架构:计算机系统的总体设计或结构,包括其硬件和支持系统运行的软件,尤其是微处理器内部的结构
构件：一组基本的构成要素, **连接件**：这些要素之间的连接关系, **物理分布**：这些要素连接之后形成的拓扑结构, **约束**：要素或连接关系上的限制条件 **性能**：质量
架构公式 = 构件 + 连接件 + 拓扑结构 + 约束 + 质量
软件架构(SA)：提供了一个结构、行为和属性的高级抽象；从一个较高的层次来考虑组成系统的构件、构件之间的连接,以及由构件与构件交互形成的拓扑结构；这些要素应该满足一定的限制,遵循一定的设计规则,能够在一定的环境下进行演化；反映系统开发中具有重要影响的设计决策,便于各种人员的交流,反映多种关注,据此开发的系统能完成系统既定的功能和性能需求
架构规模:对于大规模的复杂软件系统来说,对系统全局结构的设计比起对算法的 选择和数据结构的设计明显重要得多
目标: 建立一个一致的系统及其视图集,并表达为最终用户和软件设计者需 要的结构形式,支持用户和设计者之间的交流与理解; 分为两方面: **外向**目标：建立满足最终用户要求的系统需求; **内向**目标：建立满足系统设计者需要以及易于系统实现、维护和扩展的系统构件构成。
作用：1.交流的手段: 在软件设计者、最终用户之间方便的交流; 2. 可传递的、可复用的模型(可重复利用的、可转移的系统抽象): 用到其它的项目、提高大规模重复利用率；3.关键决策的体现:折衷,关于性能与安全性、可维护性与可靠性、当前开发费用和未来开发代价 **意义**：SA是软件开发过程初期的产品,在开发的早期阶段就考虑系统的正确设计与方案选择,为以后开发、测试、维护各个阶段提供了保证
架构师关注:软件质量, 各种因素....
物理~：系统如何被安装,部署,配置在分布式的环境下,部署图

中间件是一组程序,应用于分布式系统各应用之中,屏蔽底层通讯和提供公共服务,保障系统的高可靠性、高可用性、高灵活性
分布式应用借助中间件在不同技术之间共享资源 ● 位于客户机和服务器的操作系统之上,管理计算机资源和网络通讯 ● 连接两个独立应用程序或独立系统的软件,即使它们具有不同的接口 ● 通过中间件,应用程序可以工作于多平台或OS 环境 ● **Mail System**不是中间件
作用：**1.**屏蔽异构性：异构性表现在计算机的软硬件差异,包括硬件(CPU和指令集等,操作系统、数据库(不同存储和访问格式等
2.实现互操作：因为异构性,产生的结果是软件依赖于计算环境,使得各种不同软件之间在不同平台之间不能移植,或者移植困难。而且,因为网络协议和通信机制不同,这些系统不能有效相互集成
3.共性凝练和复用：软件应用领域越来越多,相同领域的应用系统之间许多基础功能和结构是有相似性的。通过中间件提供简单、一致、集成的开发和运行环境,简化分布式系统的设计、编程和管理
意义：缩短开发周期、节约应用程序开发成本、降低运行成本、降低故障率、改善决策、应用系统群集/集成、减少软件维护、提高质量、改进技术、提高产品吸引力
分类: 1.应用服务类中间件: 为应用系统提供一个综合的计算环境和支撑平台,包括对象请求代理~、事务监控交易~、Java应用服务器~等
2.应用集成类中间件: 提供各种不同网络应用系统之间的消息通信、服务集成和数据集成的功能, 如消息中间件、企业集成EAI、企业服务总线以及相配套的适配器。
3.业务架构类中间件: 除了可以将底层共性技术的特征抽象到中间件,还可以将业务共性抽象至中间件,形成应用模式,如业务流程等
4+1视图模型 用例视图：描述系统的典型场景与功能
逻辑~：系统的抽象概念与功能(类、接口等)类图,协作图,时序图等；
开发~：系统中的子系统,模块,文件,资源的关系,组件图、包图等；
进程~：系统的进程及其之间的通信协作关系,活动图,时序图等

