1: 软件体系结构是构建计算机软件实践的基础。与建筑师设定建筑项目的设计原则和目标，作为绘图员画图的基础一样，一个软件架构师或者系统架构师陈述软件构架以作为满足不同客户需求的实际系统设计方案的基础。

软件构架是一个容易理解的概念，多数工程师（尤其是经验不多的工程师）会从直觉上来认识它，但要给出精确的定义很困难。

构架不仅是结构。

在 Rational Unified ProcESs 中，软件系统的构架是指系统重要构件的组织或结构，这些重要构件通过接口与不断减小的构件，与接口所组成的构件进行交互。

从与目的、主题、材料和结构的联系上来说，软件架构可以和建筑物的架构相比拟。一个软件架构师需要有广泛的软件理论知识和相应的经验来实施和管理软件产品的高级设计。软件架构师定义和设计软件的模块化，模块之间的交互，用户界面风格，对外接口方法，创新的设计特性，以及高层事物的对象操作、逻辑和流程。

是一般而言，软件系统的架构（Architecture）有两个要素：

·它是一个软件系统从整体到部分的最高层次的划分。

一个系统通常是由元件组成的，而这些元件如何形成、相互之间如何发生作用，则是关于这个系统本身结构的重要信息。

详细地说，就是要包括架构元件（Architecture Component）、联结器（Connector）、任务流（Task-flow）。所谓架构元素，也就是组成系统的核心"砖瓦"，而联结器则描述这些元件之间通讯的路径、通讯的机制、通讯的预期结果，任务流则描述系统如何使用这些元件和联结器完成某一项需求。

·建造一个系统所作出的最高层次的、以后难以更改的，商业的和技术的决定。

在建造一个系统之前会有很多的重要决定需要事先作出，而一旦系统开始进行详细设计甚至建造，这些决定就很难更改甚至无法更改。显然，这样的决定必定是有关系统设计成败的最重要决定，必须经过非常慎重的研究和考察。

2：因为一个架构不好的系统会有很多很多的问题需要改进，而当一个系统部署好做好了之后再进行改进就会有很大的消耗，而在开始做之前做软件构架的分析，不仅可以减少以后在维护和扩展时候的成本，还可以让我们站个一个比较高的角度来看整个系统，并且使其他利益相关者都对系统有了解和认识。并且可以对一个大的系统和项目进行分解，分解成小的模块进行分析和构建。

3：

1. 万维网的商业目标是让所有用户都可以访问到，将资源以超媒体文件（html文件）传输展示给访问的用户。
2. 亚马逊云可在云中提供大小可调的计算容量。该服务旨在降低开发人员进行网络规模级云计算的难度。主要是售卖这种云计算处理能力的服务。
3. 目的是通过利用开源的系统形成无线通讯的产业链，卖点是全球的信息资源共享服务。

4：

1.用例大部分是使用系统时候的一组操作和输入输出，而质量属性场景是一个刺激，刺激源，以及刺激的部分，环境，和应该做出的响应，及响应的度量，对于不同的质量属性，刺激可能不止发生在使用系统的时候，比如可以发生在编码阶段。

要想把一个用例变成一个质量属性场景需要添加：环境，制品，响应度量。

2．使用任何语言都可以实现较好的系统可用性，但是有的语言实现起来比较复杂，可测试性就降低了，如果需要增加某些功能，在有的语言上比较麻烦，有的语言优点是高并发，提升了性能，有的语言有较好的线程安全机制，不同的语言对易用性影响较小，不过有的语言需要依赖于特定的平台，还是有些影响。

5：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Source | Stimulus | Artifact | Environment | Response | Response measure |
| 外部 | 一条记录缺失了必要的字段 | 进程 | 正常 | 发生异常并记录，正常运行 | 无影响 |
| 外部 | 一条请求在约定时间之内没有收到回复 | 会话 | 正常 | 通知重发机制再次发送请求 | 2秒内收到回复 |
| 内部 | 两个进程对同一个资源的访问发生死锁，长时间没有回复 | 进程 | 正常 | 禁用其中一个进程3秒钟 | 3秒钟后恢复使用 |
| 内部 | 某一个进程在无意义死循环 | 进程 | 正常 | 停止该进程 | 不可用 |
| 外部 | 服务器收到一条空的消息 | 会话 | 正常 | 抛弃该消息继续执行 | 无影响 |

