假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换
3. 并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）。
4. 业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

作业(进程)调度算法

1. 先来先服务调度算法(FCFS) 每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。
2. 短作业(进程)优先调度算法(SPF) 短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。缺点:长作业的运行得不到保证

3. 优先权调度算法(HPF) 当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程，这时，又可进一步把该算法分成如下两种。 可以分为:

    - 非抢占式优先权算法
    - 抢占式优先权调度算法

4. 高响应比优先调度算法(HRN) 每次选择高响应比最大的作业执行，响应比=(等待时间+要求服务时间)/要求服务时间。该算法同时考虑了短作业优先和先来先服务。

    - 如果作业的等待时间相同，则要求服务的时间愈短，其优先权愈高，因而该算法有利于短作业。

    - 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。

    - 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高，从而也可获得处理机。简言之，该算法既照顾了短作业，又考虑了作业到达的先后次序，不会使长作业长期得不到服务。因此，该算法实现了一种较好的折衷。当然，在利用该算法时，每要进行调度之前，都须先做响应比的计算，这会增加系统开销。

5. 时间片轮转法（RR） 在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU分配给队首进程，并令其执行一个时间片。时间片的大小从几ms到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。

6. 多级反馈队列调度算法 它是目前被公认的一种较好的进程调度算法。

    - 应设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第i+1个队列的时间片要比第i个队列的时间片长一倍。

    - 当一个新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按FCFS原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第n队列后，在第n 队列便采取按时间片轮转的方式运行。

    - 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第1～(i-1)队列均空时，才会调度第i队列中的进程运行。如果处理机正在第i队列中为某进程服务时，又有新进程进入优先权较高的队列(第1～(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第i队列的末尾，把处理机分配给新到的高优先权进程。

讲讲线程池的实现原理

以下资源来源

首先要明确为什么要使用线程池，使用线程池会带来什么好处？

- 线程是稀缺资源，不能频繁的创建。
- 应当将其放入一个池子中，可以给其他任务进行复用。
- 解耦作用，线程的创建于执行完全分开，方便维护。

创建一个线程池

以一个使用较多的

```
1. ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit,
   BlockingQueue<Runnable> workQueue, RejectedExecutionHandler handler);
```

为例：

- 其中的 corePoolSize 为线程池的基本大小。
- maximumPoolSize 为线程池最大线程大小。
- keepAliveTime 和 unit 则是线程空闲后的存活时间。
- workQueue 用于存放任务的阻塞队列。
- handler 当队列和最大线程池都满了之后的饱和策略。

处理流程

当提交一个任务到线程池时它的执行流程是怎样的呢？

首先第一步会判断核心线程数有没有达到上限，如果没有则创建线程(会获取全局锁)，满了则会将任务丢进阻塞队列。

如果队列也满了则需要判断最大线程数是否达到上限，如果没有则创建线程(获取全局锁)，如果最大线程数也满了则会根据饱和策略处理。

常用的饱和策略有：

- 直接丢弃任务。
- 调用者线程处理。
- 丢弃队列中的最近任务，执行当前任务。

所以当线程池完成预热之后都是将任务放入队列，接着由工作线程一个个从队列里取出执行。

合理配置线程池

线程池并不是配置越大越好，而是要根据任务的熟悉来进行划分： 如果是 CPU 密集型任务应当分配较少的线程，比

如 CPU 个数相当的大小。

如果是 IO 密集型任务，由于线程并不是一直在运行，所以可以尽可能的多配置线程，比如 CPU 个数 * 2 。

当是一个混合型任务，可以将其拆分为 CPU 密集型任务以及 IO 密集型任务，这样来分别配置。

synchronize 实现原理

以下资源来源

众所周知 Synchronize 关键字是解决并发问题常用解决方案，有以下三种使用方式：

- 同步普通方法，锁的是当前对象。
- 同步静态方法，锁的是当前 Class 对象。
- 同步块，锁的是 {} 中的对象。

实现原理： JVM 是通过进入、退出对象监视器(Monitor)来实现对方法、同步块的同步的。

具体实现是在编译之后在同步方法调用前加入一个 monitor.enter 指令，在退出方法和异常处插入 monitor.exit 的指令。

其本质就是对一个对象监视器(Monitor)进行获取，而这个获取过程具有排他性从而达到了同一时刻只能一个线程访问的目的。

而对于没有获取到锁的线程将会阻塞到方法入口处，直到获取锁的线程 monitor.exit 之后才能尝试继续获取锁。

流程图如下：

synchronize 很多都称之为重量锁，JDK1.6 中对 synchronize 进行了各种优化，为了能减少获取和释放锁带来的消耗引入了偏向锁和轻量锁。

轻量锁

当代码进入同步块时，如果同步对象为无锁状态时，当前线程会在栈帧中创建一个锁记录(Lock Record)区域，同时将锁对象的对象头中 Mark Word 拷贝到锁记录中，再尝试使用 CAS 将 Mark Word 更新为指向锁记录的指针。

如果更新成功，当前线程就获得了锁。

如果更新失败 JVM 会先检查锁对象的 Mark Word 是否指向当前线程的锁记录。

如果是则说明当前线程拥有锁对象的锁，可以直接进入同步块。

不是则说明有其他线程抢占了锁，如果存在多个线程同时竞争一把锁，轻量锁就会膨胀为重量锁。

解锁

轻量锁的解锁过程也是利用 CAS 来实现的，会尝试锁记录替换回锁对象的 Mark Word 。如果替换成功则说明整个同步操作完成，失败则说明有其他线程尝试获取锁，这时就会唤醒被挂起的线程(此时已经膨胀为重量锁)

轻量锁能提升性能的原因是:

认为大多数锁在整个同步周期都不存在竞争，所以使用 CAS 比使用互斥开销更少。但如果锁竞争激烈，轻量锁就不但有互斥的开销，还有 CAS 的开销，甚至比重量锁更慢。

偏向锁

为了进一步的降低获取锁的代价，JDK1.6 之后还引入了偏向锁。

偏向锁的特征是:锁不存在多线程竞争，并且应由一个线程多次获得锁。

当线程访问同步块时，会使用 CAS 将线程 ID 更新到锁对象的 Mark Word 中，如果更新成功则获得偏向锁，并且之后每次进入这个对象锁相关的同步块时都不需要再次获取锁了。

释放锁

当有另外一个线程获取这个锁时，持有偏向锁的线程就会释放锁，释放时会等待全局安全点(这一时刻没有字节码运行)，接着会暂停拥有偏向锁的线程，根据锁对象目前是否被锁来判定将对象头中的 Mark Word 设置为无锁或者是轻量锁状态。

偏向锁可以提高带有同步却没有竞争的程序性能，但如果程序中大多数锁都存在竞争时，那偏向锁就起不到太大作用。可以使用 -XX:-userBiasedLocking=false 来关闭偏向锁，并默认进入轻量锁。

线程池的几种方式与使用场景

参考: 线程池的种类，区别和使用场景

volatile 实现原理

禁止指令重排、刷新内存

参考:

- 【死磕Java并发】--深入分析volatile的实现原理
- 深入分析Volatile的实现原理

synchronized 实现原理

（对象监视器）

说说 Semaphore 原理

说说 Exchanger 原理

线程的生命周期

重入锁的概念，重入锁为什么可以防止死锁

如何检查死锁（通过jConsole检查死锁）

AQS同步队列

什么是ABA问题，出现ABA问题JDK是如何解决的

乐观锁的业务场景及实现方式

JVM

JVM

JVN内存结构

方法区和对是所有线程共享的内存区域；而java栈、本地方法栈和程序员计数器是运行是线程私有的内存区域。

- Java堆（Heap）,是Java虚拟机所管理的内存中最大的一块。Java堆是被所有线程共享的一块内存区域，在虚拟机启动时创建。此内存区域的唯一目的就是存放对象实例，几乎所有的对象实例都在这里分配内存。
- 方法区（Method Area）,方法区（Method Area）与Java堆一样，是各个线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。
- 程序计数器（Program Counter Register）,程序计数器（Program Counter Register）是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。
- JVM栈（JVM Stacks）,与程序计数器一样，Java虚拟机栈（Java Virtual Machine Stacks）也是线程私有的，它的生命周期与线程相同。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（Stack Frame）用于存储局部变量表、操作栈、动态链接、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。
- 本地方法栈（Native Method Stacks）,本地方法栈（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。

对象分配规则

- 对象优先分配在Eden区，如果Eden区没有足够的空间时，虚拟机执行一次Minor GC。
- 大对象直接进入老年代（大对象是指需要大量连续内存空间的对象）。这样做的目的是避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。
- 长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1次Minor GC那么对象会进入Survivor区，之后每经过一次Minor GC那么对象的年龄加1，知道达到阀值对象进入老年区。
- 动态判断对象的年龄。如果Survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入老年代。
- 空间分配担保。每次进行Minor GC时，JVM会计算Survivor区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次Full GC，如果小于检查HandlePromotionFailure设置，如果true则只进行Monitor GC,如果false则进行Full GC。

解释内存中的栈(stack)、堆(heap)和静态区(static area)的用法

通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用内存中的栈空间；而通过new关键字和构造器创建的对象放在堆空间；程序中的字面量（literal）如直接书写的100、”hello”和常量都是放在静态区中。栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，理论上整个内存没有被其他进程使用的空间甚至硬盘上的虚拟内存都可以被当成堆空间来使用。

```
1. String str = new String("hello");
```

上面的语句中变量str放在栈上，用new创建出来的字符串对象放在堆上，而”hello”这个字面量放在静态区。

Perm Space中保存什么数据？会引起OutOfMemory吗？

Perm Space中保存的是加载class文件。

会引起，出现异常可以设置 -XX:PermSize 的大小。JDK 1.8后，字符串常量不存放在永久带，而是在堆内存中，JDK8以后没有永久代概念，而是用元空间替代，元空间不存在虚拟机中，二是使用本地内存。

详细查看Java8内存模型—永久代(PermGen)和元空间(Metaspace)

什么是类的加载

类的加载指的是将类的.class文件中的二进制数据读入到内存中，将其放在运行时数据区的方法区内，然后在堆区创建一个java.lang.Class对象，用来封装类在方法区内的数据结构。类的加载的最终产品是位于堆区中的Class对象，Class对象封装了类在方法区内的数据结构，并且向Java程序员提供了访问方法区内的数据结构的接口。

类加载器

- 启动类加载器：Bootstrap ClassLoader，负责加载存放在JDK\jre\lib(JDK代表JDK的安装目录，下同)下，或被-Xbootclasspath参数指定的路径中的，并且能被虚拟机识别的类库
- 扩展类加载器：Extension ClassLoader，该加载器由sun.misc.Launcher$ExtClassLoader实现，它负责加载DK\jre\lib\ext目录中，或者由java.ext.dirs系统变量指定的路径中的所有类库（如javax.*开头的类），开发者可以直接使用扩展类加载器。
- 应用程序类加载器：Application ClassLoader，该类加载器由sun.misc.Launcher$AppClassLoader来实现，它负责加载用户类路径（ClassPath）所指定的类，开发者可以直接使用该类加载器

双亲委派机制：类加载器收到类加载请求，自己不加载，向上委托给父类加载，父类加载不了，再自己加载。优势就是避免Java核心API篡改。

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？

自定义类加载的意义：

1. 加载特定路径的class文件
2. 加载一个加密的网络class文件
3. 热部署加载class文件

描述一下JVM加载class文件的原理机制？

JVM中类的装载是由类加载器（ClassLoader）和它的子类来实现的，Java中的类加载器是一个重要的Java运行时系统组件，它负责在运行时查找和装入类文件中的类。

由于Java的跨平台性，经过编译的Java源程序并不是一个可执行程序，而是一个或多个类文件。当Java程序需要使用某个类时，JVM会确保这个类已经被加载、连接（验证、准备和解析）和初始化。

类的加载是指把类的.class文件中的数据读入到内存中，通常是创建一个字节数组读入.class文件，然后产生与所加载类对应的Class对象。加载完成后，Class对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。最后JVM对类进行初始化，包括：1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；2)如果类中存在初始化语句，就依次执行这些初始化语句。类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader的子类）。从Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。PDM更好的保证了Java平台的安全性，在该机制中，JVM自带的Bootstrap是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM不会向Java程序提供对Bootstrap的引用。

下面是关于几个类加载器的说明：

- ootstrap：一般用本地代码实现，负责加载JVM基础核心类库（rt.jar）；
- Extension：从java.ext.dirs系统属性所指定的目录中加载类库，它的父加载器是Bootstrap；
- System：又叫应用类加载器，其父类是Extension。它是应用最广泛的类加载器。它从环境变量classpath或者系统属性java.class.path所指定的目录中记载类，是用户自定义加载器的默认父加载器。

Java对象创建过程

1. JVM遇到一条新建对象的指令时首先去检查这个指令的参数是否能在常量池中定义到一个类的符号引用。然后加载这个类（类加载过程在后边讲）
2. 为对象分配内存。一种办法“指针碰撞”、一种办法“空闲列表”，最终常用的办法“本地线程缓冲分配(TLAB)”
3. 将除对象头外的对象内存空间初始化为0
4. 对对象头进行必要设置

类的生命周期

类的生命周期包括这几个部分，加载、连接、初始化、使用和卸载，其中前三部是类的加载的过程,如下图：

- 加载，查找并加载类的二进制数据，在Java堆中也创建一个java.lang.Class类的对象
- 连接，连接又包含三块内容：验证、准备、初始化。 1）验证，文件格式、元数据、字节码、符号引用验证；2）准备，为类的静态变量分配内存，并将其初始化为默认值； 3）解析，把类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
- 使用，new出对象程序中使用
- 卸载，执行垃圾回收

Java 中会存在内存泄漏吗，请简单描述。

理论上Java因为有垃圾回收机制（GC）不会存在内存泄露问题（这也是Java被广泛使用于服务器端编程的一个重要原因）；然而在实际开发中，可能会存在无用但可达的对象，这些对象不能被GC回收，因此也会导致内存泄露的发生。例如hibernate的Session（一级缓存）中的对象属于持久态，垃圾回收器是不会回收这些对象的，然而这些对象中可能存在无用的垃圾对象，如果不及时关闭（close）或清空（flush）一级缓存就可能导致内存泄露。下面例子中的代码也会导致内存泄露。

```
1. import java.util.Arrays;
2. import java.util.EmptyStackException;
3.
4. public class MyStack<T> {
5.     private T[] elements;
6.     private int size = 0;
7.
8.     private static final int INIT_CAPACITY = 16;
9.
10.     public MyStack() {
11.         elements = (T[]) new Object[INIT_CAPACITY];
12.     }
13.
14.     public void push(T elem) {
15.         ensureCapacity();
16.         elements[size++] = elem;
17.     }
18.
19.     public T pop() {
20.         if(size == 0)
21.             throw new EmptyStackException();
22.         return elements[--size];
23.     }
24.
25.     private void ensureCapacity() {
26.         if(elements.length == size) {
27.             elements = Arrays.copyOf(elements, 2 * size + 1);
28.         }
29.     }
30. }
```

上面的代码实现了一个栈（先进后出（FILO））结构，乍看之下似乎没有什么明显的问题，它甚至可以通过你编写的各种单元测试。
然而其中的pop方法却存在内存泄露的问题，当我们用pop方法弹出栈中的对象时，该对象不会被当作垃圾回收，即使

使用栈的程序不再引用这些对象，因为栈内部维护着对这些对象的过期引用（obsolete reference）。在支持垃圾回收的语言中，内存泄露是很隐蔽的，这种内存泄露其实就是无意识的对象保持。
如果一个对象引用被无意识的保留起来了，那么垃圾回收器不会处理这个对象，也不会处理该对象引用的其他对象，即使这样的对象只有少数几个，也可能会导致很多的对象被排除在垃圾回收之外，从而对性能造成重大影响，极端情况下会引发Disk Paging（物理内存与硬盘的虚拟内存交换数据），甚至造成OutOfMemoryError。

GC是什么？为什么要有GC？

GC是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java提供的GC功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java语言没有提供释放已分配内存的显示操作方法。
Java程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：System.gc() 或Runtime.getRuntime().gc() ，但JVM可以屏蔽掉显示的垃圾回收调用。
垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。
在Java诞生初期，垃圾回收是Java最大的亮点之一，因为服务器端的编程需要有效的防止内存泄露问题，然而时过境迁，如今Java的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得iOS的系统比Android系统有更好的用户体验，其中一个深层次的原因就在于Android系统中垃圾回收的不可预知性。

补充：垃圾回收机制有很多种，包括：分代复制垃圾回收、标记垃圾回收、增量垃圾回收等方式。标准的Java进程既有栈又有堆。栈保存了原始型局部变量，堆保存了要创建的对象。Java平台对堆内存回收和再利用的基本算法被称为标记和清除，但是Java对其进行了改进，采用“分代式垃圾收集”。这种方法会跟Java对象的生命周期将堆内存划分为不同的区域，在垃圾收集过程中，可能会将对象移动到不同区域：

- 伊甸园（Eden）：这是对象最初诞生的区域，并且对大多数对象来说，这里是它们唯一存在过的区域。
- 幸存者乐园（Survivor）：从伊甸园幸存下来的对象会被挪到这里。
- 终身颐养园（Tenured）：这是足够老的幸存对象的归宿。年轻代收集（Minor-GC）过程是不会触及这个地方的。当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC），这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。