软件架构风格

定义：描述用于以组织一类软件系统的惯用模式,反映了领域中众多系统所共有的结构和语义特性,并指导如何将各个模块和子系统有效地组织成一个完整的系统。定义一些构件和连接件类型,施加一组约束描述组合方式
分类 数据流风格：批处理；管道/过滤器；过程控制；
调用/返回风格：主程序/子程序；面向对象；分层结构
独立构件风格：事件系统；
虚拟机风格：解释器；基于规则的系统；
以数据为中心的风格：仓库；黑板；
其他架构风格：MVC；P2P；Grid；SOA

面向对象风格

系统 被看作对象的集合, 每个对象都有一个它自己的操作集合。数据及作用在数据上的操作被封装成抽象数据类型(对象)
可以做到信息隐藏,内部的设计决策被封装
构件：类和对象 **连接件**：对象之间通过函数与过程调用实现交互
特性: 封装：限制对某些信息的访问 ● **交互**：通过过程调用或类似的协议 ● **多态**：在运行时选择具体的操作 ● **继承**：对共享的功能保持唯一的接口 ● **动态绑定**：运行时决定实际调用的操作
复用和维护
优点：复用和维护；反映现实世界；容易分解一个系统
缺点：管理大量的对象；必须知道对象的身份标识；继承引起复杂度,关键系统中慎用
应用在KWIC: 数据不再被构件直接共享,而是被封装在了Object中, 每个对象提供了一个接口,允许其他对象通 过该接口调用对该对象内封装的数据的操作
优点: 修改不影响其他构件, 依赖性降低, 复用性提高
缺点:不适合功能的扩展(要么修改已有的, 要么新增模块)

KWIC案例

接收按行排列的数据, 重复地把第一个词删除,然后接到行末,把所有行的各种移位结果按照字母表顺序输出

主程序-子过程风格

构件：主程序,子程序 **连接器**：调用-返回机制 **拓扑结构**：层次化结构
非结构化程序：所有的程序代码均包含在一个主程序文件中
缺陷：逻辑不清；无法复用；难以与其他代码合并；难于修改;难以测试特定部分的代码
结构化程序：逐层分解 ● 基于"定义-使用"关系 ● 用过程调用作为交互机制 ● 主程序的正确性依赖于它所调用的子程序的正确性
本质: 将大系统分解为若干模块(模块化),主程序调用这些模块实现完整的系统功能。
优点：已被证明是成功的设计方法,可以被用于较大程序
缺点：代码太多,表现不好；程序太大,开发太慢,测试越来越困难
分解成模块的**四大原则**：
模块**独立性**：高聚合、低耦合 模块**规模适中性**：过大分解不充分难理解; 过小开销大接口复杂, 模块**复用性**：高扇入+低扇出
作用域与控制域适当性：作用域要包含在控制域之中
- C的控制域包括C自己和C下属(调用)的模块
- 作用域：一个模块里有判断语句,这个语句会影响多个模块,多个模块依赖于这个判定条件,那么影响的范围就是作用域
- C控制域是CEF,作用域CEFD,作用域>控制域,一旦C做了修改,你就要去找D的位置,因为你的修改可能影响D,一旦修改了你还要去找D在哪(因为它不受你控制),会带来负面的影响,这是一种耦合性,会对维护带来很大的麻烦
应用在KWIC: **1.**分为四个基本功能：输入、移位、排序、输出 **2.**主程序按次序调用这四个模块 **3.**通过共享的数据存储、使用无约束的读-写协议在模块之间进行 数据交换)

