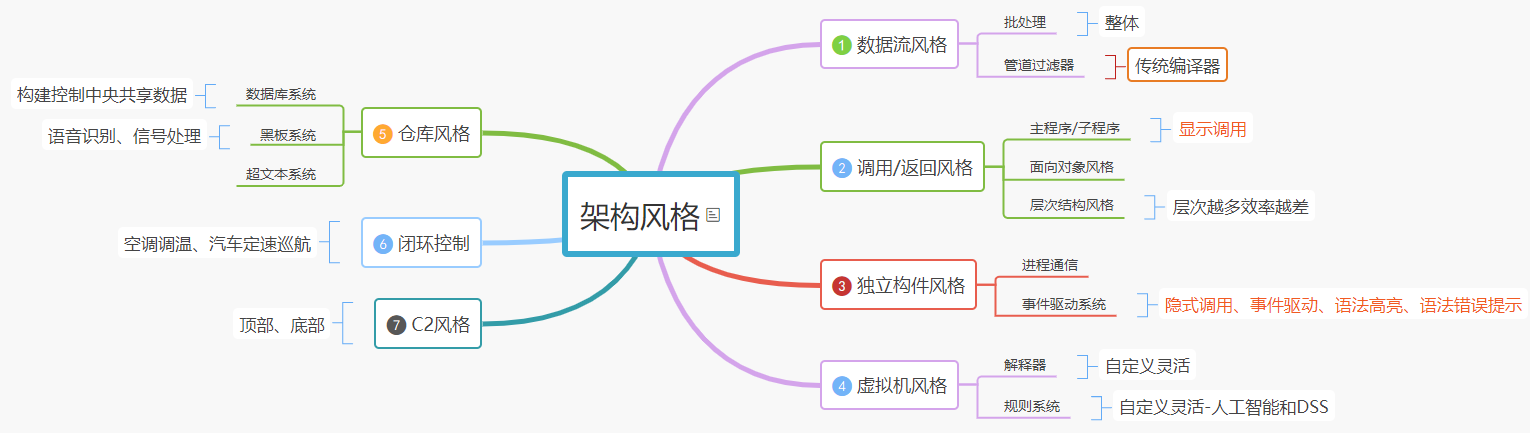
**系统架构设计师案例分析考点**

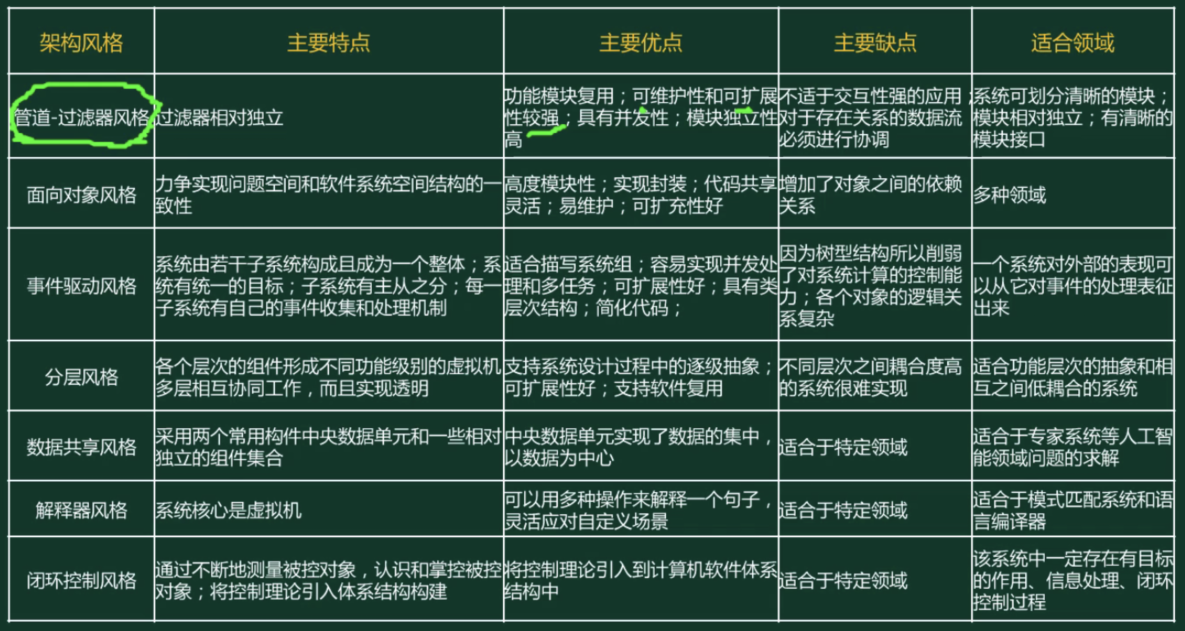
# 一 软件架构风格（重点，今年案例可能考）

**定义：**

软件架构风格是描述某一特定应用领域中系统组织方式的惯用模式。架构风格定义一个系统家族，即一个体系结构定义一个词汇表和一组约束。词汇表中包含一些构件和连接件类型，而这组约束指出系统是如何将这些构件和连接件组合起来的。



|  |  |  |  |
| --- | --- | --- | --- |
| 架构风格大类 | 架构小类 | 构件 | 连接件 |
| 数据流风格 | 批处理序列 | 计算单元 |  |
| 管理过滤器 | 过滤器 | 数据流传输的管道 |
| 调用/返回风格 | 主/子程序 | 主子程序 | 过程调用 |
| 面向对象 | 对象 | 对象间的交付方式 |
| 层次结构 | 每一层 | 层间的交付方式 |
| 独立构件 | 进程通信 | 独立的进程 | 消息传递 |
|  | 事件驱动 | 模块 | 隐式调用 |
| 仓库风格 | 黑板系统 | 知识源 | 黑板系统或数据库系统 |



## 1 数据流风格

* **批处理序列（口决记忆法：构构每每数以）**

**定义：**

构建为一序列**固定顺序**的计算单元，构建之间只通过**数据传递**进行交付，每个处理步骤是一个**独立的程序，**每一步必须在其前一步结束后才能开始，数据必须是完整的，以**整体的方式**进行传递

**特点：**强调整体性，无交互

* **管理过滤器（口决记忆法：每构经然，变换三通）**