与垃圾回收相关的JVM参数：

- -Xms / -Xmx — 堆的初始大小 / 堆的最大大小
- -Xmn — 堆中年轻代的大小
- -XX:-DisableExplicitGC — 让System.gc()不产生任何作用
- -XX:+PrintGCDetails — 打印GC的细节
- -XX:+PrintGCDateStamps — 打印GC操作的时间戳
- -XX:NewSize / XX:MaxNewSize — 设置新生代大小/新生代最大大小
- -XX:NewRatio — 可以设置老生代和新生代的比例
- -XX:PrintTenuringDistribution — 设置每次新生代GC后输出幸存者乐园中对象年龄的分布
- -XX:InitialTenuringThreshold / -XX:MaxTenuringThreshold: 设置老年代阀值的初始值和最大值
- -XX:TargetSurvivorRatio: 设置幸存区的目标使用率

做GC时，一个对象在内存各个Space中被移动的顺序是什么？

标记清除法，复制算法，标记整理、分代算法。

新生代一般采用复制算法 GC，老年代使用标记整理算法。

垃圾收集器：串行新生代收集器、串行老生代收集器、并行新生代收集器、并行老年代收集器。

CMS（Current Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器，它是一种并发收集器，采用的是Mark-Sweep算法。

详见 Java GC机制。

你知道哪些垃圾回收算法？

GC最基础的算法有三种： 标记 -清除算法、复制算法、标记-压缩算法，我们常用的垃圾回收器一般都采用分代收集算法。

- 标记-清除算法，“标记-清除”（Mark-Sweep）算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。
- 复制算法，“复制”（Copying）的收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。
- 标记-压缩算法，标记过程仍然与“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都向一端移动，然后直接清理掉端边界以外的内存
- 分代收集算法，“分代收集”（Generational Collection）算法，把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。

更详细的内容参见深入理解垃圾回收算法： `http://blog.csdn.net/dd864140130/article/details/50084471`

垃圾回收器

- Serial收集器，串行收集器是最古老，最稳定以及效率高的收集器，可能会产生较长的停顿，只使用一个线程去回收。
- ParNew收集器，ParNew收集器其实就是Serial收集器的多线程版本。
- Parallel收集器，Parallel Scavenge收集器类似ParNew收集器，Parallel收集器更关注系统的吞吐量。
- Parallel Old 收集器，Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记－整理”算法
- CMS收集器，CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。
- G1收集器，G1 (Garbage-First)是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足GC停顿时间要求的同时,还具备高吞吐量性能特征

如何判断一个对象是否应该被回收

判断对象是否存活一般有两种方式：

- 引用计数：每个对象有一个引用计数属性，新增一个引用时计数加1，引用释放时计数减1，计数为0时可以回收。此方法简单，无法解决对象相互循环引用的问题。

- 可达性分析（Reachability Analysis）：从GC Roots开始向下搜索，搜索所走过的路径称为引用链。当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的，不可达对象。

JVM的永久代中会发生垃圾回收么？

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免Full GC是非常重要的原因。请参考下Java8：从永久代到元数据区 (注：Java8中已经移除了永久代，新加了一个叫做元数据区的native内存区)

引用的分类

- 强引用：GC时不会被回收
- 软引用：描述有用但不是必须的对象，在发生内存溢出异常之前被回收
- 弱引用：描述有用但不是必须的对象，在下一次GC时被回收
- 虚引用（幽灵引用/幻影引用）:无法通过虚引用获得对象，用PhantomReference实现虚引用，虚引用用来在GC时返回一个通知。

调优命令

Sun JDK监控和故障处理命令有jps jstat jmap jhat jstack jinfo

- jps，JVM Process Status Tool,显示指定系统内所有的HotSpot虚拟机进程。
- jstat，JVM statistics Monitoring是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。
- jmap，JVM Memory Map命令用于生成heap dump文件
- jhat，JVM Heap Analysis Tool命令是与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看
- jstack，用于生成java虚拟机当前时刻的线程快照。
- jinfo，JVM Configuration info 这个命令作用是实时查看和调整虚拟机运行参数。

调优工具

常用调优工具分为两类,jdk自带监控工具：jconsole和jvisualvm，第三方有：MAT(Memory Analyzer Tool)、GChisto。

- jconsole，Java Monitoring and Management Console是从java5开始，在JDK中自带的java监控和管理控制台，用于对JVM中内存，线程和类等的监控
- jvisualvm，jdk自带全能工具，可以分析内存快照、线程快照；监控内存变化、GC变化等。
- MAT，Memory Analyzer Tool，一个基于Eclipse的内存分析工具，是一个快速、功能丰富的Java heap分析工具，它可以帮助我们查找内存泄漏和减少内存消耗
- GChisto，一款专业分析gc日志的工具

jstack 是干什么的？ jstat 呢？如果线上程序周期性地出

现卡顿，你怀疑可 能是 GC 导致的，你会怎么来排查这个问题？线程日志一般你会看其中的什么 部分？

jstack 用来查询 Java 进程的堆栈信息。

jvisualvm 监控内存泄露，跟踪垃圾回收、执行时内存、cpu分析、线程分析。

详见Java jvisualvm简要说明，可参考 线上FullGC频繁的排查。

Minor GC与Full GC分别在什么时候发生？

新生代内存不够用时候发生MGC也叫YGC，JVM内存不够的时候发生FGC

对象头，详细讲下

- 【Java对象解析】不得不了解的对象头
- JVM源码分析之java对象头实现
- JVM—深入分析对象的内存布局

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？

参考：参考 触发JVM进行Full GC的情况及应对策略

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？

permgen space、heap space 错误。

常见的原因

- 内存加载的数据量太大：一次性从数据库取太多数据；
- 集合类中有对对象的引用，使用后未清空，GC不能进行回收；
- 代码中存在循环产生过多的重复对象；
- 启动参数堆内存值小。

详见 Java 内存溢出（java.lang.OutOfMemoryError）的常见情况和处理方式总结。

JDK 1.8之后Perm Space有哪些变动？ MetaSpace大小默认是无限的么？ 还是你们会通过什么方式来指定大小？

JDK 1.8后用元空间替代了 Perm Space；字符串常量存放到堆内存中。

MetaSpace大小默认没有限制，一般根据系统内存的大小。JVM会动态改变此值。

-XX:MetaspaceSize: 分配给类元数据空间（以字节计）的初始大小（Oracle逻辑存储上的初始高水位，the initial high-water-mark）。此值为估计值，MetaspaceSize的值设置的过大会延长垃圾回收时间。垃圾回收过后，引起下一次垃圾回收的类元数据空间的大小可能会变大。

-XX:MaxMetaspaceSize: 分配给类元数据空间的最大值，超过此值就会触发Full GC，此值默认没有限制，但应取决于系统内存的大小。JVM会动态地改变此值。

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？

栈内存溢出，一般由栈内存的局部变量过爆了，导致内存溢出。出现在递归方法，参数个数过多，递归过深，递归没有出口。

IO

NIO

解释一下java.io.Serializable接口

类通过实现 Java.io.Serializable 接口以启用其序列化功能。未实现此接口的类将无法使其任何状态序列化或反序列化。

IO操作最佳实践

- 使用有缓冲的IO类,不要单独读取字节或字符
- 使用NIO和NIO 2或者AIO,而非BIO
- 在finally中关闭流
- 使用内存映射文件获取更快的IO

Java IO 分类

- Java BIO： 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。
- Java NIO ： 同步非阻塞，服务器实现模式为一个请求一个线程，即当一个连接创建后，不需要对应一个线程，这个连接会被注册到多路复用器上面，所以所有的连接只需要一个线程就可以搞定，当这个线程中的多路复用器进行轮询的时候，发现连接上有请求的话，才开启一个线程进行处理，也就是一个请求一个线程模式。BIO与NIO一个比较重要的不同，是我们使用BIO的时候往往会引入多线程，每个连接一个单独的线程；而NIO则是使用单线程或者只使用少量的多线程，每个连接共用一个线程。
- Java AIO(NIO.2) ： 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理。

说出 5 条 IO 的最佳实践

IO 对 Java 应用的性能非常重要。理想情况下，你不应该在你应用的关键路径上避免 IO 操作。下面是一些你应该遵循的 Java IO 最佳实践:

- 使用有缓冲区的 IO 类，而不要单独读取字节或字符。
- 使用 NIO 和 NIO2
- 在 finally 块中关闭流，或者使用 try-with-resource 语句。
- 使用内存映射文件获取更快的 IO。

BIO、NIO、AIO适用场景分析

- BIO（同步并阻塞）方式适用于连接数目比较小且固定的架构，这种方式对服务器资源要求比较高，并发局限于

应用中，JDK1.4以前的唯一选择，但程序直观简单易理解。

- NIO（同步非阻塞）方式适用于连接数目多且连接比较短（轻操作）的架构，比如聊天服务器，并发局限于应用中，编程比较复杂，JDK1.4开始支持。
- AIO（ 异步非阻塞）方式使用于连接数目多且连接比较长（重操作）的架构，比如相册服务器，充分调用OS参与并发操作，编程比较复杂，JDK7开始支持。

Java NIO和IO的主要区别

1. 面向流与面向缓冲.
   Java NIO和IO之间第一个最大的区别是，IO是面向流的，NIO是面向缓冲区的。Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。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。
2. 阻塞与非阻塞IO
   Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。 Java NIO的非阻塞模式，使一个线程从某通道发送请求读取数据，但是它仅能得到目前可用的数据，如果目前没有数据可用时，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）。
3. 选择器（Selectors）
   Java NIO的选择器允许一个单独的线程来监视多个输入通道，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

Java I/O库的两个设计模式

Java I/O库的总体设计是符合装饰模式和适配器模式的。如前所述，这个库中处理流的类叫流类。

装饰模式（Decorator）：在由InputStream、OutputStream、Reader和Writer代表的等级结构内部，有一些流处理器可以对另一些流处理器起到装饰作用，形成新的、具有改善了的功能的流处理器。

适配器模式（Adapter）：在由InputStream、OutputStream、Reader和Writer代表的等级结构内部，有一些流处理器是对其他类型的流处理器的适配。这就是适配器的应用。

设计模式

设计模式

简述一下你了解的设计模式。

所谓设计模式，就是一套被反复使用的代码设计经验的总结（情境中一个问题经过证实的一个解决方案）。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。设计模式使人们可以更加简单方便的复用成功的设计和体系结构。将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。

在GoF的《Design Patterns: Elements of Reusable Object-Oriented Software》中给出了三类：

- 创建型[对类的实例化过程的抽象化]
- 结构型[描述如何将类或对象结合在一起形成更大的结构]
- 行为型[对在不同的对象之间划分责任和算法的抽象化]

共23种设计模式，包括：

- Abstract Factory（抽象工厂模式）
- Builder（建造者模式）
- Factory Method（工厂方法模式）
- Prototype（原始模型模式）
- Singleton（单例模式）
- Facade（门面模式）
- Adapter（适配器模式）
- Bridge（桥梁模式）
- Composite（合成模式）
- Decorator（装饰模式）
- Flyweight（享元模式）
- Proxy（代理模式）
- Command（命令模式）
- Interpreter（解释器模式）
- Visitor（访问者模式）
- Iterator（迭代子模式）
- Mediator（调停者模式）
- Memento（备忘录模式）
- Observer（观察者模式）
- State（状态模式）
- Strategy（策略模式）
- Template Method（模板方法模式）
- Chain Of Responsibility（责任链模式）

面试被问到关于设计模式的知识时，可以拣最常用的作答，例如：

- 工厂模式：工厂类可以根据条件生成不同的子类实例，这些子类有一个公共的抽象父类并且实现了相同的方法，但是这些方法针对不同的数据进行了不同的操作（多态方法）。当得到子类的实例后，开发人员可以调用基类中的方法而不必考虑到底返回的是哪一个子类的实例。
- 代理模式：给一个对象提供一个代理对象，并由代理对象控制原对象的引用。实际开发中，按照使用目的的不同，代理可以分为：远程代理、虚拟代理、保护代理、Cache代理、防火墙代理、同步化代理、智能引用代理。
- 适配器模式：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起使用的类能够一起工作。
- 模板方法模式：提供一个抽象类，将部分逻辑以具体方法或构造器的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法（多态实现），从而实现不同的业务逻辑。

除此之外，还可以讲讲上面提到的门面模式、桥梁模式、单例模式、装潢模式（Collections工具类和I/O系统中都使用装潢模式）等，反正基本原则就是拣自己最熟悉的、用得最多的作答，以免言多必失。

设计模式在实际场景中的应用

Spring中用到了哪些设计模式

MyBatis中用到了哪些设计模式

你项目中有使用哪些设计模式

说说常用开源框架中设计模式使用分析

数据结构与算法

算法

算法

一致性Hash算法

参考:

- 一致性Hash算法
- 一致 Hash 算法

九种内部排序算法的Java实现及其性能测试

参考: 九种内部排序算法的Java实现及其性能测试

排序算法汇总

参考: 排序算法汇总

查找算法

参考: 查找算法

限流算法

参考: 限流算法

深度有限算法

广度优先算法

克鲁斯卡尔算法

普林母算法

迪克拉斯算法

数据结构

数据结构

树

参考：树

Hash

参考：Hash

JavaWeb

JavaWeb基础

JSP 和 Servlet

Servlet接口中有哪些方法？

Servlet接口定义了5个方法，其中前三个方法与Servlet生命周期相关：

- void init(ServletConfig config) throws ServletException
- void service(ServletRequest req, ServletResponse resp) throws ServletException, java.io.IOException
- void destory()
- java.lang.String getServletInfo()
- ServletConfig getServletConfig()

Web容器加载Servlet并将其实例化后，Servlet生命周期开始，容器运行其init()方法进行Servlet的初始化；请求到达时调用Servlet的service()方法，service()方法会根据需要调用与请求对应的doGet或doPost等方法；当服务器关闭或项目被卸载时服务器会将Servlet实例销毁，此时会调用Servlet的destroy()方法。

转发（forward）和重定向（redirect）的区别？

forward是容器中控制权的转向，是服务器请求资源，服务器直接访问目标地址的URL，把那个URL 的响应内容读取过来，然后把这些内容再发给浏览器，浏览器根本不知道服务器发送的内容是从哪儿来的，所以它的地址栏中还是原来的地址。

redirect就是服务器端根据逻辑，发送一个状态码，告诉浏览器重新去请求那个地址，因此从浏览器的地址栏中可以看到跳转后的链接地址，很明显redirect无法访问到服务器保护起来资源，但是可以从一个网站redirect到其他网站。

forward更加高效，所以在满足需要时尽量使用forward（通过调用RequestDispatcher对象的forward()方法，该对象可以通过ServletRequest对象的getRequestDispatcher()方法获得），并且这样也有助于隐藏实际的链接；在有些情况下，比如需要访问一个其它服务器上的资源，则必须使用重定向（通过HttpServletResponse对象调用其sendRedirect()方法实现）。

JSP有哪些内置对象？作用分别是什么？

JSP有9个内置对象：

- request：封装客户端的请求，其中包含来自GET或POST请求的参数；
- response：封装服务器对客户端的响应；
- pageContext：通过该对象可以获取其他对象；
- session：封装用户会话的对象；
- application：封装服务器运行环境的对象；

- out：输出服务器响应的输出流对象；
- config：Web应用的配置对象；
- page：JSP页面本身（相当于Java程序中的this）；
- exception：封装页面抛出异常的对象。

JSP和Servlet是什么关系？

ervlet是一个特殊的Java程序，它运行于服务器的JVM中，能够依靠服务器的支持向浏览器提供显示内容。JSP本质上是Servlet的一种简易形式，JSP会被服务器处理成一个类似于Servlet的Java程序，可以简化页面内容的生成。Servlet和JSP最主要的不同点在于，Servlet的应用逻辑是在Java文件中，并且完全从表示层中的HTML分离开来。而JSP的情况是Java和HTML可以组合成一个扩展名为.jsp的文件。有人说，Servlet就是在Java中写HTML，而JSP就是在HTML中写Java代码，当然这个说法是很片面且不够准确的。JSP侧重于视图，Servlet更侧重于控制逻辑，在MVC架构模式中，JSP适合充当视图（view）而Servlet适合充当控制器（controller）。

讲解JSP中的四种作用域。

答：JSP中的四种作用域包括page、request、session和application，具体来说：

- page代表与一个页面相关的对象和属性。
- request代表与Web客户机发出的一个请求相关的对象和属性。一个请求可能跨越多个页面，涉及多个Web组件；需要在页面显示的临时数据可以置于此作用域。
- session代表与某个用户与服务器建立的一次会话相关的对象和属性。跟某个用户相关的数据应该放在用户自己的session中。
- application代表与整个Web应用程序相关的对象和属性，它实质上是跨越整个Web应用程序，包括多个页面、请求和会话的一个全局作用域。

实现会话跟踪的技术有哪些？

由于HTTP协议本身是无状态的，服务器为了区分不同的用户，就需要对用户会话进行跟踪，简单的说就是为用户进行登记，为用户分配唯一的ID，下一次用户在请求中包含此ID，服务器据此判断到底是哪一个用户。