数据流风格

处理操作：数据到达即被激活,无数据时不工作
特征：数据的可用性决定着处理**计算单元**是否执行；系统结构：数据在各处理之间的有序移动；在纯数据流系统中,处理操作之间除了数据交换,没有任何其他的交互
构件：数据处理, **构件接口**：输入端口和输出端口,从输入端口读取数据,向输出端口写入数据, **计算模型**：从输入端口读数,经过计算/处理,然后写到输出端口, **连接件**：数据流 (单向、通常是异步、有缓冲 ● 接口角色：reader和writer ● 计算模型: 把数据从一个处理的输出端口传送到另一个处理的输入端口)
拓扑结构：任意拓扑结构的图

管道-过滤器风格

适用场景：数据源源不断的产生,系统需要对这些数据进行若干处理。**解决方案**：把系统分解为几个顺序的处理步骤,这些步骤之间通过数据流连接,一个步骤的输出是另一个步骤的输入；每个处理步骤由一个过滤器实现；处理步骤之间的数据传输由管道负责。每个处理步骤(过滤器)都有一组输入和输出,过滤器从管道中读取 输入的数据流,经过内部处理,然后产生输出数据流并写入管道中 **构件**：过滤器, 处理数据流 **连接件**：管道,连接一个源和一个目的过滤器。连接器定义了数据流的图, 形成拓扑结构
过滤器 目标：将源数据递增的变换成目标数据
数据流变换的方式：增加**丰富**数据；通过浓缩和删减**精炼**数据；改变数据表现方式**转化**数据；**分解**为多个流；数据流**合并**为一个
读取与处理数据流方式: 递增的读取和消费数据: 数据到来时便处理,不是收集完然后处理,即在输入被完全消费之前输出就产生了
其他特征：无上下文信息；不保留状态；对上下游的其他过滤器无任何了解

过滤器状态：**1.**停止状态：处于待启动状态,外部启动过滤器后,过滤器处于处理状态。**2.**处理状态：正处理输入数据队列中的数据。**3.**等待状态：输入数据队列为空,此时过滤器等待,当有新的数据输入时,过滤器处于处理状态。
管道 作用：在过滤器之间传送数据 ● 是单向流 ● 可能具有缓冲区 ● 数据缓冲区可以是文件、数组、字典、树等集合类型 ● 管道形成数据传输图 ● 管道的先后顺序不影响输出的结果 ● 不同的管道中流动的数据流,可能具有不同的数据格式 ● **实现机制** 限制管道的大小 ● 管道写满时**write**函数会阻塞 ● 管道为空时**read**函数被阻塞,这解决了read()调用返回文件结束的问题。从管道读数据是一次性操作,数据一旦被读,它就从管道中被抛弃,释放空间以便写更多的数据。
优点：允许对一些如吞吐量、死锁等属性的分析 ● 支持并行执行；使得系统中的构件具有良好的隐蔽性和高内聚、低耦合的特点 ● 支持软件复用：系统的行为是多个过滤器的行为的简单合成 ● 在两个过滤器之间提供适合数据,任何两个过滤器都可被连接起来 ● 系统维护和增强系统性能简单：新的过滤器可以添加到现有系统中,旧的可以被改进的换掉；
缺点：通常导致进程成为批处理的结构：每个过滤器是一个完整的从输入到输出的转换 ● 不适合处理交互的应用：当需要增量地显示改变时,这个问题尤为严重 ● 因为在数据传输上没有通用的标准,每个过滤器都增加了解析和合成数据的工作,这样就导致了系统性能下降,并增加了编写过滤器的复杂性。
应用在KWIC:
四个过滤器：输入、移位、排序、输出
每个过滤器处理数据,然后将结果送至下一个过滤器
控制机制是分布式的：只要有数 据传如,过滤器即开始工作
过滤器之间的数据共享被严格限制在管道传输

