1.设计模式相关问题:

* 请用一个例子表明，全局对象是邪恶的存在。

|  |
| --- |
| * 1.全局变量会污染共享命名空间,可能导致意外的命名冲突,数据被修改导致功能异常 * 2.不利于模块化,全部变量会导致程序中独立组件间出现不必要耦合 * 3.全局对象/变量如果依赖另一全局变量的初始化,会导致各种异常问题.统一份文件的全局变量,初始化顺序和声明顺序有关(和编译器有关) |

假设你工作的系统不支持事务性，你会如何从头开始实现它？

* 什么是好莱坞原则（Hollywood Principles）？

|  |
| --- |
| * 模板方法模式:IOC是inversion of control的简称,IOC原理基于好莱坞原则,所有的组件都是被动的(Passive),所有组件初始化和调用都由容器负责 * Framework 使用好莱坞原则目的: * 1.对基于接口编程的支持 * 2.减少单件和抽象工厂的依赖 * 3.降低业务和框架的耦合 * 4.业务组件可复用,可插拔 |
| * Don’t call me;I’ll call you * 类似现实生活中,乘客/you打车到某地,沿途问司机/me某个景点,天经地义;但是乘客/you不要从上车的第一秒开始,时时刻刻或者每隔5秒问一下司机/me;到了目的地没有; 即好莱坞原则中的me是指 下层模块 * 通常,client即you(上层模块)调用下层server即me天经地义,但是对于某些方法,不要轮询/骚扰我, 由我通知你 * 好莱坞原则,回调,并不会改变分层结构的依赖方向 -- 上层依赖与下层 * 如果不采用通知方式 -- 即不应用好莱坞原则,上层可以轮询 * 好莱坞原则的核心: 以通知替代轮询 * https://blog.csdn.net/yqj2065/article/details/8758101 |

关于迪米特法则(最少知识原则): 写一段代码违反它, 然后修复它。

（注: 迪米特法则：the Law of Demeter, 最少知识原则： the Principle of Least Knowledge）

|  |
| --- |
| 一个实体应当尽量少的与其他实体之间发生相互作用,使得系统功能模块相对独立,也就是说一个软件实体应当尽可能少的与其他实体发生相互作用.这样,档一个模块修改时,就会尽量少的影响其他的模块,扩展会相对容易,这是对冉健实体之间通信的限.他要求限制软件实体之间的通信的深度和宽度 |
| 迪米特法则初衷是为了降低类之间的耦合,由于每个类都减少了不必要的依赖,因此可以降低耦合关系.虽然可以避免与非直接的类通信,但是要通信,必然会通过一个中介来发生联系.过分的使用迪米特原则,会产生大量这样的中介和传递类,导致系统复杂度变大. 所以在使用迪米特法则时要做到结构清晰,又要高内聚低耦合. |
| 核心思想是:低耦合 高内聚  <https://www.jianshu.com/p/30931aab5ea0>  <https://www.jianshu.com/p/15edb371c0b5>  <https://coolshell.cn/articles/4535.html> |
| 对于 对象”O”中一个方法 “M”, M应该只能够访问一下对象中的方法  1. 对象O 2. 与O直接相关的 component object  3. 由方法M创建或者实例化的对象 4. 作为方法M的参数的对象 |