1）URL 重写：在URL中添加用户会话的信息作为请求的参数，或者将唯一的会话ID添加到URL结尾以标识一个会话。

2） 设置表单隐藏域：将和会话跟踪相关的字段添加到隐式表单域中，这些信息不会在浏览器中显示但是提交表单时会提交给服务器。

这两种方式很难处理跨越多个页面的信息传递，因为如果每次都要修改URL或在页面中添加隐式表单域来存储用户会话相关信息，事情将变得非常麻烦。

3）cookie：cookie有两种，一种是基于窗口的，浏览器窗口关闭后，cookie就没有了；另一种是将信息存储在一个临时文件中，并设置存在的时间。当用户通过浏览器和服务器建立一次会话后，会话ID就会随响应信息返回存储在基于窗口的cookie中，那就意味着只要浏览器没有关闭，会话没有超时，下一次请求时这个会话ID又会提交给服务器让服务器识别用户身份。会话中可以为用户保存信息。会话对象是在服务器内存中的，而基于窗口的cookie是在客户端内存中的。如果浏览器禁用了cookie，那么就需要通过下面两种方式进行会话跟踪。当然，在使用cookie时要注意几点：首先不要在cookie中存放敏感信息；其次cookie存储的数据量有限（4k），不能将过多的内容存储cookie中；再者浏览器通常只允许一个站点最多存放20个cookie。当然，和用户会话相关的其他信息（除了会话

ID）也可以存在cookie方便进行会话跟踪。

4）HttpSession：在所有会话跟踪技术中，HttpSession对象是最强大也是功能最多的。当一个用户第一次访问某个网站时会自动创建HttpSession，每个用户可以访问他自己的HttpSession。可以通过HttpServletRequest对象的getSession方法获得HttpSession，通过HttpSession的setAttribute方法可以将一个值放在HttpSession中，通过调用HttpSession对象的getAttribute方法，同时传入属性名就可以获取保存在HttpSession中的对象。与上面三种方式不同的是，HttpSession放在服务器的内存中，因此不要将过大的对象放在里面，即使目前的Servlet容器可以在内存将满时将HttpSession中的对象移到其他存储设备中，但是这样势必影响性能。添加到HttpSession中的值可以是任意Java对象，这个对象最好实现了Serializable接口，这样Servlet容器在必要的时候可以将其序列化到文件中，否则在序列化时就会出现异常。

过滤器有哪些作用和用法？

Java Web开发中的过滤器（filter）是从Servlet 2.3规范开始增加的功能，并在Servlet 2.4规范中得到增强。对Web应用来说，过滤器是一个驻留在服务器端的Web组件，它可以截取客户端和服务器之间的请求与响应信息，并对这些信息进行过滤。当Web容器接受到一个对资源的请求时，它将判断是否有过滤器与这个资源相关联。如果有，那么容器将把请求交给过滤器进行处理。在过滤器中，你可以改变请求的内容，或者重新设置请求的报头信息，然后再将请求发送给目标资源。当目标资源对请求作出响应时候，容器同样会将响应先转发给过滤器，在过滤器中你可以对响应的内容进行转换，然后再将响应发送到客户端。

常见的过滤器用途主要包括：对用户请求进行统一认证、对用户的访问请求进行记录和审核、对用户发送的数据进行过滤或替换、转换图象格式、对响应内容进行压缩以减少传输量、对请求或响应进行加解密处理、触发资源访问事件、对XML的输出应用XSLT等。

过滤器相关的接口主要有：Filter、FilterConfig和FilterChain。

监听器有哪些作用和用法？

Java Web开发中的监听器（listener）就是application、session、request三个对象创建、销毁或者往其中添加修改删除属性时自动执行代码的功能组件，如下所示：

- ServletContextListener：对Servlet上下文的创建和销毁进行监听。
- ervletContextAttributeListener：监听Servlet上下文属性的添加、删除和替换。
- HttpSessionAttributeListener：对Session对象中属性的添加、删除和替换进行监听。
- ServletRequestListener：对请求对象的初始化和销毁进行监听。
- ServletRequestAttributeListener：对请求对象属性的添加、删除和替换进行监听。
- HttpSessionListener：对Session的创建和销毁进行监听。

补充： session的销毁有两种情况：

- session超时（可以在web.xml中通过`<session-config>/<session-timeout>`标签配置超时时间）；
- 通过调用session对象的invalidate()方法使session失效。

Servlet的生命周期

Spring系列

Spring

什么是Spring？

Spring是一个开源的Java EE开发框架。Spring框架的核心功能可以应用在任何Java应用程序中，但对Java EE平台上的Web应用程序有更好的扩展性。Spring框架的目标是使得Java EE应用程序的开发更加简捷，通过使用POJO为基础的编程模型促进良好的编程风格。

Spring有哪些优点？

轻量级：Spring在大小和透明性方面绝对属于轻量级的，基础版本的Spring框架大约只有2MB。

控制反转(IOC)：Spring使用控制反转技术实现了松耦合。依赖被注入到对象，而不是创建或寻找依赖对象。

面向切面编程(AOP)： Spring支持面向切面编程，同时把应用的业务逻辑与系统的服务分离开来。

容器：Spring包含并管理应用程序对象的配置及生命周期。

MVC框架：Spring的web框架是一个设计优良的web MVC框架，很好的取代了一些web框架。

事务管理：Spring对下至本地业务上至全局业务(JAT)提供了统一的事务管理接口。

异常处理：Spring提供一个方便的API将特定技术的异常(由JDBC, Hibernate, 或JDO抛出)转化为一致的、Unchecked异常。

Spring 事务实现方式

- 编程式事务管理：这意味着你可以通过编程的方式管理事务，这种方式带来了很大的灵活性，但很难维护。
- 声明式事务管理：这种方式意味着你可以将事务管理和业务代码分离。你只需要通过注解或者XML配置管理事务。

Spring框架的事务管理有哪些优点

- 它为不同的事务API(如JTA, JDBC, Hibernate, JPA, 和JDO)提供了统一的编程模型。
- 它为编程式事务管理提供了一个简单的API而非一系列复杂的事务API(如JTA).
- 它支持声明式事务管理。
- 它可以和Spring 的多种数据访问技术很好的融合。

spring事务定义的传播规则

- PROPAGATION_REQUIRED: 支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。
- PROPAGATION_SUPPORTS: 支持当前事务，如果当前没有事务，就以非事务方式执行。
- PROPAGATION_MANDATORY: 支持当前事务，如果当前没有事务，就抛出异常。
- PROPAGATION_REQUIRES_NEW: 新建事务，如果当前存在事务，把当前事务挂起。
- PROPAGATION_NOT_SUPPORTED: 以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。
- PROPAGATION_NEVER: 以非事务方式执行，如果当前存在事务，则抛出异常。
- PROPAGATION_NESTED: 如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与PROPAGATION_REQUIRED类似的操作。

Spring 事务底层原理

- 划分处理单元—IoC

由于spring解决的问题是对单个数据库进行局部事务处理的，具体的实现首先用spring中的IoC划分了事务处理单元。并且将对事务的各种配置放到了ioc容器中（设置事务管理器，设置事务的传播特性及隔离机制）。

- AOP拦截需要进行事务处理的类

Spring事务处理模块是通过AOP功能来实现声明式事务处理的，具体操作（比如事务实行的配置和读取，事务对象的抽象），用TransactionProxyFactoryBean接口来使用AOP功能，生成proxy代理对象，通过TransactionInterceptor完成对代理方法的拦截，将事务处理的功能编织到拦截的方法中。读取ioc容器事务配置属性，转化为spring事务处理需要的内部数据结构（TransactionAttributeSourceAdvisor），转化为TransactionAttribute表示的数据对象。

- 对事务处理实现（事务的生成、提交、回滚、挂起）

spring委托给具体的事务处理器实现。实现了一个抽象和适配。适配的具体事务处理器：DataSource数据源支持、hibernate数据源事务处理支持、JDO数据源事务处理支持，JPA、JTA数据源事务处理支持。这些支持都是通过设计PlatformTransactionManager、AbstractPlatforTransaction一系列事务处理的支持。 为常用数据源支持提供了一系列的TransactionManager。

- 结合

PlatformTransactionManager实现了TransactionInterception接口，让其与TransactionProxyFactoryBean结合起来，形成一个Spring声明式事务处理的设计体系。

有没有遇到过Spring事务失效的情况？在什么情况下Spring的事务是失效的？

参考：面试必备技能：JDK动态代理给Spring事务埋下的坑！

Spring MVC 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

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)）

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

BeanFactory和ApplicationContext有什么区别？

ApplicationContext提供了一种解决文档信息的方法，一种加载文件资源的方式(如图片)，他们可以向监听他们的beans发送消息。另外，容器或者容器中beans的操作，这些必须以bean工厂的编程方式处理的操作可以在应用上下文中以声明的方式处理。应用上下文实现了MessageSource，该接口用于获取本地消息，实际的实现是可选的。

相同点：两者都是通过xml配置文件加载bean,ApplicationContext和BeanFacotry相比,提供了更多的扩展功能。

不同点：BeanFactory是延迟加载,如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常；而ApplicationContext则在初始化自身是检验，这样有利于检查所依赖属性是否注入；所以通常情况下我们选择使用ApplicationContext。

什么是Spring Beans？

Spring Beans是构成Spring应用核心的Java对象。这些对象由Spring IOC容器实例化、组装、管理。这些对象通过容器中配置的元数据创建，例如，使用XML文件中定义的创建。

在Spring中创建的beans都是单例的beans。在bean标签中有一个属性为”singleton”,如果设为true，该bean是单例的，如果设为false，该bean是原型bean。Singleton属性默认设置为true。因此，spring框架中所有的bean都默认为单例bean。

说一下Spring中支持的bean作用域

Spring框架支持如下五种不同的作用域:

- singleton: 在Spring IOC容器中仅存在一个Bean实例，Bean以单实例的方式存在。
- prototype: 一个bean可以定义多个实例。
- request: 每次HTTP请求都会创建一个新的Bean。该作用域仅适用于WebApplicationContext环境。

- session：一个HTTP Session定义一个Bean。该作用域仅适用于WebApplicationContext环境。
- globalSession：同一个全局HTTP Session定义一个Bean。该作用域同样仅适用于WebApplicationContext环境。

bean默认的scope属性是”singleton”。

Spring 的单例实现原理

Spring框架对单例的支持是采用单例注册表的方式进行实现的，而这个注册表的缓存是HashMap对象，如果配置文件中的配置信息不要求使用单例，Spring会采用新建实例的方式返回对象实例。

解释Spring框架中bean的生命周期

ApplicationContext容器中，Bean的生命周期流程如上图所示，流程大致如下：

1.首先容器启动后，会对scope为singleton且非懒加载的bean进行实例化，

2.按照Bean定义信息配置信息，注入所有的属性，

3.如果Bean实现了BeanNameAware接口，会回调该接口的setBeanName()方法，传入该Bean的id，此时该Bean就获得了自己在配置文件中的id，

4.如果Bean实现了BeanFactoryAware接口,会回调该接口的setBeanFactory()方法，传入该Bean的BeanFactory，这样该Bean就获得了自己所在的BeanFactory，

5.如果Bean实现了ApplicationContextAware接口,会回调该接口的setApplicationContext()方法，传入该Bean的ApplicationContext，这样该Bean就获得了自己所在的ApplicationContext，

6.如果有Bean实现了BeanPostProcessor接口，则会回调该接口的postProcessBeforeInitialzation()方法，

7.如果Bean实现了InitializingBean接口，则会回调该接口的afterPropertiesSet()方法，

8.如果Bean配置了init-method方法，则会执行init-method配置的方法，

9.如果有Bean实现了BeanPostProcessor接口，则会回调该接口的postProcessAfterInitialization()方法，

10.经过流程9之后，就可以正式使用该Bean了,对于scope为singleton的Bean,Spring的ioc容器中会缓存一份该bean的实例，而对于scope为prototype的Bean,每次被调用都会new一个新的对象，期生命周期就交给调用方管理了，不再是Spring容器进行管理了

11.容器关闭后，如果Bean实现了DisposableBean接口，则会回调该接口的destroy()方法，

12.如果Bean配置了destroy-method方法，则会执行destroy-method配置的方法，至此，整个Bean的生命周期结束

Resource 是如何被查找、加载的？

Resource 接口是 Spring 资源访问策略的抽象，它本身并不提供任何资源访问实现，具体的资源访问由该接口的实现类完成—每个实现类代表一种资源访问策略。 Spring 为 Resource 接口提供了如下实现类：

- UrlResource：访问网络资源的实现类。
- ClassPathResource：访问类加载路径里资源的实现类。
- FileSystemResource：访问文件系统里资源的实现类。
- ServletContextResource：访问相对于 ServletContext 路径里的资源的实现类：
- InputStreamResource：访问输入流资源的实现类。
- ByteArrayResource：访问字节数组资源的实现类。 这些 Resource 实现类，针对不同的的底层资源，提供了相应的资源访问逻辑，并提供便捷的包装，以利于客户端程序的资源访问。

解释自动装配的各种模式？

自动装配提供五种不同的模式供Spring容器用来自动装配beans之间的依赖注入：

no：默认的方式是不进行自动装配，通过手工设置ref 属性来进行装配bean。

byName：通过参数名自动装配，Spring容器查找beans的属性，这些beans在XML配置文件中被设置为byName。之后容器试图匹配、装配和该bean的属性具有相同名字的bean。

byType：通过参数的数据类型自动自动装配，Spring容器查找beans的属性，这些beans在XML配置文件中被设置为byType。之后容器试图匹配和装配和该bean的属性类型一样的bean。如果有多个bean符合条件，则抛出错误。

constructor：这个同byType类似，不过是应用于构造函数的参数。如果在BeanFactory中不是恰好有一个bean与构造函数参数相同类型，则抛出一个严重的错误。

autodetect：如果有默认的构造方法，通过 construct的方式自动装配，否则使用 byType的方式自动装配。

Spring中的依赖注入是什么？

依赖注入作为控制反转(IOC)的一个层面，可以有多种解释方式。在这个概念中，你不用创建对象而只需要描述如何创建它们。你不必通过代码直接的将组件和服务连接在一起，而是通过配置文件说明哪些组件需要什么服务。之后IOC容器负责衔接。

有哪些不同类型的IOC(依赖注入)？

构造器依赖注入：构造器依赖注入在容器触发构造器的时候完成，该构造器有一系列的参数，每个参数代表注入的对象。

Setter方法依赖注入：首先容器会触发一个无参构造函数或无参静态工厂方法实例化对象，之后容器调用bean中的setter方法完成Setter方法依赖注入。

你推荐哪种依赖注入？构造器依赖注入还是Setter方法依赖注

入？

你可以同时使用两种方式的依赖注入，最好的选择是使用构造器参数实现强制依赖注入，使用setter方法实现可选的依赖关系。

Spring IOC 如何实现

Spring中的 org.springframework.beans 包和 org.springframework.context包构成了Spring框架IoC容器的基础。

BeanFactory 接口提供了一个先进的配置机制，使得任何类型的对象的配置成为可能。ApplicationContex接口对BeanFactory（是一个子接口）进行了扩展，在BeanFactory的基础上添加了其他功能，比如与Spring的AOP更容易集成，也提供了处理message resource的机制（用于国际化）、事件传播以及应用层的特别配置，比如针对Web应用的WebApplicationContext。

org.springframework.beans.factory.BeanFactory 是Spring IoC容器的具体实现，用来包装和管理前面提到的各种bean。BeanFactory接口是Spring IoC 容器的核心接口。

Spring IoC容器是什么？

Spring IOC负责创建对象、管理对象(通过依赖注入)、整合对象、配置对象以及管理这些对象的生命周期。

IoC有什么优点？

IOC或依赖注入减少了应用程序的代码量。它使得应用程序的测试很简单，因为在单元测试中不再需要单例或JNDI查找机制。简单的实现以及较少的干扰机制使得松耦合得以实现。IOC容器支持勤性单例及延迟加载服务。

解释AOP模块

AOP模块用来开发Spring应用程序中具有切面性质的部分。该模块的大部分服务由AOP Aliance提供，这就保证了Spring框架和其他AOP框架之间的互操作性。另外，该模块将元数据编程引入到了Spring。

Spring面向切面编程(AOP)

面向切面编程（AOP）：允许程序员模块化横向业务逻辑，或定义核心部分的功能，例如日志管理和事务管理。

切面(Aspect) ：AOP的核心就是切面，它将多个类的通用行为封装为可重用的模块。该模块含有一组API提供cross-cutting功能。例如,日志模块称为日志的AOP切面。根据需求的不同，一个应用程序可以有若干切面。在Spring AOP中，切面通过带有@Aspect注解的类实现。

通知(Advice)：通知表示在方法执行前后需要执行的动作。实际上它是Spring AOP框架在程序执行过程中触发的一些代码。Spring切面可以执行一下五种类型的通知：

- before(前置通知)：在一个方法之前执行的通知。

- after(最终通知)：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。
- after-returning(后置通知)：在某连接点正常完成后执行的通知。
- after-throwing(异常通知)：在方法抛出异常退出时执行的通知。
- around(环绕通知)：在方法调用前后触发的通知。

切入点(Pointcut)：切入点是一个或一组连接点，通知将在这些位置执行。可以通过表达式或匹配的方式指明切入点。

引入：引入允许我们在已有的类上添加新的方法或属性。

目标对象：被一个或者多个切面所通知的对象。它通常是一个代理对象。也被称做被通知（advised）对象。

代理：代理是将通知应用到目标对象后创建的对象。从客户端的角度看，代理对象和目标对象是一样的。有以下几种代理：

- BeanNameAutoProxyCreator：bean名称自动代理创建器
- DefaultAdvisorAutoProxyCreator：默认通知者自动代理创建器
- Metadata autoproxying：元数据自动代理

织入：将切面和其他应用类型或对象连接起来创建一个通知对象的过程。织入可以在编译、加载或运行时完成。

Spring AOP 实现原理

实现AOP的技术，主要分为两大类：

- 一是采用动态代理技术，利用截取消息的方式，对该消息进行装饰，以取代原有对象行为的执行；
- 二是采用静态织入的方式，引入特定的语法创建“方面”，从而使得编译器可以在编译期间织入有关“方面”的代码。

Spring AOP 的实现原理其实很简单：AOP 框架负责动态地生成 AOP 代理类，这个代理类的方法则由 Advice和回调目标对象的方法所组成，并将该对象可作为目标对象使用。AOP 代理包含了目标对象的全部方法，但AOP代理中的方法与目标对象的方法存在差异，AOP方法在特定切入点添加了增强处理，并回调了目标对象的方法。

Spring AOP使用动态代理技术在运行期织入增强代码。使用两种代理机制：基于JDK的动态代理（JDK本身只提供接口的代理）和基于CGlib的动态代理。

- (1) JDK的动态代理

  JDK的动态代理主要涉及java.lang.reflect包中的两个类：Proxy和InvocationHandler。其中InvocationHandler只是一个接口，可以通过实现该接口定义横切逻辑，并通过反射机制调用目标类的代码，动态的将横切逻辑与业务逻辑织在一起。而Proxy利用InvocationHandler动态创建一个符合某一接口的实例，生成目标类的代理对象。

  其代理对象必须是某个接口的实现，它是通过在运行期间创建一个接口的实现类来完成对目标对象的代理.只能实现接口的类生成代理,而不能针对类

- (2)CGLib

  CGLib采用底层的字节码技术，为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类的调用方法，并顺势织入横切逻辑.它运行期间生成的代理对象是目标类的扩展子类.所以无法通知final、private的方法,

因为它们不能被覆写.是针对类实现代理,主要是为指定的类生成一个子类,覆盖其中方法.