**定义：**

* 每个构建都有一组**输入、输出，**构建**读输入**的数据流，经过**内部处理**，然后**产生输出**数据流，这个过程通常是通过对输入数据流的**变换或计算**来完成的，包括：
* 通过计算和增加信息以**丰富数据**
* 通过浓缩和删除以**精简数据**
* 通过改变记录方式以转化数据和递增的**转化数据**
* 构建为过滤器，连接件是数据流传输的管道

**特点：不适合处理交互式应用**

## 2 调用/返回风格

* **主程序/子程序（理解记忆法）**

**定义：**

* 所有的计算构件作为子程序协作工作，并由一个主程序顺序地调用这些子程序，构件通过共享存储区交换数据。
* 调用关系具有层次性，其语义逻辑表现为主程序的正确性取决于它调用的子程序的正确性
* 构件为主程序和子程序，连接件是过程调用
* **面向对象（理解记忆法）**

**定义：**

* 建立在数据抽象和面向对象的基础上，数据的表示和它们的相应操作被封装起来。
* 构件是对象，对象是抽象数据类型的实例，对象之间通过消息机制进行通信，连接件是对象间的交付方式。
* **层次结构（理解记忆法）**

**定义：**

* 每层为上层提供服务，并使用下层提供的服务，一般中间层只对相邻层可见。
* 构件组成一个层次结构，连接件通过层间交互的协议来定义。

## 3 独立构件风格

* **进程通信（理解记忆法）**

**定义：**

* 构件通常是命名过程，消息传递可以是点对点、异步或同步方式、以及远程过程调用等。
* 构件是独立的进程，连接件是消息传递
* **事件驱动系统（隐式调用）（理解记忆法）**

**定义：**

* 构件不直接调用一个过程，而是触发或广播一个或多个事件。构件中的过程在一个或多个事件中注册，当某个事件被触发，系统自动调用在这个事件上注册的所有过程。
* 构件是一些模块，这些模块可以是过程或事件。连接件以过程之间的隐式调用来实现。