* Active-Record模式有什么限制和缺陷？

|  |
| --- |
| ORM:  编程时候一般采用面向对象的方式 , 而对于数据库来说 ORM 就是一个代替你去写sql , 作为一个中介的角色 , 让面向对象和数据库能够和谐共处 . 好比在商店购买东西 , 售货员会帮你到仓库提货 , 你只需要告诉他需要什么就可以了 . ORM不仅可以取数据 , 增删改查都可以 |
| 实现ORM有两种流行的模式 : Active Record 另一种 Data Mapper |
| Active Record : 活动记录模式 (领域模型 模式)  Row Gateway : 数据源 模式  Data Mapper : 数据映射器 |
| ORM模型 : 表映射到记录 , 记录映射到对象 , 字段映射到对象熟悉 |
| ActiveRecord的主要思想是：  1. 每一个数据库表对应创建一个类，类的每一个对象实例对应于数据库中表的一行记录；通常表的每个字段在类中都有相应的Field；  2. ActiveRecord同时负责把自己持久化，在ActiveRecord中封装了对数据库的访问，即CURD;；  3. ActiveRecord是一种领域模型(Domain Model)，封装了部分业务逻辑；  ActiveRecord比较适用于：  1. 业务逻辑比较简单，当你的类基本上和数据库中的表一一对应时, ActiveRecord是非常方便的，即你的业务逻辑大多数是对单表操作；  2. 当发生跨表的操作时, 往往会配合使用事务脚本(Transaction Script)，把跨表事务提升到事务脚本中；  3. ActiveRecord最大优点是简单, 直观。 一个类就包括了数据访问和业务逻辑. 如果配合代码生成器使用就更方便了；  这些优点使ActiveRecord特别适合WEB快速开发。  ActiveRecord不适合于：  1. ActiveRecord虽然有业务逻辑, 但基本上都是基于单表的. 跨表逻辑一般会放到当发生跨表的操作时, 往往会配合使用事务脚本(Transaction Script)中. 如果对象间的关联越来越多, 你的事务脚本越来越庞大, 重复的代码越来越多, 你就要考虑Domain Model + O/R Mapper了;  2. ActiveRecord保存了数据, 使它有时候看上去像数据传输对象(DTO). 但是ActiveRecord有数据库访问能力, 不要把它当DTO用. 尤其在跨越进程边界调用的时候, 不能传递ActiveRecord对象。 |

* Data-Mapper模式和Active-Record模式有什么区别？

|  |
| --- |
| Active Record（活动记录模式）是领域模型对象字段和数据表字段之间存储1:1的关系，也就是一个模型字段对应一个数据表字段；然后模型对象提供一个save()方法用来将模型对象持久化到存储层中去；模型是知道数据层的，也就是和数据持久层耦合的。 |
| Data Mapper（数据映射模式）则是将领域模型对象和数据表完全松耦合，  领域对象只负责处理业务逻辑，根本不知道数据层，也就是和数据层是解耦的；使用一个实体管理器来将模型对象持久化到存储层中；模型对象的字段可以是任何名 称，只要符合业务模型即可，可以映射到数据层数据表的不同字段 |
| Active Record比较简单，但是不够灵活，而Data Mapper则是很灵活，但是多了一个实体管理器，增加了复杂性。 |

* 空对象模式(Null Object Pattern)的目的是什么？

|  |
| --- |
| 一个空对象取代NULL对象实例的检查 . NULL对象不是检查空值 , 而是反应一个不做任何动作的关系 . 这样的NULL对象也可以在数据不可用的时候提供默认的行为 . 在空对象模式中 , 我们创建一个指定各种要执行的操作的抽象类和扩展该类的实体类 , 还创建一个未对该类座任何实现的空对象类 , 该空对象将无缝地使用在需要检查空值的地方 |
| 优点 : 使用对象时无需检查空值做特殊处理  缺点 : 使用时不知道是空值 , 如果接口类的需要使用异常等不能使用 ; 增加了结构和层次的复杂性 |
| <https://blog.csdn.net/qiumengchen12/article/details/44923139> |

* 为什么组合(Composition)比继承(Inheritance)更好？

|  |
| --- |
| 继承的优点是 子类可以重写父类的方法来方便地实现对父类的扩展  缺点 : 父类的内部细节对子类可见  子类从父类继承的方法在编译时就确定下来 , 无法在运行期间改变从父类继承的方法的行为  如果对父类的方法做了修改的话 , 则子类的方法必须做出相应的修改 , 导致两者高度耦合 , 违背面向对象思想 |
| 组合也就是设计类的时候把要组合的类的对象加入到该类中作为自己的成员变量  优点 :  当前对象只能通过所包含的那个对象去调用其方法 , 所以所包含的对象的内部细节对当前对象是不可见的  当前对象与包含的对象是一低耦合关系 , 如果修改包含对象的类中代码不需要修改当前对象的代码  当前对象可以在运行时动态的绑定所包含的对象 , 可以通过set方法所包含对象赋值  缺点 :  容易产生过多的对象 为了能组合多个对象 , 必须仔细对接口进行定义 |
| 组合比继承更具有灵活性和稳定性 |
| 在继承的内存分析中可以看到，继承体系中某个子类的实例化是需要通过把所有直接/间接父类实例化完成之后，才完成其实例化的，如果遇到继承体系很“臃肿”的情况，那么实例化该类所消耗的资源比组合消耗的资源大得多 |
| 组合和继承都允许你在新类中设置子类对象，只不过组合是以明显的方式，继承使用含蓄的方式 |
| 组合经常使用于你想在你的新类中使用一个已有类的功能而不是它的结构（interface）。即你在新类中嵌入了它的对象，所以你在新类中可以使用这个被嵌入对象的功能。但是为了不让新类的使用者看到嵌入类的结构，而仅仅看到新类的结构，应该把嵌入对象设置成私有的 |
| [http://xinpaninr.com/2016/08/18/%E9%80%89%E6%8B%A9%E7%BB%A7%E6%89%BF(inheritance)%E8%BF%98%E6%98%AF%E7%BB%84%E5%90%88(composition)%EF%BC%9F/](http://xinpaninr.com/2016/08/18/选择继承(inheritance)还是组合(composition)？/) |