在spring中默认情况下使用JDK动态代理实现AOP,如果proxy-target-class设置为true或者使用了优化策略那么会使用CGLIB来创建动态代理.Spring AOP在这两种方式的实现上基本一样. 以JDK代理为例, 会使用JdkDynamicAopProxy来创建代理, 在invoke()方法首先需要织入到当前类的增强器封装到拦截器链中, 然后递归的调用这些拦截器完成功能的织入. 最终返回代理对象.

`http://zhengjianglong.cn/2015/12/12/Spring/spring-source-aop/`

如何自定义注解实现功能

SpringMVC启动流程

cgLib知道吗？他和jdk动态代理什么区别？手写一个jdk动态代理呗？

MyBatis

MyBatis

#{}和${}的区别是什么？

${}是Properties文件中的变量占位符，它可以用于标签属性值和sql内部，属于静态文本替换，比如${driver}会被静态替换为com.mysql.jdbc.Driver。#{}是sql的参数占位符，Mybatis会将sql中的#{}替换为?号，在sql执行前会使用PreparedStatement的参数设置方法，按序给sql的?号占位符设置参数值，比如ps.setInt(0, parameterValue)，#{item.name}的取值方式为使用反射从参数对象中获取item对象的name属性值，相当于param.getItem().getName()。

Xml映射文件中，除了常见的select|insert|updae|delete标签之外，还有哪些标签？

还有很多其他的标签，\、\、\、\、\，加上动态sql的9个标签，trim|where|set|foreach|if|choose|when|otherwise|bind等，其中\为sql片段标签，通过\标签引入sql片段，\为不支持自增的主键生成策略标签。

最佳实践中，通常一个Xml映射文件，都会写一个Dao接口与之对应，请问，这个Dao接口的工作原理是什么？Dao接口里的方法，参数不同时，方法能重载吗？

Dao接口，就是人们常说的Mapper接口，接口的全限名，就是映射文件中的namespace的值，接口的方法名，就是映射文件中MappedStatement的id值，接口方法内的参数，就是传递给sql的参数。

Mapper接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为key值，可唯一定位一个MappedStatement，举例：com.mybatis3.mappers.StudentDao.findStudentById，可以唯一找到namespace为com.mybatis3.mappers.StudentDao下面id = findStudentById的MappedStatement。在Mybatis中，每一个\、\、\、\标签均会被解析为MappedStatement对象，标签内的sql会被解析为BoundSql对象。

为什么说Mybatis是半自动ORM映射工具？它与全自动的区别在哪里？

Hibernate属于全自动ORM映射工具，使用Hibernate查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而Mybatis在查询关联对象或关联集合对象时，需要手动编写sql来完成，所以，称之为半自动ORM映射工具。

简单的说一下MyBatis的一级缓存和二级缓存？

Mybatis首先去缓存中查询结果集，如果没有则查询数据库，如果有则从缓存取出返回结果集就不走数据库。Mybatis内部存储缓存使用一个HashMap，key为hashCode+sqlId+Sql语句。value为从查询出来映射生成的java对象

Mybatis的二级缓存即查询缓存，它的作用域是一个mapper的namespace，即在同一个namespace中查询sql可以从缓存中获取数据。二级缓存是可以跨SqlSession的。

数据库与缓存

数据库

数据库基本理论

数据库基础

数据库范式

第一范式：列不可分，eg:【联系人】（姓名，性别，电话），一个联系人有家庭电话和公司电话，那么这种表结构设计就没有达到 1NF；

第二范式：有主键，保证完全依赖。eg:订单明细表【OrderDetail】（OrderID，ProductID，UnitPrice，Discount，Quantity，ProductName），Discount（折扣），Quantity（数量）完全依赖（取决）于主键（OderID，ProductID），而 UnitPrice，ProductName 只依赖于 ProductID，不符合2NF；

第三范式：无传递依赖(非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况)，eg:订单表【Order】（OrderID，OrderDate，CustomerID，CustomerName，CustomerAddr，CustomerCity）主键是（OrderID），CustomerName，CustomerAddr，CustomerCity 直接依赖的是 CustomerID（非主键列），而不是直接依赖于主键，它是通过传递才依赖于主键，所以不符合 3NF。

什么是反模式

范式可以避免数据冗余，减少数据库的空间，减轻维护数据完整性的麻烦。

然而，通过数据库范式化设计，将导致数据库业务涉及的表变多，并且可能需要将涉及的业务表进行多表连接查询，这样将导致性能变差，且不利于分库分表。因此，出于性能优先的考量，可能在数据库的结构中需要使用反模式的设计，即空间换取时间，采取数据冗余的方式避免表之间的关联查询。至于数据一致性问题，因为难以满足数据强一致性，一般情况下，使存储数据尽可能达到用户一致，保证系统经过一段较短的时间的自我恢复和修正，数据最终达到一致。

需要谨慎使用反模式设计数据库。一般情况下，尽可能使用范式化的数据库设计，因为范式化的数据库设计能让产品更加灵活，并且能在数据库层保持数据完整性。

有的时候，提升性能最好的方法是在同一表中保存冗余数据，如果能容许少量的脏数据，创建一张完全独立的汇总表或缓存表是非常好的方法。举个例子，设计一张“下载次数表”来缓存下载次数信息，可使在海量数据的情况下，提高查询总数信息的速度。

另外一个比较典型的场景，出于扩展性考虑，可能会使用 BLOB 和 TEXT 类型的列存储 JSON 结构的数据，这样的好处在于可以在任何时候，将新的属性添加到这个字段中，而不需要更改表结构。但是，这个设计的缺点也比较明显，就是需要获取整个字段内容进行解码来获取指定的属性，并且无法进行索引、排序、聚合等操作。因此，如果需要考虑更加复杂的使用场景，更加建议使用 MongoDB 这样的文档型数据库。

数据库事务

事务是一个不可分割的数据库操作序列，也是数据库并发控制的基本单位，其执行的结果必须使数据库从一种一致性

状态变到另一种一致性状态。

(1). 事务的特征

原子性(Atomicity): 事务所包含的一系列数据库操作要么全部成功执行，要么全部回滚;

一致性(Consistency): 事务的执行结果必须使数据库从一个一致性状态到另一个一致性状态;

隔离性(Isolation): 并发执行的事务之间不能相互影响;

持久性(Durability): 事务一旦提交，对数据库中数据的改变是永久性的。

(2). 事务并发带来的问题

脏读: 一个事务读取了另一个事务未提交的数据;

不可重复读: 不可重复读的重点是修改，同样条件下两次读取结果不同，也就是说，被读取的数据可以被其它事务修改;

幻读: 幻读的重点在于新增或者删除，同样条件下两次读出来的记录数不一样。

(3). 隔离级别

隔离级别决定了一个session中的事务可能对另一个session中的事务的影响。

ANSI标准定义了4个隔离级别，MySQL的InnoDB都支持，分别是:

READ UNCOMMITTED（未提交读）: 最低级别的隔离，通常又称为dirty read，它允许一个事务读取另一个事务还没commit的数据，这样可能会提高性能，但是会导致脏读问题;

READ COMMITTED（提交读）: 在一个事务中只允许对其它事务已经commit的记录可见，该隔离级别不能避免不可重复读问题;

REPEATABLE READ（可重复读）: 在一个事务开始后，其他事务对数据库的修改在本事务中不可见，直到本事务commit或rollback。但是，其他事务的insert/delete操作对该事务是可见的，也就是说，该隔离级别并不能避免幻读问题。在一个事务中重复select的结果一样，除非本事务中update数据库。

SERIALIZABLE（可串行化）: 最高级别的隔离，只允许事务串行执行。

MySQL默认的隔离级别是REPEATABLE READ。

	脏读	不可重复读	幻读可能性	加锁读
未提交读	YES	YES	YES	NO
提交读	NO	YES	YES	NO
可重复读	NO	NO	YES	NO
可串行化	NO	NO	NO	YES

什么是存储过程？有哪些优缺点？

存储过程是事先经过编译并存储在数据库中的一段SQL语句的集合。进一步地说，存储过程是由一些T-SQL语句组成的

代码块，这些T-SQL语句代码像一个方法一样实现一些功能（对单表或多表的增删改查），然后再给这个代码块取一个名字，在用到这个功能的时候调用他就行了。存储过程具有以下特点：

- 存储过程只在创建时进行编译，以后每次执行存储过程都不需再重新编译，而一般 SQL 语句每执行一次就编译一次，所以使用存储过程可提高数据库执行效率；
- 当SQL语句有变动时，可以只修改数据库中的存储过程而不必修改代码；
- 减少网络传输，在客户端调用一个存储过程当然比执行一串SQL传输的数据量要小；
- 通过存储过程能够使没有权限的用户在控制之下间接地存取数据库，从而确保数据的安全。

简单说一说drop、delete与truncate的区别

SQL中的drop、delete、truncate都表示删除，但是三者有一些差别：

Delete用来删除表的全部或者一部分数据行，执行delete之后，用户需要提交(commmit)或者回滚(rollback)来执行删除或者撤销删除， delete命令会触发这个表上所有的delete触发器；

Truncate删除表中的所有数据，这个操作不能回滚，也不会触发这个表上的触发器，TRUNCATE比delete更快，占用的空间更小；

Drop命令从数据库中删除表，所有的数据行，索引和权限也会被删除，所有的DML触发器也不会被触发，这个命令也不能回滚。

因此，在不再需要一张表的时候，用drop；在想删除部分数据行时候，用delete；在保留表而删除所有数据的时候用truncate。

什么叫视图？游标是什么？

视图是一种虚拟的表，通常是有一个表或者多个表的行或列的子集，具有和物理表相同的功能，可以对视图进行增，删，改，查等操作。特别地，对视图的修改不影响基本表。相比多表查询，它使得我们获取数据更容易。

游标是对查询出来的结果集作为一个单元来有效的处理。游标可以定在该单元中的特定行，从结果集的当前行检索一行或多行。可以对结果集当前行做修改。一般不使用游标，但是需要逐条处理数据的时候，游标显得十分重要。

在操作mysql的时候，我们知道MySQL检索操作返回一组称为结果集的行。这组返回的行都是与 SQL语句相匹配的行（零行或多行）。使用简单的 SELECT语句，例如，没有办法得到第一行、下一行或前 10行，也不存在每次一行地处理所有行的简单方法（相对于成批地处理它们）。有时，需要在检索出来的行中前进或后退一行或多行。这就是使用游标的原因。游标（cursor）是一个存储在MySQL服务器上的数据库查询，它不是一条 SELECT语句，而是被该语句检索出来的结果集。在存储了游标之后，应用程序可以根据需要滚动或浏览其中的数据。游标主要用于交互式应用，其中用户需要滚动屏幕上的数据，并对数据进行浏览或做出更改。

什么是触发器？

触发器是与表相关的数据库对象，在满足定义条件时触发，并执行触发器中定义的语句集合。触发器的这种特性可以协助应用在数据库端确保数据库的完整性。

超键、候选键、主键、外键

- 超键：在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。
- 候选键：是最小超键，即没有冗余元素的超键。
- 主键：数据库表中对储存数据对象予以唯一和完整标识的数据列或属性的组合。一个数据列只能有一个主键，且主键的取值不能缺失，即不能为空值（Null）。
- 外键：在一个表中存在的另一个表的主键称此表的外键。

什么是事务？什么是锁？

- 事务：就是被绑定在一起作为一个逻辑工作单元的 SQL 语句分组，如果任何一个语句操作失败那么整个操作就被失败，以后操作就会回滚到操作前状态，或者是上有个节点。为了确保要么执行，要么不执行，就可以使用事务。要将有组语句作为事务考虑，就需要通过 ACID 测试，即原子性，一致性，隔离性和持久性。
- 锁：在所以的 DBMS 中，锁是实现事务的关键，锁可以保证事务的完整性和并发性。与现实生活中锁一样，它可以使某些数据的拥有者，在某段时间内不能使用某些数据或数据结构。当然锁还分级别的。

数据库锁机制

数据库锁定机制简单来说就是数据库为了保证数据的一致性而使各种共享资源在被并发访问，访问变得有序所设计的一种规则。MySQL各存储引擎使用了三种类型（级别）的锁定机制：行级锁定，页级锁定和表级锁定。

- 表级锁定（**table-level**）：表级别的锁定是MySQL各存储引擎中最大颗粒度的锁定机制。该锁定机制最大的特点是实现逻辑非常简单，带来的系统负面影响最小。所以获取锁和释放锁的速度很快。由于表级锁一次会将整个表锁定，所以可以很好的避免困扰我们的死锁问题。当然，锁定颗粒度大所带来最大的负面影响就是出现锁定资源争用的概率也会最高，致使并大度大打折扣。表级锁分为读锁和写锁。

- 页级锁定（**page-level**）：页级锁定的特点是锁定颗粒度介于行级锁定与表级锁之间，所以获取锁定所需要的资源开销，以及所能提供的并发处理能力也同样是介于上面二者之间。另外，页级锁定和行级锁定一样，会发生死锁。

- 行级锁定（**row-level**）：行级锁定最大的特点就是锁定对象的颗粒度很小，也是目前各大数据库管理软件所实现的锁定颗粒度最小的。由于锁定颗粒度很小，所以发生锁定资源争用的概率也最小，能够给予应用程序尽可能大的并发处理能力而提高一些需要高并发应用系统的整体性能。虽然能够在并发处理能力上面有较大的优势，但是行级锁定也因此带来了不少弊端。由于锁定资源的颗粒度很小，所以每次获取锁和释放锁需要做的事情也更多，带来的消耗自然也就更大了。此外，行级锁定也最容易发生死锁。InnoDB的行级锁同样分为两种，共享锁和排他锁，同样InnoDB也引入了意向锁（表级锁）的概念，所以也就有了意向共享锁和意向排他锁，所以InnoDB实际上有四种锁，即共享锁（S）、排他锁（X）、意向共享锁（IS）、意向排他锁（IX）；

在MySQL数据库中，使用表级锁定的主要是MyISAM，Memory，CSV等一些非事务性存储引擎，而使用行级锁定的主要是Innodb存储引擎和NDBCluster存储引擎，页级锁定主要是BerkeleyDB存储引擎的锁定方式。

而意向锁的作用就是当一个事务在需要获取资源锁定的时候，如果遇到自己需要的资源已经被排他锁占用的时候，该事务可以需要锁定行的表上面添加一个合适的意向锁。如果自己需要一个共享锁，那么就在表上面添加一个意向共享锁。而如果自己需要的是某行（或者某些行）上面添加一个排他锁的话，则先在表上面添加一个意向排他锁。意向共享锁可以同时并存多个，但是意向排他锁同时只能有一个存在。