**优点：**

* **增加架构的灵活性**

**缺点：**

* 构件放弃了对系统计算的控制
* 共享数据问题

## 4 虚拟机风格

* **解释器（需要自定义业务流程的场景）（理解记忆法）**

**定义：**

* 包括一个完成解释工作的解释引擎，一个记录解释引擎当前工作状态的数据结构，一个包含将被解释代码的存储区，以及一个记录源代码被解释执行进度的数据结构。
* 含有一个虚拟机

**缺点：**

* 执行效率较低
* **基于规则的系统（通过数据分析，产生决策需要的数据）（理解记忆法）**
* 包括规则集、规则解释器、规则/数据选择器和工作内存

## 5 仓库风格

* **数据库系统（类似于目前的多个微服务开发，连共享数据库操作）（理解记忆法）**

**定义：**

* 中央共享数据库，保存当前系统的数据状态、
* 多个独立处理单元，对数据元素进行操作
* **黑板系统（信号处理、问题规划、编译器优化、语音识别）（强行记忆）**

**定义：**

* 由知识源，黑板数据结构，控制3部分组成。黑板数据即共享的中央数据仓库；控制完全由黑板的状态驱动，黑板状态决定使用特定的知识源；知识源通过不断改变黑板数据来解决问题。
* 构件是中央数据结构和外部独立构件即知识源。连接子取决于控制原则的选取，若外部构件控制共享数据，则仓库是传统型数据库；若共享数据当前状态触发控制外部构件，则仓库是黑板系统。
* **超文本系统（强行记忆）**

**定义：**

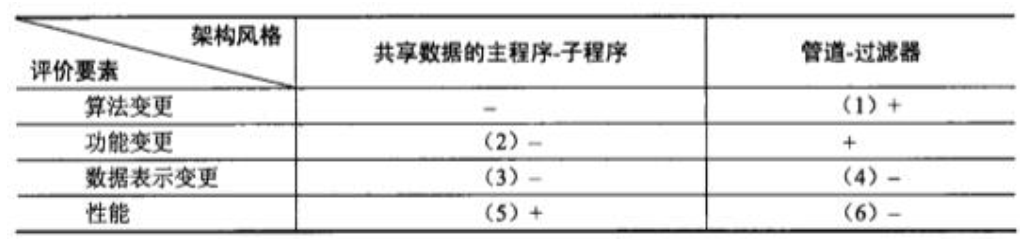
* 构件以网状链接方式相互连接
* 用户可以在构件之间进行按照人类联想思维的方式任意跳转到相关构件。
* 超文本以节点为单位

## 6 闭环控制（过程控制）

**定义：**

* 反馈循环机制
* 是将过程输出的指定属性维护在一个特定的参考值（设定点）。控制环路风格包括输入变量、设定点、操纵变量、过程变量和被控变量等构件，通过收集实际和理想的过程状态信息，并能调整过程变量使得实际状态趋于理想状态。

## 7 主程序/子程序与管道-过滤器的比较



# 二 各种图形

## 1 顺序图

**定义：**是一种交互图，强调对象之间消息发送的顺序，同时显示对象之间的交互。

## 2 活动图

**定义：**将进程或其他计算结构展示为计算内部一步步的控制流和数据流，强调对象间的控制流程。活动图可以用于描述系统的工作流程和并发行为。活动图中一个活动结束后将立即进入下一个活动（在状态图中状态的转移可能需要事件的触发）。活动图侧重描述行为的动作

## 3 状态图

**定义：**状态图主要用于描述一个对象在其生存期间的动态行为，表现一个对象所经历的状态序列，引起状态转移的事件（event），以及因状态转移而伴随的动作（action）。侧重于描述行为的结果

**状态图与活动图的区别：**

状态图侧重于描述行为的结果，而活动图侧重描述行为的动作。其次活动图可描述并发行为，而状态图不能。

## 4 通信图（了解）

**定义：**通信图也是一种交互图，它强调收发消息的对象或参与者的结构组织。

通信图与顺序图的区别：顺序图强调的是时序，而通信图强调的是对象之间的组织结构（或关系）。

## 5 流程图

流程图以图形化的方式展示应用程序从数据输入开始到获得输出为止的逻辑过程，描述  
处理过程的控制流。

## 6 数据流图

### **6.1 定义**

数据流图作为一种图形化工具，用来说明业务处理过程、系统边界内所包含的功能和系  
统中的数据流。

数据流图通过外部代理（实体）描述系统与外界之间的数据交互关系，内部的活动通过处理（加工）表示，用数据流描述系统中不同活动之间的数据传输内容和方向，需要持久化存储的数据用数据存储表示，一般用文件系统或者数据库表存储数据-数据流图中所包含的四种元素：