* 什么是反腐败层(Anti-corruption Layer)?
* 你可以写一个线程安全的单例(Singleton)类吗？
* Could you implement Objects in terms of Higher Order Functions, and vice-versa?
* 数据抽象(Data Abstraction)能力是指能改变实现而不影响客户端的这种能力。请构造一个一个例子，违反这个特性，并且尝试修复它。
* 你是如何处理依赖关系地狱(Dependency Hell)的？
* 为什么说goto语句是恶魔般的存在？

2. 语言相关问题:

* 告诉我你的首选语言的三个最坏的缺陷。
* 为什么现在函数式编程这么越来越受关注？

|  |
| --- |
| 定义:  简单来说 , “函数式编程” 是一种 编程范式 也就是如何编写程序的方法论.  　　(1 + 2) \* 3 - 4  传统的过程式编程，可能这样写：  　　var a = 1 + 2;  　　var b = a \* 3;  　　var c = b - 4;  函数式编程要求使用函数，我们可以把运算过程[定义](http://lostechies.com/derickbailey/2012/01/24/some-thoughts-on-functional-javascript/)为不同的函数，然后写成下面这样：  　　var result = subtract(multiply(add(1,2), 3), 4);  这就是函数式编程。 |
| 特点:  1.函数是第一等公民 即函数和其他数据类型一样 , 处于平等地位 , 可以赋值给其他变量 , 也可以作为参数 , 传入另一个函数 , 或者作为别的函数的返回值  2.只用表达式 , 不用语句 表达式是一个单纯的运算过程 , 总有返回值 . 语句 是执行某种操作 , 没有返回值 . 函数式编程要求 , 每一步都是单纯的运算 而且都有返回值 . 原因是函数式编程一开始就是为了处理运算 , 不考虑系统的读写(I/O) , 语句属于对系统的读写操作 , 所以被排斥在外 . 实际应用中 , 不做I/O不可能的 , 所以函数式编程值要求把 I/O限制到最小 , 不要有不必要的读写行为 , 保持计算过程的的单纯性  3.没有副作用 所谓副作用 是之函数内部与外部活动 , 产生运算意外的其他结果 ; 函数式编程意味着函数要保持独立 , 所有功能就是返回一个新的值 , 没有其他行为 , 尤其是不得修改外部变量的值  4.不修改状态 函数式编程只是返回新的值，不修改系统变量。因此，不修改变量，也是它的一个重要特点。在其他类型的语言中，变量往往用来保存"状态"（state）。不修改变量，意味着状态不能保存在变量中。函数式编程使用参数保存状态，最好的例子就是递归  5.引用透明 指的是函数的运行不依赖于外部变量或者状态 , 只依赖于输入的参数 . |
| 意义  1.代码简洁 开发快速  2.接近自然语言 易于理解  3.更方便的代码管理 每个函数就是独立单元 , 有利于单元测试和排错 , 以及模块化组合  4.易于并发编程 函数式编程不需要考虑死锁 , 因为他不修改变量 , 根本不存在 锁线程 的问题 .  5.代码的热升级 只要保证接口不变 , 内部实现和外部无关 . 所以可以在运行状态下直接升级代码 , 不需要重启 , 不需要停机. |
| <http://www.ruanyifeng.com/blog/2012/04/functional_programming.html>  <https://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/0014317848428125ae6aa24068b4c50a7e71501ab275d52000> |