	共享锁（S）	排他锁（X）	意向共享锁（IS）	意向排他锁（IX）
共享锁（S）	兼容	冲突	兼容	冲突

排他锁（X）	冲突	冲突	冲突	冲突
意向共享锁（IS）	兼容	冲突	兼容	兼容
意向排他锁（IX）	冲突	冲突	兼容	兼容

参考地址：`http://www.cnblogs.com/ggjucheng/archive/2012/11/14/2770445.html`

DDL、DML、DCL分别指什么

左连接、右连接、内连接、外连接、交叉连接、笛卡儿积

缓存基本理论

缓存基础

缓存雪崩

缓存雪崩是由于原有缓存失效(过期)，新缓存未到期间。所有请求都去查询数据库，而对数据库CPU和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。

解决方法：

1. 一般并发量不是特别多的时候，使用最多的解决方案是加锁排队。
2. 给每一个缓存数据增加相应的缓存标记，记录缓存的是否失效，如果缓存标记失效，则更新数据缓存。
    - 缓存标记：记录缓存数据是否过期，如果过期会触发通知另外的线程在后台去更新实际key的缓存。
    - 缓存数据：它的过期时间比缓存标记的时间延长1倍，例：标记缓存时间30分钟，数据缓存设置为60分钟。 这样，当缓存标记key过期后，实际缓存还能把旧数据返回给调用端，直到另外的线程在后台更新完成后，才会返回新缓存。

加锁排队方案伪代码：

```
//伪代码
public object GetProductListNew() {
    int cacheTime = 30;
    String cacheKey = "product_list";
    String lockKey = cacheKey;

    String cacheValue = CacheHelper.get(cacheKey);
    if (cacheValue != null) {
        return cacheValue;
    } else {
        synchronized(lockKey) {
            cacheValue = CacheHelper.get(cacheKey);
            if (cacheValue != null) {
                return cacheValue;
            } else {
                //这里一般是sql查询数据
                cacheValue = GetProductListFromDB();
                CacheHelper.Add(cacheKey, cacheValue, cacheTime);
            }
        }
        return cacheValue;
    }
}
```

缓存标记方案伪代码：

```
//伪代码
public object GetProductListNew() {
    int cacheTime = 30;
    String cacheKey = "product_list";
    //缓存标记
    String cacheSign = cacheKey + "_sign";

    String sign = CacheHelper.Get(cacheSign);
    //获取缓存值
    String cacheValue = CacheHelper.Get(cacheKey);
    if (sign != null) {
        return cacheValue; //未过期，直接返回
    } else {
        CacheHelper.Add(cacheSign, "1", cacheTime);
        ThreadPool.QueueUserWorkItem((arg) -> {
            //这里一般是 sql查询数据
            cacheValue = GetProductListFromDB();
            //日期设缓存时间的2倍，用于脏读
            CacheHelper.Add(cacheKey, cacheValue, cacheTime * 2);
        });
        return cacheValue;
    }
}
```

缓存穿透

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

解决方案：

1. 布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。
2. 如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。通过这个直接设置的默认值存放到缓存，这样第二次到缓冲中获取就有值了，而不会继续访问数据库，这种办法最简单粗暴！

方案二伪代码：

```
//伪代码
public object GetProductListNew() {
    int cacheTime = 30;
    String cacheKey = "product_list";

    String cacheValue = CacheHelper.Get(cacheKey);
    if (cacheValue != null) {
        return cacheValue;
    }

```

```
    cacheValue = CacheHelper.Get(cacheKey);
    if (cacheValue != null) {
        return cacheValue;
    } else {
        //数据库查询不到，为空
        cacheValue = GetProductListFromDB();
        if (cacheValue == null) {
            //如果发现为空，设置个默认值，也缓存起来
            cacheValue = string.Empty;
        }
        CacheHelper.Add(cacheKey, cacheValue, cacheTime);
        return cacheValue;
    }
}
```

缓存预热

缓存预热这个应该是一个比较常见的概念，相信很多小伙伴都应该可以很容易的理解，缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

解决思路：

1. 直接写个缓存刷新页面，上线时手工操作下；
2. 数据量不大，可以在项目启动的时候自动进行加载；
3. 定时刷新缓存；

缓存更新

除了缓存服务器自带的缓存失效策略之外，我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

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

缓存降级

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

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的。

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

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

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

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

（4）严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

参考文章：

- 缓存雪崩、缓存穿透、缓存预热、缓存更新、缓存降级等问题

数据库索引

索引

索引的优点

- 大大加快数据的检索速度，这也是创建索引的最主要的原因；
- 加速表和表之间的连接；
- 在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间；
- 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；

什么情况下设置了索引但无法使用？

- 以“%(表示任意0个或多个字符)”开头的LIKE语句，模糊匹配；
- OR语句前后没有同时使用索引；
- 数据类型出现隐式转化（如varchar不加单引号的话可能会自动转换为int型）；
- 对于多列索引，必须满足 最左匹配原则 (eg: 多列索引col1、col2和col3，则 索引生效的情形包括 col1或col1，col2或col1，col2，col3)。

什么样的字段适合创建索引？

- 经常作查询选择的字段
- 经常作表连接的字段
- 经常出现在order by, group by, distinct 后面的字段

创建索引时需要注意什么？

非空字段：应该指定列为NOT NULL，除非你想存储NULL。在mysql中，含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。你应该用0、一个特殊的值或者一个空串代替空值；

取值离散大的字段：（变量各个取值之间的差异程度）的列放到联合索引的前面，可以通过count()函数查看字段的差异值，返回值越大说明字段的唯一值越多字段的离散程度高；

索引字段越小越好：数据库的数据存储以页为单位一页存储的数据越多一次IO操作获取的数据越大效率越高。

索引的缺点

时间方面：创建索引和维护索引要耗费时间，具体地，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度；

空间方面：索引需要占物理空间。

索引的分类

普通索引和唯一性索引：索引列的值的唯一性

单个索引和复合索引：索引列所包含的列数

聚簇索引与非聚簇索引：聚簇索引按照数据的物理存储进行划分的。对于一堆记录来说，使用聚集索引就是对这堆记录进行堆划分，即主要描述的是物理上的存储。正是因为这种划分方法，导致聚簇索引必须是唯一的。聚集索引可以帮助把很大的范围，迅速减小范围。但是查找该记录，就要从这个小范围中Scan了；而非聚集索引是把一个很大的范围，转换成一个小的地图，然后你需要在这个小地图中找你要寻找的信息的位置，最后通过这个位置，再去找你所需要的记录。

主键、自增主键、主键索引与唯一索引概念区别

主键：指字段 唯一、不为空值 的列；

主键索引：指的就是主键，主键是索引的一种，是唯一索引的特殊类型。创建主键的时候，数据库默认会为主键创建一个唯一索引；

自增主键：字段类型为数字、自增、并且是主键；

唯一索引：索引列的值必须唯一，但允许有空值。主键是唯一索引，这样说没错；但反过来说，唯一索引也是主键就错误了，因为唯一索引允许空值，主键不允许有空值，所以不能说唯一索引也是主键。

主键就是聚集索引吗？主键和索引有什么区别？

主键是一种特殊的唯一性索引，其可以是聚集索引，也可以是非聚集索引。

在SQLServer中，主键的创建必须依赖于索引，默认创建的是聚集索引，但也可以显式指定为非聚集索引。

InnoDB作为MySQL存储引擎时，默认按照主键进行聚集，如果没有定义主键，InnoDB会试着使用唯一的非空索引来代替。如果没有这种索引，InnoDB就会定义隐藏的主键然后在上面进行聚集。所以，对于聚集索引来说，你创建主键的时候，自动就创建了主键的聚集索引。

索引的底层实现原理和优化

索引是对数据库表中一个或多个列的值进行排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用B_TREE及其变种。索引加速了数据访问，因为存储引擎不会再去扫描整张表得到需要的数据；相反，它从根节点开始，根节点保存了子节点的指针，存储引擎会根据指针快速寻找数据。

上图显示了一种索引方式。左边是数据库中的数据表，有col1和col2两个字段，一共有15条记录；右边是以col2列为索引列的B_TREE索引，每个节点包含索引的键值和对应数据表地址的指针，这样就可以都过B_TREE在O(logn) 的时间复杂度内获取相应的数据，这样明显地加快了检索的速度。

在数据结构中，我们最为常见的搜索结构就是二叉搜索树和AVL树(高度平衡的二叉搜索树，为了提高二叉搜索树的效

率，减少树的平均搜索长度)了。然而，无论二叉搜索树还是AVL树，当数据量比较大时，都会由于树的深度过大而造成I/O读写过于频繁，进而导致查询效率低下，因此对于索引而言，多叉树结构成为不二选择。特别地，B-Tree的各种操作能使B树保持较低的高度，从而保证高效的查找效率。

B-Tree(平衡多路查找树)

B_TREE是一种平衡多路查找树，是一种动态查找效率很高的树形结构。B_TREE中所有结点的孩子结点的最大值称为B_TREE的阶，B_TREE的阶通常用m表示，简称为m叉树。一般来说，应该是m>=3。一颗m阶的B_TREE或是一颗空树，或者是满足下列条件的m叉树:

1)树中每个结点最多有m个孩子结点;

2)若根结点不是叶子节点，则根结点至少有2个孩子结点;

3)除根结点外，其它结点至少有(m/2的上界)个孩子结点;

结点的结构如下图所示，其中，n为结点中关键字个数，(m/2的上界)-1 <= n <= m-1; di(1<=i<=n)为该结点的n个关键字值的第i个，且di< d(i+1); ci(0<=i<=n)为该结点孩子结点的指针，且ci所指向的节点的关键字均大于或等于di且小于d(i+1);

所有的叶结点都在同一层上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。

下图是一棵4阶B_TREE，4叉树结点的孩子结点的个数范围[2,4]。其中，有2个结点有4个孩子结点，有1个结点有3个孩子结点，有5个结点有2个孩子结点。

B_TREE的查找类似二叉排序树的查找，所不同的是B-树每个结点上是多关键码的有序表，在到达某个结点时，先在有序表中查找，若找到，则查找成功;否则，到按照对应的指针信息指向的子树中去查找，当到达叶子结点时，则说明树中没有对应的关键码。由于B_TREE的高检索效率，B-树主要应用在文件系统和数据库中，对于存储在硬盘上的大型数据库文件，可以极大程度减少访问硬盘次数，大幅度提高数据检索效率。

B+Tree ： InnoDB存储引擎的索引实现

B+Tree是应文件系统所需而产生的一种B_TREE树的变形树。一棵m阶的B+树和m阶的B_TREE的差异在于以下三点:

- n棵子树的结点中含有n个关键码;
- 所有的叶子结点中包含了全部关键码的信息，及指向含有这些关键码记录的指针，且叶子结点本身依关键码的大小自小而大的顺序链接;
- 非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键码。

下图为一棵3阶的B+树。通常在B+树上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点。因此可以对B+树进行两种查找运算：一种是从最小关键字起顺序查找，另一种是从根节点开始，进行随机查找。

在B+树上进行随机查找、插入和删除的过程基本上与B-树类似。只是在查找时，若非终端结点上的关键码等于给定值，并不终止，而是继续向下直到叶子结点。因此，对于B+树，不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径。

为什么说B+树比B树更适合实际应用中操作系统的文件索引和数据库索引？

B+tree的磁盘读写代价更低：B+tree的内部结点并没有指向关键字具体信息的指针(红色部分)，因此其内部结点相对B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多，相对来说IO读写次数也就降低了；

B+tree的查询效率更加稳定：由于内部结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引，所以，任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当；

数据库索引采用B+树而不是B树的主要原因：B+树只要遍历叶子节点就可以实现整棵树的遍历，而且在数据库中基于范围的查询是非常频繁的，而B树只能中序遍历所有节点，效率太低。

文件索引和数据库索引为什么使用B+树？

文件与数据库都是需要较大的存储，也就是说，它们都不可能全部存储在内存中，故需要存储到磁盘上。而所谓索引，则为了数据的快速定位与查找，那么索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数，因此B+树相比B树更为合适。数据库系统巧妙利用了局部性原理与磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入，而红黑树这种结构，高度明显要深的多，并且由于逻辑上很近的节点(父子)物理上可能很远，无法利用局部性。最重要的是，B+树还有一个最大的好处：方便扫库。B树必须用中序遍历的方法按序扫库，而B+树直接从叶子结点挨个扫一遍就完了，B+树支持range-query非常方便，而B树不支持，这是数据库选用B+树的最主要原因。

分库分表

分库分表

说说分库与分表设计

面对海量数据，例如，上千万甚至上亿的数据，查询一次所花费的时间会变长，甚至会造成数据库的单点压力。因此，分库与分表的目的在于，减小数据库的单库单表负担，提高查询性能，缩短查询时间。

分表概述

随着用户数的不断增加，以及数据量的不断增加，会使得单表压力越来越大，面对上千万甚至上亿的数据，查询一次所花费的时间会变长，如果有联合查询的情况下，甚至可能会成为很大的瓶颈。此外，MySQL 存在表锁和行锁，因此更新表数据可能会引起表锁或者行锁，这样也会导致其他操作等待，甚至死锁问题。

通过分表，可以减少数据库的单表负担，将压力分散到不同的表上，同时因为不同的表上的数据量少了，起到提高查询性能，缩短查询时间的作用，此外，可以很大的缓解表锁的问题。

分表策略可以归纳为垂直拆分和水平拆分。

垂直拆分，把表的字段进行拆分，即一张字段比较多的表拆分为多张表，这样使得行数据变小。一方面，可以减少客户端程序和数据库之间的网络传输的字节数，因为生产环境共享同一个网络带宽，随着并发查询的增多，有可能造成带宽瓶颈从而造成阻塞。另一方面，一个数据块能存放更多的数据，在查询时就会减少 I/O 次数。举个例子，假设用户表中有一个字段是家庭地址，这个字段是可选字段，在数据库操作的时候除了个人信息外，并不需要经常读取或是更改这个字段的值。在这种情况下，更建议把它拆分到另外一个表，从而提高性能。

如何设计好垂直拆分，我的建议：

- 将不常用的字段单独拆分到另外一张扩展表，例如前面讲解到的用户家庭地址，这个字段是可选字段，在数据库操作的时候除了个人信息外，并不需要经常读取或是更改这个字段的值。
- 将大文本的字段单独拆分到另外一张扩展表，例如 BLOB 和 TEXT 字符串类型的字段，以及 TINYBLOB、MEDIUMBLOB、 LONGBLOB、 TINYTEXT、 MEDIUMTEXT、 LONGTEXT字符串类型。这样可以减少客户端程序和- 数据库之间的网络传输的字节数。
- 将不经常修改的字段放在同一张表中，将经常改变的字段放在另一张表中。举个例子，假设用户表的设计中，还存在“最后登录时间”字段，每次用户登录时会被更新。这张用户表会存在频繁的更新操作，此外，每次更新时会导致该表的查询缓存被清空。所以，可以把这个字段放到另一个表中，这样查询缓存会增加很多性能。
- 对于需要经常关联查询的字段，建议放在同一张表中。不然在联合查询的情况下，会带来数据库额外压力。
- 水平拆分，把表的行进行拆分。因为表的行数超过几百万行时，就会变慢，这时可以把一张的表的数据拆成多张表来存放。水平拆分，有许多策略，例如，取模分表，时间维度分表，以及自定义 Hash 分表，例如用户 ID 维度分表等。在不同策略分表情况下，根据各自的策略写入与读取。

实际上，垂直拆分后的表依然存在单表数据量过大的问题，需要进行水平拆分。因此，实际情况中，水平拆分往往会和垂直拆分结合使用。假设，随着用户数的不断增加，用户表单表存在上千万的数据，这时可以把一张用户表的数据拆成多张用户表来存放。

常见的水平分表策略归纳起来，可以总结为随机分表和连续分表两种情况。例如，取模分表就属于随机分表，而时间维度分表则属于连续分表。

连续分表可以快速定位到表进行高效查询，大多数情况下，可以有效避免跨表查询。如果想扩展，只需要添加额外的分表就可以了，无需对其他分表的数据进行数据迁移。但是，连续分表有可能存在数据热点的问题，有些表可能会被频繁地查询从而造成较大压力，热数据的表就成为了整个库的瓶颈，而有些表可能存的是历史数据，很少需要被查询到。

随机分表是遵循规则策略进行写入与读取，而不是真正意义上的随机。通常，采用取模分表或者自定义 Hash 分表的方式进行水平拆分。随机分表的数据相对比较均匀，不容易出现热点和并发访问的瓶颈。但是，分表扩展需要迁移旧的数据。此外，随机分表比较容易面临跨表查询的复杂问题。

对于日志场景，可以考虑根据时间维度分表，例如年份维度分表或者月份维度分表，在日志记录表的名字中包含年份和月份的信息，例如 log_2017_01，这样可以在已经没有新增操作的历史表上做频繁地查询操作，而不会影响时间维度分表上新增操作。

对于海量用户场景，可以考虑取模分表，数据相对比较均匀，不容易出现热点和并发访问的瓶颈。

对于租户场景，可以考虑租户维度分表，不同的租户数据独立，而不应该在每张表中添加租户 ID，这是一个不错的选择。

分库概述

库内分表，仅仅是解决了单表数据过大的问题，但并没有把单表的数据分散到不同的物理机上，因此并不能减轻 MySQL 服务器的压力，仍然存在同一个物理机上的资源竞争和瓶颈，包括 CPU、内存、磁盘 IO、网络带宽等。