事件系统风格

显式调用：各个构件之间的互动是由显性调用函数或过程完成的。调用的过程与次序是固定的、预先设定的。
隐式调用：不直接去invoke一个过程
Event Source：发布事件到EventtManager
Event Handlers：可以注册自己感兴趣的事件, 并将自己的某个过程与相应的事件进行关联
Event Manager：当一个事件被发布,系统自动调用在该事件中注册的所有过程(负责调用所有注册到该事件的EventHandler)
特点：事件的触发者并不知道哪些构件会被这些事件影响,相互保持独立 ● 不能假定构件的处理顺序,甚至不知道哪些过程会被调用 ● 各个构件之间彼此之间无连接关系,各自独立存在,通过对事件的发布和注册实现关联 ● 可以异步执行 ● 一对多通信
构件：对象或过程,并提供如下两种**接口**: **1.**过程或函数,充当事件源或事件处理器的角色；**2.**事件
连接机制：**1.**事件-过程绑定: 事件处理器的过程向特定事件注册 ● 事件源构件发布事件 ● 当某些事件被发布时,向其注册的过程被隐式调用 ● 调用的次序是不确定的 ● **2.**事件-事件的绑定, 一个事件也可能触发其他事件,形成事件链
调试器的例子: 编辑器与变量监视器向调试器注册,接收"断点事件" ● 一旦遇到断点,调试器发布事件,从而触发"编辑器"与"变量监测器" ● "编辑器"将源代码滚动到断点处,"变量监测器"则更新当前变量值并显示出来
事件调度策略 无独立(非集中式)调度模块的事件管理器: **观察者模式** 每个模块都能向其他所有模块注册事件
带有独立调度模块的事件管理器, 下面都是带有独立(集中式)调度
事件调度模块: 负责接收到来的事件并分发它们到其它模块。调度器要决定怎样分发事件, 两种策略:

全广播式：调度模块将事件广播到所有的模块,但只有感兴趣的模块才去取事件并触发自身的行为；无目的广播,靠接受者自行决定是否加以处理或者简单抛弃。
选择广播式：调度模块将事件送到那些已经注册了的模块中。
- **点对点**模式：基于消息队列。消息只能够被唯 一的消费者所消费,消费之后即从队列中删除
- **发布-订阅**模式：一个事件可以被多个订阅者消费；事件在发送给订阅者之后,并不会马上从topic中删除,topic会在事件过期之后自动将其删除。
应用于KWIC: 四个功能模块 ●共享数据并不直接暴露其格式,而是进行封装 ● 模块的调用发生在数据发生改变时, 不是主程序控制
优点：**1.**支持实现交互式系统；**2.**异步执行 **3.**容易复用**4.** 构件替换不会影响到其它构件的接口；**5.** 并发处理提高系统性能；**6.**健壮性：一个构件出错将不会影响其他构件。
缺点：**1.**分布式的控制方式不利于系统的同步、验证和调试。**2.** 独立调度模块的数据需要经过调度模块的传递,全局性能和资源管理成为了系统的瓶颈。

分层结构

严格分层：上层-直接下层 **松散**分层：上层-所有下层 **横切**关注：在严格分层的基础上,每一层都可以与另外的一个组件交互
分层原则：分离关注(减少功能重叠)；抽象(删枝节留主干)；隐藏(只暴露需要访问的接口) **构件**：各层次内部包含的构件
连接件：层间的交互协议 **拓扑结构**：分层
拓扑约束：对相邻层间交互的约束(集中式/分布式部署)
某一层中的构件一般只与同一级别中的对等实体或较低级别中的构件交互,这种单向交互可以减少不同级别中的构件之间的依赖性。

三层C/S结构: 客户端-中间层-服务端,(表现层-逻辑层-数据层) **中间层**:事务处理监 控服务器、消息服务器、应用服务器, 负责消息排队, 业务逻辑执行, 数据中转等功能
B/S架构: (表现层：浏览器 ● 逻辑层 Web服务器 应用服务器 ● 数据层：数据库服务器)
优点: 系统维护成本低 ● 客户端无任何业务逻辑 ●良好的灵活性和可扩展性 ● 较好的安全性 ● 稳定性,延展性和执行效率 ● 容错能力和负载平衡能力
缺点: 每请求一次服务器就要刷新一次页面 ● 受HTTP协议在数据查询等响应速度上,要远低于C/S架构 ● 数据提交一般以页面为单位,数据的动态交互性不强,不利于在线事务处理(OLTP)应用 ● HTML的表达能力难以支持复杂GUI (如报表等)。