* 闭包是什么？它有什么用途？闭包和类有什么共同点？

|  |
| --- |
| function makeCounter()  local count = 0  return function()  count = count + 1  return count  end  实际上就是返回一个函数指针，注意的是其中的count变量在外层函数makeCounter返回后不会失效，所以每一次对内层函数的调用都是累计的。 counter = makeCounter() counter() 1 counter() 2 .... 待返回的内层函数可能会引用外层函数的局部变量。闭包就是要把引用的变量的生存期扩大。一种很常见的实现是当调用makeCounter的时候count变量不会失效（但一般据变量都是在栈中存活），所以动态语言实现在makeCounter返回时把count变量拷贝到他出，如一个变量名和值对应的hash表中或是代表当前调用链的链表中。这样就能保证上下文环境在使用期间一直存在。 |
| 闭包等价于类。对于闭包的调用，等价于类的实例的生成。  闭包是一种形式，具体语言支持的形式不一样，但基本含义都一样。  闭包是一个函数体，通常是将这个函数体返回给外部（回调也是外部调用过来），外部再来调用这个函数体，好处是这个函数体是可以很方便访问函数体外面的一些变量。 |
| 自由变量 函数可以有他自身作用域以外的参数参与 . 这些在函数作用域以外 , 由函数的外部环境提供的参数就叫做 “ 自由变量 “ 比如 f(x) = x + y 函数 y 就是自由变量  int y = 3  add() { int x = 2; return x + y } |
| 用途:  最大两个用处:  1. 可以读取函数内部的变量  2.让这些变量的值始终保持在内存中 |
| 共同点: |
| <https://edu.aliyun.com/a/66627?spm=5176.11310711.0.0.4EwEmQ>  <http://caibaojian.com/toutiao/5411> |

* 泛型有什么用途？

|  |
| --- |
| 考虑以下场景：您希望开发一个用于在应用中传递对象的容器。但对象类型并不总是相同。因此，需要开发一个能够存储各种类型对象的容器。  鉴于这种情况，要实现此目标，显然最好的办法是开发一个能够存储和检索 Object 类型本身的容器，然后在将该对象用于各种类型时进行类型转换  虽然这个容器会达到预期效果，但就我们的目的而言，它并不是最合适的解决方案。它不是类型安全的，并且要求在检索封装对象时使用显式类型转换，因此有可能引发异常。  可以使用泛型开发一个更好的解决方案，在实例化时为所使用的容器分配一个类型，也称**泛型类型**，这样就可以创建一个对象来存储所分配类型的对象。泛型类型是一种类型参数化的类或接口，这意味着可以通过执行**泛型类型调用**分配一个类型，将用分配的具体类型替换泛型类型。然后，所分配的类型将用于限制容器内使用的值，这样就无需进行类型转换，还可以在编译时提供更强的类型检查。 |
| 使用泛型的好处  一个最重要的好处是更强的类型检查，因为避开运行时可能引发的 **ClassCastException**可以节省时间。  另一个好处是消除了类型转换，这意味着可以用更少的代码，因为编译器确切知道集合中存储的是何种类型。例如，在清单 **4** 所示代码中，我们来看看将 **Object**容器实例存储到集合中与存储 **GenericContainer**实例之间的差异。 |
| **Java**的泛型靠的还是类型擦除，目标代码只会生成一份，牺牲的是运行速度。  **C++**的模板会对针对不同的模板参数静态实例化，目标代码体积会稍大一些，运行速度会快很多。 |

* 什么是高阶函数？有什么用途？用你的首选语言写个例子出来。

|  |
| --- |
| 满足以下条件之一的函数  **1.**函数可以作为参数被传递  **2.**函数可以做为返回值输出 |
| 主要是简化代码 **,** 更加复用 |

* 讨论一下，如何写一个循环，然后把它转换成递归函 数，要避免易变性。
* 有些语言将函数视为第一公民，这是什么意思？

|  |
| --- |
| 第一公民:  一等公民 , First-Class Citizen , 此处Citizen可换为 object/value/entity 等词 . 所谓一等公民即在程序中可无限使用的对象(相比其他对象) . “一等公民”者可以:  1.表示为匿名字面值  2.存储于变量中  3.存储于数据结构中  4.作为函数的参数传递  5.作为函数的返回值  6.在运行时构造 |
| 在编程语言设计中，在给定的编程语言中，一流的公民（类型、对象、实体或值）是支持其他实体通常可用的所有操作的实体。这些操作通常包括传递作为参数、从函数返回并赋值给变量。  In programming language design, a first-class citizen (also type, object, entity, or value) in a given programming language is an entity which supports all the operations generally available to other entities. These operations typically include being passed as an argument, returned from a function, and assigned to a variable. |