分库策略也可以归纳为垂直拆分和水平拆分。

垂直拆分，按照业务和功能划分，把数据分别放到不同的数据库中。举个例子，可以划分资讯库、百科库等。

水平拆分，把一张表的数据划分到不同的数据库，两个数据库的表结构一样。实际上，水平分库与水平分表类似，水平拆分有许多策略，例如，取模分库，自定义 Hash 分库等，在不同策略分库情况下，根据各自的策略写入与读取。举个例子，随着业务的增长，资讯库的单表数据过大，此时采取水平拆分策略，根据取模分库。

以上文字来源：服务端指南 数据存储篇 | MySQL（08） 分库与分表设计

分库与分表带来的分布式困境与应对之策

- 数据迁移与扩容问题
- 表关联问题
- 分页与排序问题
- 分布式事务问题
- 分布式全局唯一ID

选择合适的分布式主键方案

如何设计可以动态扩容缩容的分库分表方案？

用过哪些分库分表中间件，有啥优点和缺点？讲一下你了解的分库分表中间件的底层实现原理？

MySQL

MySQL

实践中如何优化MySQL

实践中，MySQL的优化主要涉及SQL语句及索引的优化、数据表结构的优化、系统配置的优化和硬件的优化四个方面，如下图所示：

SQL语句及索引的优化

SQL语句的优化

SQL语句的优化主要包括三个问题，即如何发现有问题的SQL、如何分析SQL的执行计划以及如何优化SQL，下面将逐一解释。

1. 怎么发现有问题的SQL？（通过MySQL慢查询日志对有效率问题的SQL进行监控）

MySQL的慢查询日志是MySQL提供的一种日志记录，它用来记录在MySQL中响应时间超过阀值的语句，具体指运行时间超过long_query_time值的SQL，则会被记录到慢查询日志中。

long_query_time的默认值为10，意思是运行10s以上的语句。慢查询日志的相关参数如下所示：

通过MySQL的慢查询日志，我们可以查询出执行的次数多占用的时间长的SQL、可以通过pt_query_disgest(一种mysql慢日志分析工具)分析Rows examine(MySQL执行器需要检查的行数)项去找出IO大的SQL以及发现未命中索引的SQL，对于这些SQL，都是我们优化的对象。

通过explain查询和分析SQL的执行计划

使用 EXPLAIN 关键字可以知道MySQL是如何处理你的SQL语句的，以便分析查询语句或是表结构的性能瓶颈。通过explain命令可以得到表的读取顺序、数据读取操作的操作类型、哪些索引可以使用、哪些索引被实际使用、表之间的引用以及每张表有多少行被优化器查询等问题。当扩展列extra出现Using filesort和Using temporay，则往往表示SQL需要优化了。

优化SQL语句

- 优化insert语句：一次插入多值；
- 应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描；
- 应尽量避免在 where 子句中对字段进行null值判断，否则将导致引擎放弃使用索引而进行全表扫描；
- 优化嵌套查询：子查询可以被更有效率的连接(Join)替代；
- 很多时候用 exists 代替 in 是一个好的选择。

索引优化

建议在经常作查询选择的字段、经常作表连接的字段以及经常出现在order by、group by、distinct 后面的字段中建立索引。但必须注意以下几种可能会引起索引失效的情形:

- 以“%(表示任意0个或多个字符)”开头的LIKE语句，模糊匹配;
- OR语句前后没有同时使用索引;
- 数据类型出现隐式转化（如varchar不加单引号的话可能会自动转换为int型）;
- 对于多列索引，必须满足最左匹配原则(eg,多列索引col1、col2和col3，则 索引生效的情形包括col1或col1，col2或col1，col2，col3)。

数据库表结构的优化

数据库表结构的优化包括选择合适数据类型、表的范式的优化、表的垂直拆分和表的水平拆分等手段。

选择合适数据类型

- 使用较小的数据类型解决问题;
- 使用简单的数据类型(mysql处理int要比varchar容易);
- 尽可能的使用not null 定义字段;
- 尽量避免使用text类型，非用不可时最好考虑分表;

表的范式的优化

一般情况下，表的设计应该遵循三大范式。

表的垂直拆分

- 把含有多个列的表拆分成多个表，解决表宽度问题，具体包括以下几种拆分手段:
- 把不常用的字段单独放在同一个表中;
- 把大字段独立放入一个表中;
- 把经常使用的字段放在一起;
- 这样做的好处是非常明显的，具体包括:拆分后业务清晰，拆分规则明确、系统之间整合或扩展容易、数据维护简单。

表的水平拆分

表的水平拆分用于解决数据表中数据过大的问题，水平拆分每一个表的结构都是完全一致的。一般地，将数据平分到N张表中的常用方法包括以下两种:

- 对ID进行hash运算，如果要拆分成5个表，mod(id,5)取出0~4个值;
- 针对不同的hashID将数据存入不同的表中;

- 表的水平拆分会带来一些问题和挑战，包括跨分区表的数据查询、统计及后台报表的操作等问题，但也带来了一些切实的好处:

    - 表分割后可以降低在查询时需要读的数据和索引的页数，同时也降低了索引的层数，提高查询速度;
    - 表中的数据本来就有独立性，例如表中分别记录各个地区的数据或不同时期的数据，特别是有些数据常

用，而另外一些数据不常用。

- 需要把数据存放到多个数据库中，提高系统的总体可用性(分库，鸡蛋不能放在同一个篮子里)。

系统配置的优化

操作系统配置的优化：增加TCP支持的队列数

mysql配置文件优化：Innodb缓存池设置(innodb_buffer_pool_size，推荐总内存的75%)和缓存池的个数（innodb_buffer_pool_instances）

硬件的优化

CPU：核心数多并且主频高的
内存：增大内存
磁盘配置和选择：磁盘性能

MySQL中的悲观锁与乐观锁的实现

悲观锁与乐观锁是两种常见的资源并发锁设计思路，也是并发编程中一个非常基础的概念。

悲观锁

悲观锁的特点是先获取锁，再进行业务操作，即“悲观”的认为所有的操作均会导致并发安全问题，因此要先确保获取锁成功再进行业务操作。通常来讲，在数据库上的悲观锁需要数据库本身提供支持，即通过常用的select … for update操作来实现悲观锁。当数据库执行select … for update时会获取被select中的数据行的行锁，因此其他并发执行的select … for update如果试图选中同一行则会发生排斥（需要等待行锁被释放），因此达到锁的效果。select for update获取的行锁会在当前事务结束时自动释放，因此必须在事务中使用。

这里需要特别注意的是，不同的数据库对select… for update的实现和支持都是有所区别的，例如oracle支持select for update no wait，表示如果拿不到锁立刻报错，而不是等待，mysql就没有no wait这个选项。另外，mysql还有个问题是：select… for update语句执行中所有扫描过的行都会被锁上，这一点很容易造成问题。因此，如果在mysql中用悲观锁务必要确定使用了索引，而不是全表扫描。

乐观锁

乐观锁的特点先进行业务操作，只在最后实际更新数据时进行检查数据是否被更新过，若未被更新过，则更新成功；否则，失败重试。乐观锁在数据库上的实现完全是逻辑的，不需要数据库提供特殊的支持。一般的做法是在需要锁的数据上增加一个版本号或者时间戳，然后按照如下方式实现：

```
SELECT data AS old_data, version AS old_version FROM …;
//根据获取的数据进行业务操作，得到new_data和new_version
UPDATE SET data = new_data, version = new_version WHERE version = old_version
if (updated row > 0) {
// 乐观锁获取成功，操作完成
} else {
// 乐观锁获取失败，回滚并重试
```

```
8. }
```

乐观锁是否在事务中其实都是无所谓的，其底层机制是这样：在数据库内部update同一行的时候是不允许并发的，即数据库每次执行一条update语句时会获取被update行的写锁，直到这一行被成功更新后才释放。因此在业务操作进行前获取需要锁的数据的当前版本号，然后实际更新数据时再次对比版本号确认与之前获取的相同，并更新版本号，即可确认这其间没有发生并发的修改。如果更新失败，即可认为老版本的数据已经被并发修改掉而不存在了，此时认为获取锁失败，需要回滚整个业务操作并可根据需要重试整个过程。

悲观锁与乐观锁的应用场景

一般情况下，读多写少更适合用乐观锁，读少写多更适合用悲观锁。乐观锁在不发生取锁失败的情况下开销比悲观锁小，但是一旦发生失败回滚开销则比较大，因此适合用在取锁失败概率比较小的场景，可以提升系统并发性能。

MySQL存储引擎中的MyISAM和InnoDB区别详解

在MySQL 5.5之前，MyISAM是mysql的默认数据库引擎，其由早期的ISAM（Indexed Sequential Access Method：有索引的顺序访问方法）所改良。虽然MyISAM性能极佳，但却有一个显著的缺点： 不支持事务处理。不过，MySQL也导入了另一种数据库引擎InnoDB，以强化参考完整性与并发违规处理机制，后来就逐渐取代MyISAM。

InnoDB是MySQL的数据库引擎之一，其由Innobase oy公司所开发，2006年五月由甲骨文公司并购。与传统的ISAM、MyISAM相比，InnoDB的最大特色就是支持ACID兼容的事务功能，类似于PostgreSQL。目前InnoDB采用双轨制授权，一是GPL授权，另一是专有软件授权。具体地，MyISAM与InnoDB作为MySQL的两大存储引擎的差异主要包括：

存储结构：每个MyISAM在磁盘上存储成三个文件：第一个文件的名字以表的名字开始，扩展名指出文件类型。.frm文件存储表定义，数据文件的扩展名为.MYD (MYData)，索引文件的扩展名是.MYI (MYIndex)。InnoDB所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB表的大小只受限于操作系统文件的大小，一般为2GB。

存储空间：MyISAM可被压缩，占据的存储空间较小，支持静态表、动态表、压缩表三种不同的存储格式。InnoDB需要更多的内存和存储，它会在主内存中建立其专用的缓冲池用于高速缓冲数据和索引。

可移植性、备份及恢复：MyISAM的数据是以文件的形式存储，所以在跨平台的数据转移中会很方便，同时在备份和恢复时也可单独针对某个表进行操作。InnoDB免费的方案可以是拷贝数据文件、备份 binlog，或者用 mysqldump，在数据量达到几十G的时候就相对痛苦了。

事务支持：MyISAM强调的是性能，每次查询具有原子性，其执行数度比InnoDB类型更快，但是不提供事务支持。InnoDB提供事务、外键等高级数据库功能，具有事务提交、回滚和崩溃修复能力。

AUTO_INCREMENT：在MyISAM中，可以和其他字段一起建立联合索引。引擎的自动增长列必须是索引，如果是组合索引，自动增长可以不是第一列，它可以根据前面几列进行排序后递增。InnoDB中必须包含只有该字段的索引，并且引擎的自动增长列必须是索引，如果是组合索引也必须是组合索引的第一列。

表锁差异：MyISAM只支持表级锁，用户在操作MyISAM表时，select、update、delete和insert语句都会给表自动加锁，如果加锁以后的表满足insert并发的情况下，可以在表的尾部插入新的数据。InnoDB支持事务和行级锁。行锁大幅度提高了多用户并发操作的新能，但是InnoDB的行锁，只是在WHERE的主键是有效的，非主键的WHERE都会锁全表的。

全文索引：MyISAM支持 FULLTEXT类型的全文索引；InnoDB不支持FULLTEXT类型的全文索引，但是innodb可以使用sphinx插件支持全文索引，并且效果更好。

表主键：MyISAM允许没有任何索引和主键的表存在，索引都是保存行的地址。对于InnoDB，如果没有设定主键或者非空唯一索引，就会自动生成一个6字节的主键(用户不可见)，数据是主索引的一部分，附加索引保存的是主索引的值。

表的具体行数：MyISAM保存表的总行数，select count() from table;会直接取出出该值；而InnoDB没有保存表的总行数，如果使用select count() from table；就会遍历整个表，消耗相当大，但是在加了wehre条件后，myisam和innodb处理的方式都一样。

CURD操作：在MyISAM中，如果执行大量的SELECT，MyISAM是更好的选择。对于InnoDB，如果你的数据执行大量的INSERT或UPDATE，出于性能方面的考虑，应该使用InnoDB表。DELETE从性能上InnoDB更优，但DELETE FROM table时，InnoDB不会重新建立表，而是一行一行的删除，在innodb上如果要清空保存有大量数据的表，最好使用truncate table这个命令。

外键：MyISAM不支持外键，而InnoDB支持外键。

通过上述的分析，基本上可以考虑使用InnoDB来替代MyISAM引擎了，原因是InnoDB自身很多良好的特点，比如事务支持、存储过程、视图、行级锁、外键等等。尤其在并发很多的情况下，相信InnoDB的表现肯定要比MyISAM强很多。另外，必须需要注意的是，任何一种表都不是万能的，合适的才是最好的，才能最大的发挥MySQL的性能优势。如果是不复杂的、非关键的Web应用，还是可以继续考虑MyISAM的，这个具体情况具体考虑。

MyISAM：不支持事务，不支持外键，表锁；插入数据时锁定整个表，查行数时无需整表扫描。主索引数据文件和索引文件分离；与主索引无区别；

InnoDB：支持事务，外键，行锁，查表总行数时，全表扫描；主索引的数据文件本身就是索引文件；辅助索引记录主键的值；

MySQL锁类型

根据锁的类型分，可以分为共享锁，排他锁，意向共享锁和意向排他锁。

根据锁的粒度分，又可以分为行锁，表锁。

对于mysql而言，事务机制更多是靠底层的存储引擎来实现，因此，mysql层面只有表锁，而支持事务的innodb存储引擎则实现了行锁(记录锁（在行相应的索引记录上的锁），gap锁（是在索引记录间歇上的锁），next-key锁（是记录锁和在此索引记录之前的gap上的锁的结合）。Mysql的记录锁实质是索引记录的锁，因为innodb是索引组织表；gap锁是索引记录间隙的锁，这种锁只在RR隔离级别下有效；next-key锁是记录锁加上记录之前gap锁的组合。mysql通过gap锁和next-key锁实现RR隔离级别。

说明：对于更新操作(读不上锁)，只有走索引才可能上行锁；否则会对聚簇索引的每一行上写锁，实际等同于对表上写锁。

若多个物理记录对应同一个索引，若同时访问，也会出现锁冲突；

当表有多个索引时，不同事务可以用不同的索引锁住不同的行，另外innodb会同时用行锁对数据记录(聚簇索引)加锁。

MVCC(多版本并发控制)并发控制机制下，任何操作都不会阻塞读操作，读操作也不会阻塞任何操作，只因为读不上锁。

共享锁：由读表操作加上的锁，加锁后其他用户只能获取该表或行的共享锁，不能获取排它锁，也就是说只能读不能写

排它锁：由写表操作加上的锁，加锁后其他用户不能获取该表或行的任何锁，典型是mysql事务中的更新操作。

意向共享锁（IS）：事务打算给数据行加行共享锁，事务在给一个数据行加共享锁前必须先取得该表的IS锁。

意向排他锁（IX）：事务打算给数据行加行排他锁，事务在给一个数据行加排他锁前必须先取得该表的IX锁。

数据库死锁概念

多数情况下，可以认为如果一个资源被锁定，它总会在以后某个时间被释放。而死锁发生在当多个进程访问同一数据库时，其中每个进程拥有的锁都是其他进程所需的，由此造成每个进程都无法继续下去。简单的说，进程A等待进程B释放他的资源，B又等待A释放他的资源，这样就互相等待就形成死锁。

虽然进程在运行过程中，可能发生死锁，但死锁的发生也必须具备一定的条件，死锁的发生必须具备以下四个必要条件：

1）互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。

2）请求和保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。

3）不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。

4）环路等待条件：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{P0，P1，P2，•••，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

下列方法有助于最大限度地降低死锁：

- 按同一顺序访问对象。
- 避免事务中的用户交互。
- 保持事务简短并在一个批处理中。
- 使用低隔离级别。
- 使用绑定连接。

千万级MySQL数据库建立索引的事项及提高性能的手段

1. 对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。
2. 应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：select id from t where num is null可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：select id from t where num=0
3. 应尽量避免在 where 子句中使用!=或<>操作符，否则引擎将放弃使用索引而进行全表扫描。

4. 应尽量避免在 where 子句中使用or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：select id from t where num=10 or num=20可以这样查询：select id from t where num=10 union all select id from t where num=20
5. in 和 not in 也要慎用，否则会导致全表扫描，如：select id from t where num in(1,2,3) 对于连续的数值，能用 between 就不要用 in 了：select id from t where num between 1 and 3
6. 避免使用通配符。下面的查询也将导致全表扫描：select id from t where name like ‘李%’若要提高效率，可以考虑全文检索。
7. 如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：select id from t where num=@num可以改为强制查询使用索引：select id from t with(index(索引名)) where num=@num
8. 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：select id from t where num/2=100应改为:select id from t where num=100*2
9. 应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：select id from t where substring(name,1,3)=’abc’ ，name以abc开头的id应改为:select id from t where name like ‘abc%’
10. 不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。
11. 在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。
12. 不要写一些没有意义的查询，如需要生成一个空表结构：select col1,col2 into #t from t where 1=0 这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：create table #t(…)
13. 很多时候用 exists 代替 in 是一个好的选择：select num from a where num in(select num from b)用下面的语句替换：select num from a where exists(select 1 from b where num=a.num)
14. 并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。
15. 索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。
16. 应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。
17. 尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。
18. 尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。
19. 任何地方都不要使用 select * from t ，用具体的字段列表代替“*”，不要返回用不到的任何字段。
20. 尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。
21. 避免频繁创建和删除临时表，以减少系统表资源的消耗。
22. 临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，最好使用导出表。
23. 在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。
24. 如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop

table ，这样可以避免系统表的较长时间锁定。

25. 尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。
26. 使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。
27. 与临时表一样，游标并不是不可使用。对小型数据集使用 FAST_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。
28. 在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF。无需在执行存储过程和触发器的每个语句后向客户端发送DONE_IN_PROC 消息。
29. 尽量避免大事务操作，提高系统并发能力。
30. 尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

limit 20000 加载很慢怎么解决

树状结构的MYSQL存储

MongoDB

MongoDB

为什么我们要使用MongoDB？

特点:高性能、易部署、易使用，存储数据非常方便。

主要功能特性有:

- 面向集合存储，易存储对象类型的数据。
- 模式自由。
- 支持动态查询。
- 支持完全索引，包含内部对象。
- 支持查询。
- 支持复制和故障恢复。
- 使用高效的二进制数据存储，包括大型对象（如视频等）。
- 自动处理碎片，以支持云计算层次的扩展性
- 支持Python，PHP，Ruby，Java，C，C#，Javascript，Perl及C++语言的驱动程序，社区中也提供了对Erlang及.NET等平台的驱动程序。
- 文件存储格式为BSON（一种JSON的扩展）。
- 可通过网络访问。

功能:

- 面向集合的存储：适合存储对象及JSON形式的数据。
- 动态查询：Mongo支持丰富的查询表达式。查询指令使用JSON形式的标记，可轻易查询文档中内嵌的对象及数组。
- 完整的索引支持：包括文档内嵌对象及数组。Mongo的查询优化器会分析查询表达式，并生成一个高效的查询计划。
- 查询监视：Mongo包含一个监视工具用于分析数据库操作的性能。
- 复制及自动故障转移：Mongo数据库支持服务器之间的数据复制，支持主-从模式及服务器之间的相互复制。复制的主要目标是提供冗余及自动故障转移。
- 高效的传统存储方式：支持二进制数据及大型对象（如照片或图片）
- 自动分片以支持云级别的伸缩性：自动分片功能支持水平的数据库集群，可动态添加额外的机器。

适用场合:

- 网站数据：Mongo非常适合实时的插入，更新与查询，并具备网站实时数据存储所需的复制及高度伸缩性。
- 缓存：由于性能很高，Mongo也适合作为信息基础设施的缓存层。在系统重启之后，由Mongo搭建的持久化缓存层可以避免下层的数据源 过载。
- 大尺寸，低价值的数据：使用传统的关系型数据库存储一些数据时可能会比较昂贵，在此之前，很多时候程序员往往会选择传统的文件进行存储。
- 高伸缩性的场景：Mongo非常适合由数十或数百台服务器组成的数据库。Mongo的路线图中已经包含对

MapReduce引擎的内置支持。

- 用于对象及JSON数据的存储：Mongo的BSON数据格式非常适合文档化格式的存储及查询。

MongoDB要注意的问题

1. 因为MongoDB是全索引的，所以它直接把索引放在内存中，因此最多支持2.5G的数据。如果是64位的会更多。
2. 因为没有恢复机制，因此要做好数据备份
3. 因为默认监听地址是127.0.0.1，因此要进行身份验证，否则不够安全；如果是自己使用，建议配置成localhost主机名
4. 通过GetLastError确保变更。（这个不懂，实际中没用过）

ObjectId 规则

Redis

Redis

使用redis有哪些好处？

1. 速度快，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)
2. 支持丰富数据类型，支持string，list，set，sorted set，hash
3. 支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行
4. 丰富的特性：可用于缓存，消息，按key设置过期时间，过期后将会自动删除

redis相比memcached有哪些优势？

1. memcached所有的值均是简单的字符串，redis作为其替代者，支持更为丰富的数据类型
2. redis的速度比memcached快很多
3. redis可以持久化其数据

redis常见性能问题和解决方案:

1. Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件
2. 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次
3. 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内
4. 尽量避免在压力很大的主库上增加从库
5. 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3…

这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变。

redis 最适合的场景

Redis最适合所有数据in-momory的场景，虽然Redis也提供持久化功能，但实际更多的是一个disk-backed的功能，跟传统意义上的持久化有比较大的差别，那么可能大家就会有疑问，似乎Redis更像一个加强版的Memcached，那么何时使用Memcached,何时使用Redis呢?

如果简单地比较Redis与Memcached的区别，大多数都会得到以下观点：

1. Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，zset，hash等数据结构的存储。
2. Redis支持数据的备份，即master-slave模式的数据备份。
3. Redis支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。

（1）会话缓存（Session Cache）
最常用的一种使用Redis的情景是会话缓存（session cache）。用Redis缓存会话比其他存储（如Memcached）

的优势在于：Redis提供持久化。当维护一个不是严格要求一致性的缓存时，如果用户的购物车信息全部丢失，大部分人都会不高兴的，现在，他们还会这样吗？
幸运的是，随着 Redis 这些年的改进，很容易找到怎么恰当的使用Redis来缓存会话的文档。甚至广为人知的商业平台Magento也提供Redis的插件。

（2）全页缓存（FPC）
除基本的会话token之外，Redis还提供很简便的FPC平台。回到一致性问题，即使重启了Redis实例，因为有磁盘的持久化，用户也不会看到页面加载速度的下降，这是一个极大改进，类似PHP本地FPC。
再次以Magento为例，Magento提供一个插件来使用Redis作为全页缓存后端。
此外，对WordPress的用户来说，Pantheon有一个非常好的插件 wp-redis，这个插件能帮助你以最快速度加载你曾浏览过的页面。

（3）队列
Reids在内存存储引擎领域的一大优点是提供 list 和 set 操作，这使得Redis能作为一个很好的消息队列平台来使用。Redis作为队列使用的操作，就类似于本地程序语言（如Python）对 list 的 push/pop 操作。
如果你快速的在Google中搜索“Redis queues”，你马上就能找到大量的开源项目，这些项目的目的就是利用Redis创建非常好的后端工具，以满足各种队列需求。例如，Celery有一个后台就是使用Redis作为broker，你可以从这里去查看。

（4）排行榜/计数器
Redis在内存中对数字进行递增或递减的操作实现的非常好。集合（Set）和有序集合（Sorted Set）也使得我们在执行这些操作的时候变的非常简单，Redis只是正好提供了这两种数据结构。所以，我们要从排序集合中获取到排名最靠前的10个用户–我们称之为“user_scores”，我们只需要像下面一样执行即可：
当然，这是假定你是根据你用户的分数做递增的排序。如果你想返回用户及用户的分数，你需要这样执行：
ZRANGE user_scores 0 10 WITHSCORES
Agora Games就是一个很好的例子，用Ruby实现的，它的排行榜就是使用Redis来存储数据的，你可以在这里看到。

（5）发布/订阅
最后（但肯定不是最不重要的）是Redis的发布/订阅功能。发布/订阅的使用场景确实非常多。我已看见人们在社交网络连接中使用，还可作为基于发布/订阅的脚本触发器，甚至用Redis的发布/订阅功能来建立聊天系统！（不，这是真的，你可以去核实）。

Redis提供的所有特性中，我感觉这个是喜欢的人最少的一个，虽然它为用户提供如果此多功能。

redis的一些其他特点

（1）Redis是单进程单线程的
redis利用队列技术将并发访问变为串行访问，消除了传统数据库串行控制的开销

（2）读写分离模型
通过增加Slave DB的数量，读的性能可以线性增长。为了避免Master DB的单点故障，集群一般都会采用两台Master DB做双机热备，所以整个集群的读和写的可用性都非常高。
读写分离架构的缺陷在于，不管是Master还是Slave，每个节点都必须保存完整的数据，如果在数据量很大的情况下，集群的扩展能力还是受限于单个节点的存储能力，而且对于Write-intensive类型的应用，读写分离架构并不适合。

（3）数据分片模型

为了解决读写分离模型的缺陷，可以将数据分片模型应用进来。
可以将每个节点看成都是独立的master，然后通过业务实现数据分片。
结合上面两种模型，可以将每个master设计成由一个master和多个slave组成的模型。

（4）Redis的回收策略

1. volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
2. volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
3. volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
4. allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰
5. allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
6. no-enviction（驱逐）：禁止驱逐数据

注意这里的6种机制，volatile和allkeys规定了是对已设置过期时间的数据集淘汰数据还是从全部数据集淘汰数据，后面的lru、ttl以及random是三种不同的淘汰策略，再加上一种no-enviction永不回收的策略。

使用策略规则：

1. 如果数据呈现幂律分布，也就是一部分数据访问频率高，一部分数据访问频率低，则使用allkeys-lru
2. 如果数据呈现平等分布，也就是所有的数据访问频率都相同，则使用allkeys-random

mySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据

相关知识：redis 内存数据集大小上升到一定大小的时候，就会施行数据淘汰策略。redis提供6种数据淘汰策略见上面一条

假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如果将它们全部找出来？

使用keys指令可以扫出指定模式的key列表。

对方接着追问：如果这个redis正在给线上的业务提供服务，那使用keys指令会有什么问题？

这个时候你要回答redis关键的一个特性：redis的单线程的。keys指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用scan指令，scan指令可以无阻塞的提取出指定模式的key列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用keys指令长。

Redis 常见的性能问题都有哪些？如何解决？

1. Master写内存快照，save命令调度rdbSave函数，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务，所以Master最好不要写内存快照。
2. Master AOF持久化，如果不重写AOF文件，这个持久化方式对性能的影响是最小的，但是AOF文件会不断增大，AOF文件过大会影响Master重启的恢复速度。Master最好不要做任何持久化工作，包括内存快照和AOF日志文件，特别是不要启用内存快照做持久化,如果数据比较关键，某个Slave开启AOF备份数据，策略为每秒同步一次。

3. Master调用BGREWRITEAOF重写AOF文件，AOF在重写的时候会占大量的CPU和内存资源，导致服务load过高，出现短暂服务暂停现象。
4. Redis主从复制的性能问题，为了主从复制的速度和连接的稳定性，Slave和Master最好在同一个局域网内

Redis有哪些数据结构？

字符串String、字典Hash、列表List、集合Set、有序集合SortedSet。

如果你是Redis中高级用户，还需要加上下面几种数据结构HyperLogLog、Geo、Pub/Sub。

如果你说还玩过Redis Module，像BloomFilter，RedisSearch，Redis-ML，面试官得眼睛就开始发亮了。

使用过Redis分布式锁么，它是什么回事？

先拿setnx来争抢锁，抢到之后，再用expire给锁加一个过期时间防止锁忘记了释放。

这时候对方会告诉你说你回答得不错，然后接着问如果在setnx之后执行expire之前进程意外crash或者要重启维护了，那会怎么样？

这时候你要给予惊讶的反馈：唉，是喔，这个锁就永远得不到释放了。紧接着你需要抓一抓自己得脑袋，故作思考片刻，好像接下来的结果是你主动思考出来的，然后回答：我记得set指令有非常复杂的参数，这个应该是可以同时把setnx和expire合成一条指令来用的！对方这时会显露笑容，心里开始默念：摁，这小子还不错。

使用过Redis做异步队列么，你是怎么用的？

一般使用list结构作为队列，rpush生产消息，lpop消费消息。当lpop没有消息的时候，要适当sleep一会再重试。

如果对方追问可不可以不用sleep呢？list还有个指令叫blpop，在没有消息的时候，它会阻塞住直到消息到来。

如果对方追问能不能生产一次消费多次呢？使用pub/sub主题订阅者模式，可以实现1:N的消息队列。

如果对方追问pub/sub有什么缺点？在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如rabbitmq等。

如果对方追问redis如何实现延时队列？我估计现在你很想把面试官一棒打死如果你手上有一根棒球棍的话，怎么问的这么详细。但是你很克制，然后神态自若的回答道：使用sortedset，拿时间戳作为score，消息内容作为key调用zadd来生产消息，消费者用zrangebyscore指令获取N秒之前的数据轮询进行处理。

到这里，面试官暗地里已经对你竖起了大拇指。但是他不知道的是此刻你却竖起了中指，在椅子背后。

如果有大量的key需要设置同一时间过期，一般需要注意什么？

如果大量的key过期时间设置的过于集中，到过期的那个时间点，redis可能会出现短暂的卡顿现象。一般需要在时间上加一个随机值，使得过期时间分散一些。

为什么Redis需要把所有数据放到内存中？

Redis为了达到最快的读写速度将数据都读到内存中，并通过异步的方式将数据写入磁盘。所以redis具有快速和数据持久化的特征。如果不将数据放在内存中，磁盘I/O速度为严重影响redis的性能。在内存越来越便宜的今天，redis将会越来越受欢迎。
如果设置了最大使用的内存，则数据已有记录数达到内存限值后不能继续插入新值。

Redis 持久化机制

bgsave做镜像全量持久化，aof做增量持久化。因为bgsave会耗费较长时间，不够实时，在停机的时候会导致大量丢失数据，所以需要aof来配合使用。在redis实例重启时，会使用bgsave持久化文件重新构建内存，再使用aof重放近期的操作指令来实现完整恢复重启之前的状态。

对方追问那如果突然机器掉电会怎样？取决于aof日志sync属性的配置，如果不要求性能，在每条写指令时都sync一下磁盘，就不会丢失数据。但是在高性能的要求下每次都sync是不现实的，一般都使用定时sync，比如1s1次，这个时候最多就会丢失1s的数据。

对方追问bgsave的原理是什么？你给出两个词汇就可以了，fork和cow。fork是指redis通过创建子进程来进行bgsave操作，cow指的是copy on write，子进程创建后，父子进程共享数据段，父进程继续提供读写服务，写脏的页面数据会逐渐和子进程分离开来。

Redis提供了哪几种持久化方式？

1. RDB持久化方式能够在指定的时间间隔能对你的数据进行快照存储。
2. AOF持久化方式记录每次对服务器写的操作，当服务器重启的时候会重新执行这些命令来恢复原始的数据，AOF命令以redis协议追加保存每次写的操作到文件末尾。Redis还能对AOF文件进行后台重写，使得AOF文件的体积不至于过大。
3. 如果你只希望你的数据在服务器运行的时候存在，你也可以不使用任何持久化方式。
4. 你也可以同时开启两种持久化方式， 在这种情况下， 当redis重启的时候会优先载入AOF文件来恢复原始的数据，因为在通常情况下AOF文件保存的数据集要比RDB文件保存的数据集要完整。
5. 最重要的事情是了解RDB和AOF持久化方式的不同，让我们以RDB持久化方式开始。

如何选择合适的持久化方式？

一般来说， 如果想达到足以媲美PostgreSQL的数据安全性， 你应该同时使用两种持久化功能。如果你非常关心你的数据， 但仍然可以承受数分钟以内的数据丢失，那么你可以只使用RDB持久化。

有很多用户都只使用AOF持久化，但并不推荐这种方式：因为定时生成RDB快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比AOF恢复的速度要快，除此之外， 使用RDB还可以避免之前提到的AOF程序的bug。

Pipeline有什么好处，为什么要用pipeline？

可以将多次IO往返的时间缩减为一次，前提是pipeline执行的指令之间没有因果相关性。使用redis-benchmark

进行压测的时候可以发现影响redis的QPS峰值的一个重要因素是pipeline批次指令的数目。

Redis的同步机制了解么？

Redis可以使用主从同步，从从同步。第一次同步时，主节点做一次bgsave，并同时将后续修改操作记录到内存buffer，待完成后将rdb文件全量同步到复制节点，复制节点接受完成后将rdb镜像加载到内存。加载完成后，再通知主节点将期间修改的操作记录同步到复制节点进行重放就完成了同步过程。

Redis 集群方案与实现

Redis Sentinal着眼于高可用，在master宕机时会自动将slave提升为master，继续提供服务。

Redis Cluster着眼于扩展性，在单个redis内存不足时，使用Cluster进行分片存储。

一个Redis实例最多能存放多少的keys？List、Set、Sorted Set他们最多能存放多少元素？

理论上Redis可以处理多达232的keys，并且在实际中进行了测试，每个实例至少存放了2亿5千万的keys。我们正在测试一些较大的值。

任何list、set、和sorted set都可以放232个元素。

换句话说，Redis的存储极限是系统中的可用内存值。

Redis持久化数据和缓存怎么做扩容？

- 如果Redis被当做缓存使用，使用一致性哈希实现动态扩容缩容。
- 如果Redis被当做一个持久化存储使用，必须使用固定的keys-to-nodes映射关系，节点的数量一旦确定不能变化。否则的话(即Redis节点需要动态变化的情况），必须使用可以在运行时进行数据再平衡的一套系统，而当前只有Redis集群可以做到这样。

Redis 为什么是单线程的

消息队列

MQ基础

消息队列基础

消息队列的使用场景

消息的重发补偿解决思路

消息的幂等性解决思路

消息的堆积解决思路

自己如何实现消息队列

如何保证消息的有序性

如何解决消息队列丢失消息和重复消费问题

异步队列怎么实现

分布式

分布式

分布式全局ID生成方案

参考：高并发分布式系统中生成全局唯一Id汇总

分布式事务

参考：浅谈分布式事务

session 分布式处理

第一种：粘性session。粘性Session是指将用户锁定到某一个服务器上（通过NG）。

第二种：服务器session复制。

第三种：session共享机制。使用分布式缓存方案比如memcached、Redis，但是要求Memcached或Redis必须是集群。

第四种：session持久化到数据库。

谈谈业务中使用分布式的场景

分布式锁的场景

分布是锁的实现方案

集群与负载均衡的算法与实现

说说分库与分表设计

分库与分表带来的分布式困境与应对之策

分布式寻址方式都有哪些算法知道一致性hash吗？你若userId取摸分片，那我要查一段连续时间里的数据怎么办？

分布式缓存

1、redis和memcheched 什么区别为什么单线程的redis比多线程的memched效率要高啊?