MVC风格

关注点分离：模型、视图和控制器, 从开发者的角度看,实现model与view的解耦 **构件**：解释器引擎、存储区(被解释的源代码、解释器引擎当前的控制状态的表示、程序当前执行状态的表示)
连接器：对存储区的数据访问
三大构件：**Model**：负责数据存取、负责业务逻辑实现、可能负责数据验证; **View**：负责获取用户输入,向controller发送处理请求,接收来自Controller的反馈并将model的处理结果显示给用户。一个model可能有多个View; **Controller**：负责接收来自客户的请求、调用model执行业务逻辑、调用View显示执行结果
连接件：隐式调用、显式调用、或者其他机制Http
两个分离原则：展示与模型分离；控制器与视图分离
优点：代码易开发易维护；同一信息可以有不同的显示方式；业务逻辑更易测试

解释器风格

解释器是一个用来执行其他程序的程序。针对不同的硬件平台实现了一个虚拟机,将高抽象层次的程序翻译为低抽象层次所能理解的指令,以消除在程序语言与硬件之间存在的语义差异。在程序语言定义的计算和有效硬件操作确定的计算之间建立对应和联系
解释器和**编译器**：编译器不执行源程序代码,而是将其翻译为另一种语言(可执行的机器码或目标码),并输出到文件中以便随后链接为可执行文件, 在解释器中,程序源代码被解释器直接加以执行。
- 解释器的执行速度要慢于编译器产生的目标代码的执行速度,但是可能低于编译器"编译+链接+执行"的总时间
- 解释器通常省略了链接与编译的步骤,从而降低编程时间
- 解析器执行速度之所以慢,是因为每次解释执行的时候,都需要分析程序的结构,而编译代码则直接执行而无需重复编译
- 解释器对内存的分配是在解释时才进行的；而编译器则是在编译时进行,因此运行时直接将程序代码装入内存并执行即可
分类 传统解释器：纯粹的解释执行
基于字节码的：编译→解释执行。源代码首先被"编译"为高度压缩和优化的字节码,但并不是真正的机器目标代码,因而与硬件平台无关；编译后得到的字节码然后被解释器加以解释；
JIT编译器：编译 || 解释执行。只有当某个函数要被执行时才被编译, 不是编译全部的代码,而是只编译那些被频繁执行的代码段,如被执行多次的方法、包含多次循环的方法
1.编译得到字节码 **2.**字节码 被配置到目标系统中 **3.**当字节码被执行时,运行环境下的编译器将其翻译为本地机器码, **特点**是JIT模糊了解释器、字节码解释器和编译器之间 的边界与区分
执行Java: 1."编译"得到字节码(与硬件平台无关)；**2.**Java解释器直接执行该字节码