2.不是冗余策略的：ping/echo,heartbeat,santy checking,

是冗余策略的：mointor,voting,rollback,shadow,

6:公共服务的方法使得组件独立，每个组件负责单独的责任，单一职责原则，减少了组件与组件之间的耦合，所以组件与组件之间的联系就少了，降低了组件之间的凝聚力。

7：

1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Source | Stimulus | Artifact | Environment | Response | Response measure |
| 外部 | 定期读取一张表中的数据 | 系统 | 正常 | 处理事件 | 不影响正确率 |
| 内部 | 系统读取缓存中的数据 | 系统 | 正常 | 降低服务的等级 | 命中率正常 |

2．许多系统需要实时监测和响应，比如一个查询API不应该有太长的响应时间，应该返回实时的结果，要不然性能就很不好，一般在具体做的时候可以使用反范式设计，将数据库表中的字段冗余减少inner join 来提高查询的效率，减少响应的时间。

8：

1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Source | Stimulus | Artifact | Environment | Response | Response measure |
| 人 | Break CIA 修改了数据 | 数据 | 线上 | 通过备份的记录保持数据的完整性 | 通过一个post请求修改 |
| 人 | Break CIA 窃听了消息 | 数据 | 线上 | 记录中间人的IP地址 |  |
| 系统 | 发现了木马文件 | 服务 | 线下 | 提示用户和操作系统发现病毒。 | 由用户自己下载的文件 |

2.数据要有备份，要不然遭遇突发情况会导致数据丢失。

9：

1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Source | Stimulus | Artifact | Environment | Response | Response measure |
| 测试人员 | 测试方法 | 单元测试 | coding | 单元测试测试一个类中的所有方法，显示测试结果 | 覆盖80%代码 |

2.Control and observe system state :这种方法是通过明确接口，记录，回放，定位存储位置，抽象数据源，沙箱，assert来使结果和过程可见可控。抽象数据源，比如：在测试数据库操作的方法时，不需要连接真实数据库，mock一些假数据来测试方法是否可行。

Limit complexity: 这种方法是在设计的时侯不要让一个模块太过复杂，不要有太多的分支，那样就会提高测试难度。应该明确细化每个模块的作用，减少耦合。避免一个点的修改影响其他功能，设计上应避免太多的回归测试。

10：

1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Source | Stimulus | Artifact | Environment | Response | Response measure |
| 用户 | 用户在旅行网订票 | 系统 | 系统运行 | 用户根据系统提示一步步预订，最后成功了票 | 五分钟之内完成。 |

2．可用性和性能的平衡首先要区分系统的使用场景。不同的使用场景对于性能和可用性的权重是不一样的。比如电子商务网站需要有一定的可用性，避免用户进行误操作，导致出现复杂的问题。在设计的时候应该区分开主要功能模块和支持可用性的模块，减少两个模块之间的耦合，减少可用性模块太过影响性能。此外，我觉得在用户接口的设计上也比较容易提高可用性。这样就可以在不影响性能的情况下提高可用性了。

使用两个栈的结构，用户每输入一个操作数或者操作符就将它压入第一个栈，在用户选择计算结果的时候就出栈一个操作符和两个操作数，将这三个数据压入第二个栈，如果用户选择undo操作，就从第二个栈中出栈两个操作数和一个操作符，压入第一个栈。

13：1.SOA 敏捷性很好，可以更迅速的适应改变。可靠性不是很好，因为难以实现数据的所有特点统一，建立统一的服务接口比较困难。

2.不应该使用外部的扩展服务，因为不是很可靠，不知道那个服务具体怎么做，什么时候会出现问题。