* 用一个例子说明匿名函数是有用的。

|  |
| --- |
| 1.避免占用全局变量名 2.参数保护,函数是独立作用域,传递参数可以保护临时变量,闭包可以保存循环中需要保留的临时变量,还有组件开发时将命名空间传递到函数中用闭包保护起来,即使命名空间被后面的代码重置,原变量被闭包保护将仍然生存 3.降低风险,因为没有变量名,中间代码又被闭包保护,js注入无法访问,减少被攻击风险 |
| <https://blog.csdn.net/zhenyu5211314/article/details/51259299> |

* 什么是动态方法调度(Dynamic Method Dispatch)？

|  |
| --- |
| 方法重载的真正意义在于它构成了Java一个最强大的概念基础: 动态方法调度  动态方法调度是一种在程序运行时而不是编译时调用重载方法的机制 , 它是实现运行时多态性的基础 . |

* 名字空间(Namespace)有什么用？有什么可以替代它的吗？

|  |
| --- |
| 主要是为了控制标识符的作用域 |

* 谈谈Java和C#之间的互操作性(Interoperability) (任选其他两门语言都行)
* 为什么很多软件工程师不喜欢Java？
* 你认为好的语言好在哪里？差的语言差在哪里？
* 写两个函数，一个是"引用透明的(Referentially Transparent)"，另一个是"引用不透明的(Referentially Opaque)"。讨论之。

|  |
| --- |
| 引用透明:  函数的返回值只依赖于其输入值 , 这种特性就称为引用透明性  可测试性:  纯函数具有引用透明性 , 也就是说 , 不论何时调用它 , 对于同一组参数它永远给出同样的结果 , 这使它跟那些与其他系统相互交织的东西比起来更易于使用 |

* 什么是栈？什么是堆？

|  |
| --- |
| 内存分配方式:  1.从静态存储区域分配 . 内存在程序编译的时候就已经分配好 , 这块内存在程序的整个运行期间都存在 . 例如 全局变量 static变量  2.在栈上创建 . 在执行函数时 , 函数内局部变量的存储单元都可以在栈上创建 , 函数执行结束时这些存储单元自动被释放 . 栈内存分配运算内置于处理器的指令集中 , 效率很高 , 但分配的内存容量有限  3.从堆上分配 , 亦称动态内存分配 . 程序在运行的时候用 malloc 或者 new 申请任意多的内存 , 程序自己负责 free 或 delete 释放内存 . 如果在堆上分配空间 , 则有责任回收 , 否则运行程序会出现内存泄露 |
| 程序内存空间:  1. 代码区  2.全局数据区  3.堆区 由使用者自行肥培释放  4.栈区 由编译器自动分配释放 , 存放为运行函数尔分配的局部变量 函数参数 返回数据 返回地址等 . 类似数据结构中的栈 |
|  |

* 为什么一个语言中，"函数是第一公民"是很重要的？

|  |
| --- |
| 简单概括来说函数可作为参数传入也可以作为返回值抛出 |
| 函数式编程的特点:  1. 函数是第一公民  2. 无副作用 : 函数副作用是指函数被调用的时候除了给出返回值以外还修改了函数外部的状态  3. 状态不变  4. 描述”做什么 , 是什么”而不是”如何做 , 为何是”  5. 过程分解 , 函数叠加 |
| <https://blog.csdn.net/shenpibaipao/article/details/78622498> |

* 模式匹配(Pattern Matching)和Switch语句(Switch clauses)的区别在哪儿？
* 为什么有些语言设计上没有异常机制？这有什么优缺点？

|  |
| --- |
| 异常处理机制是提高软件可靠性的有效手段 , 在错误检测 , 危险控制和评价 , 错误处理及继续系统服务等方面提供了一定的支持 .  在正常的程序执行流之外开辟了专门的信道用来在不同模块之间报告错误 , 增加了程序的复杂度 , 降低了程序编译时的速度 . |

* 如果Cat是一个Animal, 那么TaskCare<Cat>是一个TakeCare<Animal>吗？

3.数据库相关问题:

* 如果要你将一个项目从MySQL迁移至PostgreSQL中，你会如何迁移？
* 为什么SELECT \* FROM table WHERE field = null不能匹配空的字段？
* 什么是ACID(原子性，一致性，隔离性，持久性)原则？
* 你是如何进行数据库模式(Database schema)迁移的？
* 延迟加载(lazy loading)是如何实现的？什么场景下有用？他有什么缺陷？
* 什么是N+1问题？
* 如何找出应用中开销最大的查询？

4.非关系型数据库相关问题:

* 什么是最终一致性(Eventual Consistency)？
* 关于CAP理论，举一些CP、AP、CA系统的例子。
* NoSQL是如何解决可伸缩性的挑战的？
* 你在什么时候会使用Request/Response模式,什么时候使用Publish/Subscribe模式?

5.分布式系统相关问题:

* 怎么测试一个分布式系统？
* 什么场景下你会在两个系统中采用异步通信机制？
* 远程过程调用的通用缺点是什么？
* 如果你为了可扩展性和鲁棒性而构建一个分布式的系统，分别在封闭安全的网络环境情况下，和地理上的位置不同但是网络环境不是封闭和安全的情况下，你会考虑什么不同的事情？
* 在Web应用中如何管理容错性？在桌面端呢？
* 在分布式系统中，如何处理故障？
* 让我们来谈谈网络分裂(network partitions)后有的几种恢复的手段吧。
* 你认为分布式计算中有哪些谬论？
* 你在什么时候会使用Request/Response模式，什么时候使用Publish/Subscribe模式？

6.逻辑算法相关问题:

* 只用LIFO栈如何构造一个FIFO队列？只用FIFO队列如何构造一个LIFO栈？
* 写一段有栈溢出的代码。
* 写一个尾递归版本的阶乘函数。
* 使用任何一个语言，写一个REPL，功能是echo你输入的字符串。然后将它演化成一个逆波兰表达式的计算器。
* How would you design a "defragger" utility?
* 写一个生成随机迷宫的程序。
* 写一段有内存泄漏的示例代码。
* 随机生成一个的数字序列，里面每个数字都不同。
* 写一个简单的垃圾回收系统。
* 使用任何一门语言，写一个基本的消息代理。
* 写一个基础的web服务器，然后画一张线路图，展示你将来还想要实现的功能。
* 如何对一个10GB的文

7.软件架构相关问题:

* 什么情况下缓存是没用的，甚至是危险的？
* 为什么事件驱动的架构能提高可扩展性(scalability)？
* 什么样的代码是可读性强的代码？
* 紧急设计(Emergent Design)和演化架构(Evolutionary Architecture)之间的区别是什么？
* 横向扩展(scale out) vs 纵向扩展(scale up): 有什么区别？分别在什么场景下使用？
* 分布式系统中如何处理"故障切换(failover)"和"用户会话(user session)"？
* 什么是CQRS(Command Query Responsibility Segregation)?他和最早的Command-Query Separation原则有什么区别？
* 什么是三层架构？
* 如何设计一个可扩展性高的系统？
* 处理C10k问题的策略有哪些？
* 如果让你来设计一个去中心化的P2P系统，你会如何设计？
* 为什么CGI的扩展性不好？
* 在设计系统时，你如何防止供应商依赖([Vendor Lock-in](https://sourcemaking.com/antipatterns/vendor-lock-in))？
* 在可扩展性上，发布/订阅(Publish-Subscribe)模式有什么缺点？
* 80年代以后，CPU有哪些变化？这些变化，对编程产生了什么影响？
* 性能生命周期(performace lifecycle)中，你认为哪个部分是需要考虑进去的？ 如何管理？
* 除了恶意攻击造成的拒绝服务现象以外，哪些设计或者架构上的问题会导致拒绝服务？
* 性能和可扩展性之间有什么关系？
* 什么时候紧耦合是OK的？
* 一个系统要有什么特征才能适配云计算环境(Cloud Ready)？
* Does unity of design imply an aristocracy of architects?

8.面向服务架构相关问题:

* 在SOA中，为什么长期存活的事务(Long-lived transation)不被看好，而Saga却被看好？
* SOA和MicroService之间有什么区别？
* 我们来谈谈Web服务的版本管理、版本兼容性、重大变更管理这些事情吧.
* 在saga中事务和补偿操作(compensation operation)之间的区别是什么？在SOA中呢？
* 微服务不能做得太"微"，你认为什么时候微服务太"微"了？
* MicroService架构的优劣是什么？