计算层的软件架构

计算模式变革: 单机->互联网 | 串行 | 多核,并核
性能度量: CPU速度(MIPS) ● 网络带宽(Mbps) ● 吞吐量(MIPS、TFLOPS、TPS、QPS)指系统在单位时间内处理请求的数量。● RT响应时间(s) : 指系统对请求作出响应的时间。● 网络延时(s) ● 并发用户数: 指系统可以同时承载的正常使用系统功能的用户的数量
TPS(Transactions Per Second) 一次事务(访问)包括了**三个**过程: **1.**用户请求服务器 **2.**服务器自己的内部处理 **3.**服务器返回给用户, TPS是每秒能够完成的**访问**(三个过程)的数量
QPS (Queries Per Second) 每秒能处理**请求**数目。对于一个页面的一次访问,形成一个TPS, 可能产生多次请求
垂直扩展: 向一个节点中增加资源,提高硬件配置
水平扩展: 增加新的节点(新的计算机分布式计算)
高扩展性: 系统在几乎任何时刻都可被正常访问, 通常量化为一年中正常运行的时间比 ●平均故障时间MTTF ●平均修复时间MTTR ●系统可用性=MTTF/(MTTF+ MTTR)
代码级优化: ● 发现瓶颈：压测工具、抽象模型 ● 线程、内存参数调整 ● Java特性优化 ● 减少并发冲突 ● 减少序列化 ● 减少字符到字节的转换● 使用长连接
通信模式优化
PPC 每次有新的连接就新建一个专门的进程,传统UNIX 网络
优点：简单,适合连接数少的情况, 如数据库服务器。
缺点：● fork 代价高：创建一个进程需要分配很多内核资源,内存映像从父进程复制到子进程。 ●父子进程通信复杂●支持的并发连接数量有限
prefork 系统在启动的时候就预先创建好进程,然后才开始接受

优点：●新的连接省去 fork 进程的操作,访问更快、体验更好
缺点：● 存在父子进程通信复杂、支持的并发连接数量有限的问题
TPC 每次有新的连接就新建一个**线程**去专门处理这个连接的请求
优点: ●线程更轻量级,创建线程的消耗比进程要少得多 ●多线程是共享进程内存空间,线程通信相比进程通信更简单。解决或者弱化了两个问题 ● PPC fork 代价高● 父子进程通信复杂
缺点：●创建线程仍然有代价,高并发时 (上万连接/s)有性能问题。●无须进程间通信,但是线程间的互斥和共享又引入了复杂度,可能 一不小心就导致了死锁问题。 ●多线程会互相影响,某个线程出现异常时,可能导致整个进程退出(例如内存越界) ● 存在 CPU 线程调度和切换代价的问题。连接不多(几百)的情况还是使用PPC
prethread 预先创建线程, 实现方式: ●主进程 accept,然后将连接交给某个线程处理。 ●子线程都尝试去 accept,最终只有一个线程accept 成功 ●比prefork支持更多的并发连接
Reactor: ●创建一个进程池,将连接分配给进程,一个进程可以处理多个连接的业务。来了一个事件我就有相应的反应" ● Reactor 会根据事件类型来调用相应的代码进行处理 **核心组成**两部分: ● Reactor ,负责监听和分配事件 ●处理资源池(进程池或线程池),负责处理事件●两者的数量都可以变化 **三种实现方案: 单Reactor单进程/线程**
● Reactor 对象通过 select 监控连接事件,收到事件后通过dispatch进行分发。● 如果是连接建立的事件,由 Acceptor 处理,通过 accept 接受连接,并创建一个Handler来处理连接后续的各种事件。
● 如果不是连接建立事件, 则Reactor 会调用连接对应的Handler (第 2 步中创建的 Handler)来进行响应● Handler 会完成 read→业务处理→send 的完整业务流程。 ●优点: 简单,没有进程间通信和竞争, 全部在同一个进程内● 缺点：一个进程无法发挥多核 CPU 的性能 ,只能部署多个系统来利用多核 CPU,这样需要维护多套系统●Handler 只