* 外部实体（External Agent)定义位于项目范围之外，但与正在被研发的系统有交互关系的人、部门、外部系统或组织；
* 加工（Process)在输入数据流或条件上执行，或者对输入数据流或条件做出响应的工作；
* 数据存储（Data Store)描述静止的数据，表示系统中需要保存的数据；
* 数据流（Data Flow)描述运动中的数据，表示到一个过程的数据输入，或者来自一个过程的数据输出。

### **6.2 数据流图和流程图的区别**

* 数据流图中的处理过程可并行；流程图在某个时间点只能处于一个处理过程。
* 数据流图展现系统的数据流；流程图展现系统的控制流。
* 数据流图展现全局的处理过程，过程之间遵循不同的计时标准；流程图中处理过程  
  遵循一致的计时标准。
* 数据流图适用于系统分析中的逻辑建模阶段；流程图适用于系统设计中的物理建模阶段。

### 6.3 数据流图的常见错误

实体->加工->数据存储->加工->实体

实体->实体：X

加工->加工：X

数据存储->数据存储：X

实体->数据存储：X

数据存储->实体：X

* 只有输入没有输出，产生数据黑洞；
* 只有输出没有输入，无中生有；
* 外部实体没有经过加工处理，直接到数据存储;
* 外部实体之间没有加工处理，存在直接数据流。
* 数据存储没有输出的数据流。
* 加工不能只进数据流，同样也不能只出数据流
* 实体与实体之间有数据流
* 实体与数据存储之间有数据流,存储和存储之间有数据流

### 6.4 设计原则

* 复杂性最小化原则。
* 接口最小化原则。
* 数据流一致性原则。

## 7 类图

**类图的关系：**依赖、泛化、关联（组合、聚合）、实现

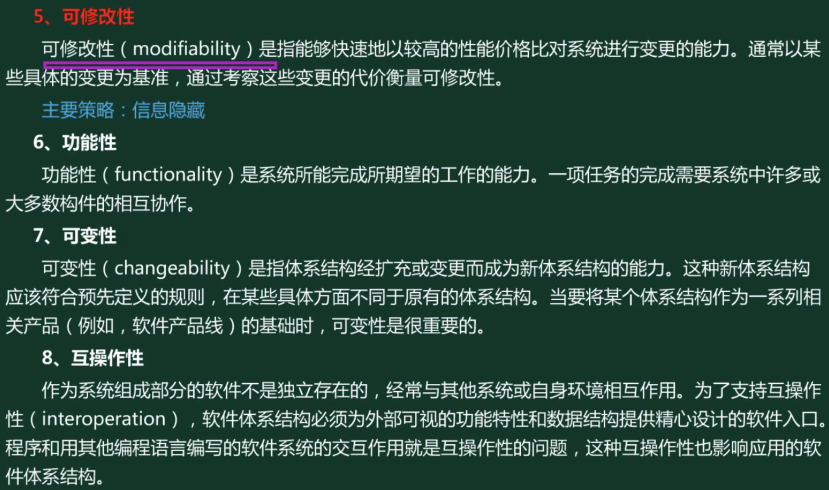
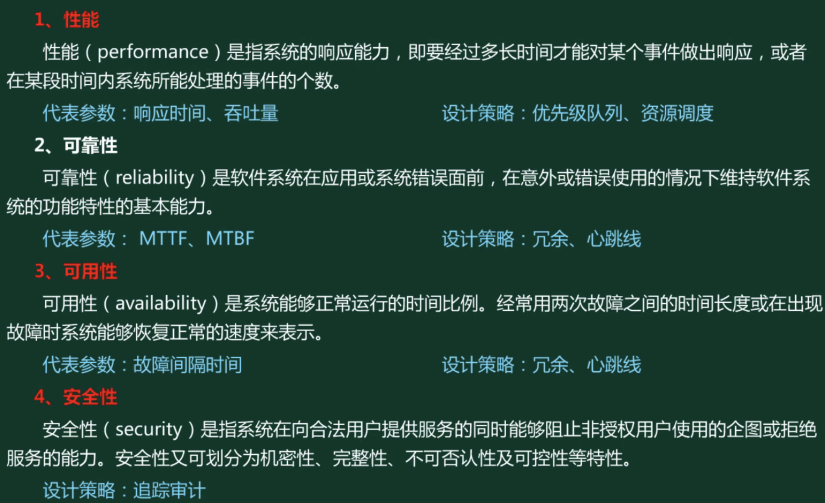
**依赖和关联的区别：**依赖是一个类中的某个方法使用了另一个类做为参数，而关联是一个类把另一个类做成成员变量。