2、redis有什么数据类型都在哪些场景下使用啊?

3、reids的主从复制是怎么实现的redis的集群模式是如何实现的呢redis的key是如何寻址的啊?

4、使用redis如何设计分布式锁?使用zk可以吗?如何实现啊这两种哪个效率更高啊??

5、知道redis的持久化吗都有什么缺点优点啊? ?具体底层实现呢?

6、redis过期策略都有哪些LRU 写一下java版本的代码吧??

分布式服务框架

1、说一下dubbo的实现过程注册中心挂了可以继续通信吗??

2、zk原理知道吗zk都可以干什么Paxos算法知道吗?说一下原理和实现??

3、dubbo支持哪些序列化协议?hessian 说一下hessian的数据结构PB知道吗为啥PB效率是最高的啊??

4、知道netty吗’netty可以干嘛呀NIO,BIO,AIO 都是什么啊有什么区别啊?

5、dubbo复制均衡策略和高可用策略都有哪些啊动态代理策略呢?

6、为什么要进行系统拆分啊拆分不用dubbo可以吗’dubbo和thrift什么区别啊?

分布式消息队列

1、为什么使用消息队列啊消息队列有什么优点和缺点啊?

2、如何保证消息队列的高可用啊如何保证消息不被重复消费啊

3、kafka ，activemq,rabbitmq ，rocketmq都有什么优点，缺点啊???

4、如果让你写一个消息队列，该如何进行架构设计啊?说一下你的思路

分布式搜索引擎

1、es的工作过程实现是如何的?如何实现分布式的啊

2、es在数据量很大的情况下(数十亿级别)如何提高查询效率啊?

3、es的查询是一个怎么的工作过程?底层的lucence介绍一下呗倒排索引知道吗?es和mongdb什么区别啊都在什么场景下使用啊?

高并发高可用架构设计

1、如何设计一个高并发高可用系统

2、如何限流?工程中怎么做的，说一下具体实现

3、缓存如何使用的缓存使用不当会造成什么后果?

4、如何熔断啊?熔断框架都有哪些?具体实现原理知道吗?

5、如何降级如何进行系统拆分，如何数据库拆分????

分布式架构原理

1、分布式架构演进过程

2、如何把应用从单机扩展到分布式

3、CDN加速静态文件访问

4、系统监控、容灾、存储动态扩容

5、架构设计及业务驱动划分

6、CAP、Base卢纶以及其应用

分布式架构策略

1、 分布式架构网络通信原理剖析

2、通信协议中的序列化和反序列化

3、基于框架RPC技术 Webservice/RMI/hessian

4、基于ZooKeeper实现分布式服务器动态上下线感知

5、深入分析ZooKeeper在disconfi配置中心的应用

6、深入分析ZooKeeper Zab协议及选举机制源码解读

7、Dubbo管理中心及监控平台安装部署

8、基于Dubbo的分布式系统架构实战

9、Dubbo容错机制及高扩展性分析

分布式架构中间件

1、 分布式消息通信ActiveMQ/kafka/rabbitmq

2、Redis主从复制原理及无磁盘复制分析

3、图解Redis中AOF和RDB持久化策略的原理

5、Session跨域共享以及企业级单点登录解决方案实战

6、分布式事务解决方案实战

7、高并发下的服务降级、限流实战

8、基于分布式架构下分布式锁的解决方案实战

9、分布式架构下实现分布式定时调度

分布式锁的应用场景、分布式锁的产生原因、基本概念

分布是锁的常见解决方案

分布式事务的常见解决方案

集群与负载均衡的算法与实现

说说分库与分表设计，可参考《数据库分库分表策略的具体实现方案》

分库与分表带来的分布式困境与应对之策

说说 CAP 定理、 BASE 理论

怎么考虑数据一致性问题

说说最终一致性的实现方案

请解释什么是C10K问题或者知道什么是C10K问题吗？

微服务

微服务

微服务哪些框架

你怎么理解 RPC 框架

说说 RPC 的实现原理

说说 Dubbo 的实现原理

你怎么理解 RESTful

如何理解 RESTful API 的幂等性

可以参考:如何理解RESTful的幂等性

如何保证接口的幂等性

你怎么看待微服务

参考:服务端指南 | 微服务架构概述

微服务与SOA的区别

SOA (Service-Oriented Architecture，面向服务的架构)是一种面向服务的思维方式，它将应用程序的不同功能（服务）通过服务之间定义良好的接口和契约联系起来。SOA 核心思想是服务是一种可重复的业务，将其经过标准封装达到复用的目的。SOA 可以允许各种不同的技术来表达 SOA 的架构理念，而业界比较流行的实现是 WebService，其中 WebService 采用 HTTP 协议传输数据，采用 XML 格式封装数据。微服务架构和 SOA 的思想没有太大的差别，从实现的方式而言，微服务架构强调实现的轻量化，做到服务粒度更细。这里，微服务的“微”指的并不是服务，而实际上是应用粒度。为了更好地识别 SOA 与微服务架构之间的区别，我们来做一个横向对比。

方面	SOA	微服务架构
应用粒度	多个系统整合成一个服务，粒度大	一个系统拆分成多个服务，粒度小
服务架构	企业服务总线（ESB），集中式架构	服务自治，松散式架构

服务规模	服务规模较小	服务规模膨胀
服务部署	单体架构，业务耦合	功能独立，独立部署

总结下，微服务架构可以理解成 SOA 的升级版，强调实现的轻量化，做到服务粒度更细。随着敏捷开发、持续交付、虚拟化技术、DevOps 理论的实践，微服务架构越来越被重视与应用。

如何拆分服务

参考：如何拆分服务

微服务如何进行数据库管理

参考：论微服务的数据库管理

如何应对微服务的链式调用异常

参考：应对微服务的链式调用异常

对于快速追踪与定位问题

参考：如何快速追踪与定位问题

微服务的安全

参考：微服务的安全

说说服务的治理（怎么注册怎么发现）

安全和性能

安全

安全问题

安全要素与 STRIDE 威胁

参考：安全要素与 STRIDE 威胁

防范常见的 Web 攻击

参考：如何防范常见的Web攻击

服务端通信安全攻防

参考：服务端通信安全攻防详解

HTTPS 原理剖析

参考：HTTPS原理剖析与项目场景

HTTPS 降级攻击

参考：HTTPS 降级攻击的场景剖析与解决之道

性能

性能

性能基准

性能优化到底是什么

性能的衡量纬度

网络与服务器

计算机网络

计算机网络

参考：计算机网络

说一下TCP/IP四层？

Nginx

Nginx

什么是Nginx？

Nginx是一个高性能的HTTP和反向代理服务器，也是一个IMAP/POP3/SMTP服务器

Nginx是一款轻量级的Web服务器/反向代理服务器及电子邮件（IMAP/POP3）代理服务器 目前使用的最多的web服务器或者代理服务器，像淘宝、新浪、网易、迅雷等都在使用

为什么要用Nginx？

优点:

- 跨平台、配置简单
- 非阻塞、高并发连接：处理2-3万并发连接数，官方监测能支持5万并发
- 内存消耗小：开启10个nginx才占150M内存 成本低廉：开源
- 内置的健康检查功能：如果有一个服务器宕机，会做一个健康检查，再发送的请求就不会发送到宕机的服务器了。重新将请求提交到其他的节点上。
- 节省宽带：支持GZIP压缩，可以添加浏览器本地缓存
- 稳定性高：宕机的概率非常小
- master/worker结构：一个master进程，生成一个或者多个worker进程
- 接收用户请求是异步的：浏览器将请求发送到nginx服务器，它先将用户请求全部接收下来，再一次性发送给后端web服务器，极大减轻了web服务器的压力
- 一边接收web服务器的返回数据，一边发送给浏览器客户端
- 网络依赖性比较低，只要ping通就可以负载均衡
- 可以有多台nginx服务器
- 事件驱动：通信机制采用epoll模型

为什么Nginx性能这么高？

得益于它的事件处理机制： 异步非阻塞事件处理机制：运用了epoll模型，提供了一个队列，排队解决

Nginx是如何实现高并发的

service nginx start之后，然后输入#ps -ef|grep nginx，会发现Nginx有一个master进程和若干个worker进程，这些worker进程是平等的，都是被master fork过来的。在master里面，先建立需要listen的socket（listenfd），然后再fork出多个worker进程。当用户进入nginx服务的时候，每个worker的listenfd变的可读，并且这些worker会抢一个叫accept_mutex的东西，accept_mutex是互斥的，一个worker得到了，其他的worker就歇菜了。而抢到这个accept_mutex的worker就开始“读取请求–解析请求–处理请求”，数据彻底返回客户端之后（目标网页出现在电脑屏幕上），这个事件就算彻底结束。

nginx用这个方法是底下的worker进程抢注用户的要求，同时搭配“异步非阻塞”的方式，实现高并发量。

为什么不使用多线程？

因为线程创建和上下文的切换非常消耗资源，线程占用内存大，上下文切换占用cpu也很高，采用epoll模型避免了这个缺点

Nginx是如何处理一个请求的呢？

首先，nginx在启动时，会解析配置文件，得到需要监听的端口与ip地址，然后在nginx的master进程里面

先初始化好这个监控的socket(创建socket，设置addrreuse等选项，绑定到指定的ip地址端口，再listen)

然后再fork(一个现有进程可以调用fork函数创建一个新进程。由fork创建的新进程被称为子进程)出多个子进程出来

然后子进程会竞争accept新的连接。此时，客户端就可以向nginx发起连接了。当客户端与nginx进行三次握手，与nginx建立好一个连接后

此时，某一个子进程会accept成功，得到这个建立好的连接的socket，然后创建nginx对连接的封装，即ngx_connection_t结构体

接着，设置读写事件处理函数并添加读写事件来与客户端进行数据的交换。最后，nginx或客户端来主动关掉连接，到此，一个连接就寿终正寝了

正向代理

一个位于客户端和原始服务器(origin server)之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标(原始服务器)

然后代理向原始服务器转交请求并将获得的内容返回给客户端。客户端才能使用正向代理

正向代理总结就一句话：代理端代理的是客户端

反向代理

反向代理（Reverse Proxy）方式是指以代理服务器来接受internet上的连接请求，然后将请求，发给内部网络上的服务器

并将从服务器上得到的结果返回给internet上请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器

反向代理总结就一句话：代理端代理的是服务端

动态资源、静态资源分离

动态资源、静态资源分离是让动态网站里的动态网页根据一定规则把不变的资源和经常变的资源区分开来，动静资源

做好了拆分以后

我们就可以根据静态资源的特点将其做缓存操作，这就是网站静态化处理的核心思路

动态资源、静态资源分离简单的概括是：动态文件与静态文件的分离

为什么要做动、静分离？

在我们的软件开发中，有些请求是需要后台处理的（如：.jsp,.do等等），有些请求是不需要经过后台处理的（如：css、html、jpg、js等等文件）

这些不需要经过后台处理的文件称为静态文件，否则动态文件。因此我们后台处理忽略静态文件。这会有人又说那我后台忽略静态文件不就完了吗

当然这是可以的，但是这样后台的请求次数就明显增多了。在我们对资源的响应速度有要求的时候，我们应该使用这种动静分离的策略去解决

动、静分离将网站静态资源（HTML，JavaScript，CSS，img等文件）与后台应用分开部署，提高用户访问静态代码的速度，降低对后台应用访问

这里我们将静态资源放到nginx中，动态资源转发到tomcat服务器中

负载均衡

负载均衡即是代理服务器将接收的请求均衡的分发到各服务器中

负载均衡主要解决网络拥塞问题，提高服务器响应速度，服务就近提供，达到更好的访问质量，减少后台服务器大并发压力。

性能瓶颈可能出现在哪

- CPU太弱
- worker数量比CPU核心数大太多，导致频繁上下文切换
- 连接数，包括最大连接数，和当前是否由太多连接占着茅坑不拉屎
- 解决方案
    - CPU affinity
    - 提高连接数
    - 正确设置worker数

Nginx几种常见的负载均衡策略

Nginx服务器上的Master和Worker进程分别是什么

使用“反向代理服务器”的优点是什么？

参考：Nginx面试题

Tomcat

Tomcat

Tomcat的基础架构

（Server、Service、Connector、Container）

Tomcat如何加载Servlet的

Pipeline-Valve机制

Netty

软件工程

UML

业务

设计能力

说说你在项目中使用过的 UML 图

你如何考虑组件化

你如何考虑服务化

你如何进行领域建模

你如何划分领域边界

说说你项目中的领域建模

说说概要设计

你系统中的前后端分离是如何做的

说说你的开发流程

你和团队是如何沟通的

你如何进行代码评审

说说你对技术与业务的理解

说说你在项目中经常遇到的 Exception

说说你在项目中遇到感觉最难Bug，怎么解决的

说说你在项目中遇到印象最深困难，怎么解决的

你觉得你们项目还有哪些不足的地方

你是否遇到过 CPU 100% ，如何排查与解决

你是否遇到过 内存 OOM ，如何排查与解决

说说你对敏捷开发的实践

说说你对开发运维的实践

介绍下工作中的一个对自己最有价值的项目，以及在这个过程中的角色

操作系统

操作系统

相关概念

- 同步异步，阻塞非阻塞
    - 同步：等待执行完
    - 异步：不等待执行完就开始执行其他的，例如多线程
    - 阻塞：调用之后不一定会立即返回
    - 非阻塞：调用之后，会立即返回
- 孤儿进程，僵尸进程
    - 孤儿进程是父进程退出的，那个子进程叫做孤儿进程
    - 僵尸进程是子进程退出了但是父进程没有wait的，那个退出的子进程是僵尸进程
- 操作系统调度算法
    - 先来先服务
    - 短作业优先
    - 优先级调度
    - 时间片轮转
    - 多级反馈队列调度算法
    - Linux内核：完全公平调度算法
- 死锁出现条件
    - 互斥
    - 持有并等待
    - 循环等待
    - 没有抢占
- 线程和进程的区别
    - 进程是程序执行的实体，是操作系统提供的一个抽象概念
    - 线程是操作系统调度的最小单元
    - 线程是在进程的空间内，同一进程内的所有线程共享进程的所有资源如fds，stack等
- 进程间通信方式
    - 管道
    - 文件
    - socket
    - 共享内存(mmapped-file也算一种共享内存)

    - 信号量
    - 消息队列

- fork之后子进程从父进程继承的东西

    - real user id, real group id, effective id, effective group id
    - process group id
    - session id
    - controlling terminal
    - set-user, set-group flags
    - current working directory
    - file mode creation mask
    - environment
    - memory mappings
    - resource limits
    - opened file descriptors

- fork之后父子进程的区别

    - the return value from fork
    - parent pid
    - pid

- fork之后只有调用fork的线程被保存，其他线程被销毁。但是其他属性如opened file descriptors，锁等会被保留

- 线程间同步方式

    - mutex
    - read-write lock & lock
    - condition variables
    - spin locks
    - barriers(内存屏障)

- 分页，分段

    - 早期直接使用物理内存，缺点：
        - 地址空间不隔离
        - 内存使用效率低，一个程序执行时，需要将整个程序载入内存，由于程序的地址空间是连续的，如果忽然要执行C，可能就要将已有的程序A放到磁盘。
        - 程序的运行地址不确定
    - 虚拟地址
        - 不直接使用物理内存，而是通过映射。这样解决了程序运行地址不确定的问题
        - 分段：基本思路是把一段程序所需要的内存空间大小的虚拟空间映射到某个地址，然后从物理地址找一块一样大小的地址，进行映射。解决了上面所说的第一个和第三个问题。
            - A程序和B程序分别被映射到了不痛得物理空间区域，他们没有任何重叠
            - 无论实际上被分配到哪一个物理区域，对于程序来说都是透明的，程序不需要关心物理地址的变化，只需要按照地址从0x00000000到0x00A000000写程序即可
        - 分页。根据程序的局部性原理，一个程序运行时，总是频繁的用到其中一小段数据。分页的基本方

法是把地址空间人为的分为固定的大小，每一页的大小由硬件决定，通常是4KB-4MB。当我们把进程的虚拟空间按照分页之后，就可以只加载其中一部分，从而提高了内存的使用效率。

        - 页错误：当进程需要用到某个分页的内容，但是却不在内存里，硬件就会报页错误。然后由操作系统接管进程，负责将需要的页加到内存然后继续后面的动作。

- 内核线程与用户线程的三种模型
    - 一对一
    - 一对多
    - 多对多