单Reactor多线程 ●主线程中,Reactor对象通过 select 监控连接事件,收到事件后通过 dispatch 进行分发 ●如果是连接建立的事件,则由 Acceptor 处理,Acceptor 通过 accept 接受连接,并创建一个 Handler 来处理连接后续的各种事件 ●如果不是连接建立事件,则 Reactor 会调用连接对应的 Handler来进行响应
●Handler 只负责响应事件,不进行业务处理● Handler 通过 read 读取到数据后,会发给 Processor 进行业务处理。
●Processor 会在独立的子线程中完成真正的业务处理,然后将响应结果发给主进程的Handler处理；Handler 收到响应后通过send将响应结果返回给client。**优点**：利用多核多CPU **缺点**：●多线程数据共享和访问比较复杂。例如,子线程传递给主线程涉及共享数据的互斥和保护机制。以Java 的NIO为例, Selector 是线程安全的,但是通过 Selector.selectKeys() 返回的键的集合是非线程安全的,对 selected keys 的处理必须单线程处理或者采取同步措施进行保护。●Reactor承担所有事件的监听和响应,只在主线程中运行,瞬间高并发时会成为性能瓶颈。
多Reactor多进程/线程 ●父进程中mainReactor对象通过select监控连接建立事件, 收到事件后通过Acceptor接收,将新的连接分配给某个子进程 ●子进程的 subReactor将mainReactor 分配的连接加入连接队列进行监听,并创建一个 Handler用于处理连接的各种事件。● 当有新的事件发生时,subReactor 会调用连接对应的 Handler进行响应 ●Handler 完成 read→业务处理→send 的完整业务流程。**优点:** ●职责明确: 父进程只接收新连接, 子进程完成后续的业务处理。 ●交互很简单,父进程只需要把新连接传给子进程,子进程无须返回数据。
●子进程之间是互相独立的,无须同步共享之类的处理(select、read、send 等无须同步共享, 业务处理可能需要同步共享)

Reactor是**非阻塞同步**网络模型,因为read和send 操作都需要用户进程同步操作。如果把 I/O 操作改为**异步**就能够提升性能--**Proactor**。
●来了事件我来处理,处理完了我通知你● Proactor Initiator 创建Proactor 和 Handler,并将Proactor和Handler都通过 Asynchronous Operation Processor注册到内核。●AOP 处理注册请求,并完成 I/O 操作。● AOP 完成 I/O 操作后通知 Proactor。● Proactor 根据不同事件类型回调不同的 Handler 进行业务处理。● Handler完成业务处理,Handler 也可以注册新的 Handler 到内核进程。 ●效率更高 ●异步I/O 能够充分利用 DMA 特性,让 I/O 操作与计算重叠, 但操作系统需要做大量的工作● Windows 通过IOCP实现了真正的异步● Linux 系统下的AIO 并不完善,以Reactor 模式为主
MapReduce: ●2个函数Map和Reduce ●6个过程 input 输入数据 split 对数据进行有依据的分组 map 把分组之后的数据拆分/映射为其他的新数据 shuffle 把新数据在本地分别有序储存起来 reduce 把shuffle后各处的数据放在一起进行组装 finalize 输出组装后的数据
设计思路: ●Job Tracker-框架中心,定时与集群中机器通信 , 管理哪些程序跑在哪些机器上,管理所有job ●TaskTracker 集群中每台机器都有,监视自己机器的资源和tasks运行状况 ● TaskTracker 把这些信息通过heartbeat 发送给 JobTracker ● JobTracker 会根据信息确定新的job分配运行在哪些机器上 **问题**● JobTracker 存在单点故障 ●JobTracker任务太多,过多的资源消耗 ●当 job非常多时,内存开销大(上限4000) ●以task 的数目表示资源,没有考虑到 cpu/ 内存的占用情况,内存很大很容易出现溢出 ●把资源强制划分为 map task slot 和 reduce task slot, 如果当系统中只有其一时,会造成资源的浪费
Yarn优点: 减小了JobTracker的资源消耗,并且让监测每一个Job子任务状态的程序分布式化了,更安全、更优美● 用户可以对不同的编程模型写自己的ApplicationMaster, 支持不同的编程模型 ●以内存为单位, 没