## 8 用例图

用例之间的关系：包含、扩展、泛化

# 三 软件质量属性

## 3.1 常见的几个质量属性定义



## 3.2 风险点、敏感点、权衡点

**系统架构风险：**架构设计中潜在的、存在问题的架构决策所带来的隐患。

**敏感点：**为了实现某种特定的质量属性，一个或多个系统组件所具有的特性。

**权衡点：**影响多个质量属性，并对多个质量属性来说都是敏感点的系统属性。

## 3.3 常见场景对应的质量属性

* 用户的信用卡支付必须保证 99.999%的安全性，对应安全性属性。
* 用户信息数据库授权必须保证 99.999%可用，对应安全性属性。
* 系统拟采用新的加密算法，这会提高系统安全性，但同时会降低系统的性能，系统权衡点。
* 对交易请求处理时间的要求将影响系统数据传输协议和交易处理过程的设计，对应系统敏感点。
* 现有架构设计中的支付部分与第三方支付平台紧耦合，当系统需要支持新的支付平台时，这种设计会导致支付部分代码的修改，影响系统的可修改性，对应系统风险。
* 系统需要为后端工程师提供远程调试接口，并支持远程调试，对应可测试性属性。

# 四 安全



## 4.1 安全威胁

信息系统面临的安全威胁多种多样，来自多个方面。请指出信息系统面临哪些方面的安全威胁并分别予以简要描述。

信息系统面临的安全威胁来自于管理、应用系统、通信链路、网络系统、物理环境、操作系统等多个方面。口决：管应通，网物操

* 物理安全威胁是指对系统所用设备的威胁，如自然灾害、电源故障、数据库故障和设备被盗等造成数据丢失或信息泄漏。
* 通信链路安全威胁是指在传输线路上安装窃听装置或对通信链路进行干扰。
* 网络安全威胁当前主要是指由于因特网的开放性、国际性与无安全管理性，对内部网络形成的严重安全威胁。
* 操作系统安全威胁指的是操作系统本身的后门或安全缺陷，如“木马”和“陷阱 门”等。
* 应用系统安全威胁是指对于网络服务或用户业务系统安全的威胁，包括应用系统自身漏洞，也受到“木马”的威胁。
* 管理系统安全威胁指的是人员管理和各种安全管理制度。

# 五 ESB

## 5.1 定义

企业服务总线（Enterprise Service Bus,ESB)是传统中间件技术与XML、Web服务等技术结合的产物，主要支持异构系统集成。ESB基于内容的路由和过滤，具备复杂数据的传输能力，并可以提供一系列的标准接口。

## 5.2 ESB的主要功能

1. 服务位置透明性；
2. 传输协议转换；
3. 消息格式转换；
4. 消息路由；
5. 消息增强；
6. 安全性；
7. 监控与管理。

# 六 可靠性

****

## 6.1 定义

**可靠性：**是系统在规定的时间内及规定的环境条件下，完成规定功能的能力，也就是系统无故障运行的概率。

**可靠度：**系统在规定的条件下、规定的时间内不发生失效的概率。

**失效率：**又称风险函数，也可以称为条件失效强度，是指运行至此刻系统未出现失效的情况下，单位时间系统出现失效的概率。

## 6.2 检错技术

请给出检错技术的优缺点，并说明检测技术常见的实现方式和处理方式。

**优点**：检错技术实现的代价一般低于容错技术和冗余技术。

**缺点**：就是不能自动解决故障，出现故障后如果不进行人工干预，将最终导致软件系统不能正常运行。

**实现方式：**

1. **判断返回结果**，如果返回结果超出正常范围，则进行异常处理；
2. **计算运行时间**，如果某个模块或函数运行时间超过预期时间，可以判断出现故障；
3. **置状态标志位**。

**处理方式：**大多数都采用“査出故障-停止软件运行-报警”的处理方式。 但根据故障的不同情况，也有采用不停止或部分停止软件系统运行的情况，这一般由故障是否需要实时处理来决定。

## 6.3 可靠性的子特性

**口决：成容（成龙）易依**

* 成熟性：成熟性是指系统避免因错误的发生而导致失效的能力；
* 容错性：容错性是指在系统发生故障或违反指定接口的情况下，系统维持规定的性能级别的能力；
* 易恢复性：易恢复性是指系统发生失效的情况下，重建规定的性能级别并恢复受直接影响的数据的能力；
* 依从性：可靠性的依从性是指系统依附于与可靠性相关的标准、约定或规定的能力。

## 6.4 可靠性模型

* 时间模型
* 数据模型
* 故障植入模型

## 6.5 可靠性设计原则

* **软件可靠性设计是软件设计的一部分，必须在软件的总体设计框架中使用，并且不能与其他设计原则冲突。**
* **软件可靠性设计在满足提高软件质量的前提下，以提高和保障软件可靠性为最终目标。（软件质量是前提，可靠性为最终目标）**
* **软件可靠性设计应确定软件可靠性目标，不能无限扩大化，并且排在功能度、用户需求和开发费用之后考虑。**

## 6.6 可靠性设计技术

1. **容错设计（4后面的是我总结的）**

* **N版本程序设计**
* **恢复块设计**
* **冗余设计**

**（1）时间冗余**

**执行出错时，循环执行几次，以避免因网络繁忙或抖动或资料被占用引起的失 败**

**（2）结构冗余（硬件冗余、软件冗余）**

**（3）信息冗余（校验码）**

1. **检错设计**
2. **降低复杂度设计**
3. **集群技术**
4. **避错设计/防卫式程序设计**
5. **双机容错**

* **双机热备：主服务器正常时，备份服务器空闲**
* **双机互备：同时提供不同的服务，心不跳则接管**
* **双机双工：同时提供相同的服务，集群的一种**

## 6.7 影响可靠性的因素

口决：剖规内开可

1. **运行剖面（环境）**
2. **软件规模**
3. **软件内部结构**
4. **软件的开发方法和开发环境**
5. **软件的可靠性投入**

## 6.8 可靠性分析技术

1. **故障树分析方法**

一种自项向下的软件可靠性分析方法，即从软件系统不希望发生的事件（顶事件），特别是对人员和设备的安全及可靠性产生重大影响的事件开始向下逐步追查导致事件发生的原因，直至基本事件（底事件）。从而确定软件故障原因的各种组合方式和发生概率。基本的步骤是软件故障树的建立、定性分析和定量分析。

1. **失效模式与效应分析方法**

在软件开发阶段的早期，通过识别软件失效模式分析造成的后果。研究分析各种失效模式生产的原因，寻找消除和减少其有害后果的方法。以便尽早发现潜在的问题，并采取相应措施，从而提高软件的可靠性和安全性。

# 七 数据库

## 7.1 反规范化技术

**定义：**

规范化设计后，数据库设计者希望牺牲部分规范化来提髙性能，这种从规范化设计的回退方法叫做反规范化技术。

**优点：**

反规范化设计允许保留或者新增一些冗余数据，从而减少数据查询中表连接的数目或简化计算过程，提高数据访问效率。

采用反规范化技术的益处：能够减少数据库查询时SQL连接的数目，从而减少磁盘I/O数据量，提高查询效率。

**缺点：**

数据的重复存储，浪费了磁盘空间；为了保障数据的一致性，增加了数据维护的复杂性。

**常见的反规范化技术包括：**

1. 增加冗余列：在多个表中保留相同的列，通过增加数据冗余减少或避免查询时的连接操作；
2. 增加派生列：在表中增加可以由本表或其他表中数据计算生成的列，减少查询时的连接操作并避免计算或使用集合函数；
3. 表水平分割：根据一列或多列数据的值，把数据放到多个独立的表中，主要用于表数据规模很大、表中数据相对独立或数据需要存放到多个介质上时使用；
4. 表垂直分割：对表进行分割，将主键与部分列放到一个表中，主键与其他列放到另一个表中，在查询时减少I/O次数。