DNS 负载均衡：最简单,最常见,地理级别。**优点**：●简单、成本低：交给 DNS 服务器处理,无须自己开发、维护负载均衡设备 ●就近访问,提升访问速度：根据请求来源 IP,解析成距离用户最近的服务器地址,加快访问速度,改善性能
缺点: ●更新不及时：DNS 缓存的时间比较长,修改 DNS 配置后,有可能会继续访问修改前的 IP,访问失败,影响正常使用业务 ●扩展性差：DNS 负载均衡的控制权在域名商那里,无法根据业务特点针对其做更多的定制化功能和扩展特性。 ●分配策略比较简单：DNS 负载均衡支持的算法少；不能区分服务器的差异；也无法感知后端服务器的状态。
硬件负载均衡：通过单独的硬件设备来实现负载均衡功能,这类设备和路由器、交换机类似。如F5和 A10 **优点** ●功能强大：全面支持各层级的负载均衡,支持全面的负载均衡算法,支持全局负载均衡 ●性能强大：100 万以上的并发 ●稳定性高：商用硬件负载均衡, 严格测试,经过大规模使用 ●支持安全防护：具备防火墙、防 DDoS 攻击等安全功能 **缺点**●价格昂贵 ●扩展能力差
软件负载均衡: 优点●简单：无论是部署还是维护都比较简单 ●便宜: 只要买个 Linux 服务器,装上软件即可； ●灵活: 4 层和7层负载均衡可以根据业务进行选择；也可以根据业务进行比较方便的扩展**缺点**: ●性能一般 ●功能没有硬件负载均衡那么强大●一般不具备防火墙和防 DDoS 攻击等安全功能。
负载均衡典型架构 ●地理级别负载均衡：DNS 会根据用户地理位置决定返回哪个机房的 IP ●集群~：F5 收到请求后,进行集群级别的负载均衡 ●机器~：Nginx 收到用户请求后将请求发送给集群里面的某台服务器
负载均衡算法 ●**轮询**：按顺序轮流分配, 最简单的策略,无须关

●**加权轮询**：根据服务器权重进行任务分配,权重一般由根据硬件决定,因为不同服务器处理能力有差异,但没有考虑根据服务器的状态差异
●**负载最低优先:** (服务器的角度)● LVS以连接数来判断,●Nginx以HTTP 请求数判断●CPU 密集型--CPU 负载 ● I/O 密集型--I/O 负载 **特点** 可以感知服务器状态,但是复杂度要增加很多● CPU 负载最低优先的算法要求先获取每个服务器的CPU负载●通常效率不如轮询/加权轮询
●**性能最优类:** (客户端的角度) ●优先将任务分配给处理速度最快的服务器,相当于只是通过响应时间衡量服务器状态 ● 复杂度很高: ● 需要收集和分析每个服务器对每个任务的响应时间,在大量任务处理的场景下,这种收集和统计本身也会消耗较多的性能● 工业上使用采样,并调优采样率
●**Hash 类:** ●将相同Hash值的请求分到同一台服务器上--特定的业务需求 ●源地址Hash: 同一个源 IP 地址的任务分配给同一个服务器进行处理,适合于存在事务、会话的业务--网上银行同一个会话,总是访问同一台服务器 ● ID Hash: 将某个 ID 标识的业务分配到同一个服务器,ID 一般是临时性数据的 ID(如 session id)。网上银行登录用session id hash 可以实现同一个会话期间总是访问到同一台服务器的目的。
消息中间件 沟通模型分类
●寻址：直接、间接 ●阻塞：同步、异步 ●缓存：有、无
●消息内容：事件、命令、数据、流 ●确认：无确认、有确认、三次握手
●接收方数量：点对点、多播、任意播、地理多播、广播
●沟通方向：单向、全双工、半双工
●初始化方式：客户端初始化拉取、服务器初始化推送
特点：异步交互、客户端服务器松散耦合、可靠送达(数据持久存储)、利于应用集成、消息暂存在队列里、进程通过中间消息服务器来传递信息、对于数据库集成很自然 **消息模式**：一对一、一对多、多对一、多对多(发布/订阅模式 **队列**：发送者和接受者之间扮演一个中间地址目标,各个进程独立发送和失效,各个进程间的失效和交流失败可忽略——实现了时间解