## 7.2 NOSQL

**优点：**

* 支持高并发数据访问，性能较高。
* 存储结构松散，能够灵活支持多种类型的数据格式。
* 能够支持海量数据的存储，且易于横向扩展。
* 基于分布式数据存储，不存在单点故障和性能瓶颈，系统可用性高。

缺点：

* NoSQL数据库的现有产品不够成熟，大多数产品处于初创期。
* 并未形成一定的标准，产品种类繁多，缺乏官方支持。
* 不提供对SQL的支持，学习和应用迁移成本较高。
* 支持的特性不够丰富，现有产品提供的功能比较有限

## 7.3 主从复制

引入主从复制机制所带来的好处：

* 避免数据库单点故障，提升可用性

主服务器实时、异步复制数据到从服务器，当主数据库宕机时，可在从数据库中选择一个升级为主服务器，从而防止数据库单点故障。

* 读写分离，提高査询效率

根据系统数据库访问特点，可以使用主数据库进行数据的插入、删除及更新等写操作，而从数据库则专门用来进行数据査询操作，从而将査询操作分担到不同的从服务器以提高数据库访问效率。

* 可扩展性更优  
  如果采用单台数据库服务器，则访问量持续增加时，数据库瓶颈暴露，且无法迅速解决问题。而主从结构可以快速增加从服务器数量，以满足需求
* 负载均衡  
  一主多从分担任务，相当于负载均衡
* 提升数据安全性  
  系统中的数据冗余存放多份，不会因为某台机器硬件故障而导致数据丢失

# 八 WEB架构设计

## 8.1 REST

**定义：**REST 从资源的角度来定义整个网络系统结构，分布在各处的资源由统一资源标识符（ URI)确定，客户端应用程序通过 URI 获取资源的表现，并通过获得资源表现使得其状态发生改变。

REST 中将资源、资源的表现和获取资源的动作三者进行分离

## 8.2 MVC

**使用MVC设计表现层，具有以下优点：**

* 允许多种用户界面的扩展。在MVC模式中，视图与模型没有必然的联系，都是通过控制器发生联系，如果增加新类型的用户界面，只需修改响应的控制器和视图即可，模型无需变动；
* 易于维护。控制器和视图随着模型的扩展而扩展，只要保持公共接口，控制器和视图的旧版本可以继续使用；
* 支持功能强大的用户界面。用户界面与模型方法调用组合起来，使程序的使用更清晰，可将友好的界面发布给用户。

## 8.3 负载均衡

负载均衡机制是大型Web应用解决高负荷访问和大量并发请求时常用的有效解决方法，典型的负载均衡机制包括基于DNS的负载均衡、基于反向代理的负载均衡等。

基于DNS的负载均衡机制通过DNS服务器实现，通常通过循环复用具有同一域名的多个主机地址的服务器实现负载均衡，可以看出，该机制具有实现简单、容易实施及低成本的特性。反向代理负载均衡则是将来自Internet的连接请求以反向代理的方式动态转发给内部网络上的多台服务器进行处理，从而达到负载均衡的目的。

从系统执行效率方面讲，基于DNS的负载均衡机制实现简单，但其通常不能区分服务器的差异，也不能反映服务器的当前运行状态。基于反向代理的则可以根据内部服务器的性能差异及实时负载情况进行动态负载均衡，当系统多个Web服务器性能存在明显差异或内部Web服务器出现故障时，负载均衡器可以更快做出响应，从而保证客户端的访问效率。采用基于反向代理的负载均衡机制，可在代理服务器中引入调速缓存机制，对Web服务器返回的静态页面或图片等静态资源进行缓存，由代理服务器承担对原始服务器的静态资源访问请求，从而进一步降低原始Web服务器的负载。

从安全性方面讲，采用基于反向代理的负载均衡机制，代理服务器屏蔽了客户端对真实Web服务器的直接访问，恶意用户无法对真实Web服务器进行攻击，且可以通过代理服务器为原本不安全的客户端与Web服务器之间的连接建立安全通道。因此采用基于反向代理的负载均衡机制可为系统提供更好的安全性保障。

# 九 构件

**基于构件的软件开发中，可以通过不同的途径来获取构件，主要包括以下4种方法：**

* 从现有构件中获得符合要求的构件，直接使用或做适应性修改，得到可复用的构件；
* 通过遗留工程（Legacy Engineering)，将具有潜在复用价值的软件提取出来，得到可复用的构件；
* 从市场上购买现成的商业构件，BPCOTS(Commercial Off-The-Shell)构件；
* 开发新的符合要求的构件。

**开发构件通常采取3种策略：**

* 分区（partitioning)：指的是将问题情景的空间分割成几乎可以独立研究的部分；
* 抽象（abstraction)：是对在给定实践内执行指定计算的软/硬件申.元的一种抽象；
* 分割（segmentation)：是将结构引入构件的行为，支持对行为性质进行时序推理。

**当前主流构件标准有：**

* CORBA：由OMG(对象管理集团）制定；
* COM/DCOM：由Microsoft制定；
* EJB：由SUN的Java企业Bean制定。

# 十 设计模式

**创建型模式主要用于创建对象，为设计类实例化新对象提供指南。**

1. **单例（Singleton）模式**：某个类只能生成一个实例，该类提供了一个全局访问点供外部获取该实例，其拓展是有限多例模式。
2. **原型（Prototype）模式**：将一个对象作为原型，通过对其进行复制而克隆出多个和原型类似的新实例。
3. **工厂方法（FactoryMethod）模式**：定义一个用于创建产品的接口，由子类决定生产什么产品。
4. **抽象工厂（AbstractFactory）模式**：提供一个创建产品族的接口，其每个子类可以生产一系列相关的产品。
5. **建造者（Builder）模式**：将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建它们，最后构建成该复杂对象。

**结构型模式主要用于处理类或对象的组合，对类如何设计以形成更大的结构提供指南。**

1. **代理（Proxy）模式**：为某对象提供一种代理以控制对该对象的访问。即客户端通过代理间接地访问该对象，从而限制、增强或修改该对象的一些特性。
2. **适配器（Adapter）模式**：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。
3. **桥接（Bridge）模式**：将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现的，从而降低了抽象和实现这两个可变维度的耦合度。
4. **装饰（Decorator）模式**：动态地给对象增加一些职责，即增加其额外的功能。
5. **外观（Facade）模式**：为多个复杂的子系统提供一个一致的接口，使这些子系统更加容易被访问。
6. **享元（Flyweight）模式**：运用共享技术来有效地支持大量细粒度对象的复用。
7. **组合（Composite）模式**：将对象组合成树状层次结构，使用户对单个对象和组合对象具有一致的访问性。

**行为型模式主要用于描述类或对象的交互以及职责的分配，对类之间交互以及分配责任的方式提供指南。**

1. **模板方法（Template Method）模式**：定义一个操作中的算法骨架，将算法的一些步骤延迟到子类中，使得子类在可以不改变该算法结构的情况下重定义该算法的某些特定步骤。
2. **策略（Strategy）模式**：定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的改变不会影响使用算法的客户。
3. **命令（Command）模式**：将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。
4. **职责链（Chain of Responsibility）模式**：把请求从链中的一个对象传到下一个对象，直到请求被响应为止。通过这种方式去除对象之间的耦合。
5. **状态（State）模式**：允许一个对象在其内部状态发生改变时改变其行为能力。
6. **观察者（Observer）模式**：多个对象间存在一对多关系，当一个对象发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为。
7. **中介者（Mediator）模式**：定义一个中介对象来简化原有对象之间的交互关系，降低系统中对象间的耦合度，使原有对象之间不必相互了解。
8. **迭代器（Iterator）模式**：提供一种方法来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。
9. **访问者（Visitor）模式**：在不改变集合元素的前提下，为一个集合中的每个元素提供多种访问方式，即每个元素有多个访问者对象访问。
10. **备忘录（Memento）模式**：在不破坏封装性的前提下，获取并保存一个对象的内部状态，以便以后恢复它。