**Software System Design-Architecture**

**Assignment-3**

成员：顾明政 王青雅 宗琳 陆梦葶

**第一部分：ADD方法的两次迭代**

**第一次迭代**

1. 功能性需求、非功能性需求及限制
2. 功能性需求：

C4系统应该能够维持和客户、服务端及可能出现的新的服务端之间的联系，应对客户的提出新的服务例如增加新的电话号码，以及应对客户对现有服务配置的改变例如电话号码变更或长途公司变更等。

C4系统应该能够维护和客户之间的交互，根据客户需求和客服安排发送各种请求给NOSS、Downstream systems等系统，等到响应并最终完成一次服务协商或账户管理或问题解决服务。系统能够快速响应客户通过Quick Service发出的请求并按照请求内容进行处理，也能投根据需要读取或更新database中相关的部分表。

C4系统能够保持和服务器端的联系，能够以直接请求-响应和建立连接-请求-响应两种方式向NOSS系统发送要求NOSS重新配置网络、向NOSS申请获取现有网络的状态或能力、要求NOSS“锁住”某些资源这三种请求。系统能够通过P/S component向Downstream Systems广播请求，同时对Downstream Systems关于影响用户配置的重要业务事件进行及时的响应和处理。

1. 非功能性需求的场景分析

（1）可靠性分析

**表1： 质量需求场景1：可靠性**

|  |  |
| --- | --- |
| **场景的组成部分** | **可能的值** |
| 刺激源 | 系统可以应对由外部或内部引发的错误或失败 |
| 刺激 | 客户输入信息格式错误、内容错误；系统和Billing System之间由于同时试图读写数据库而产生的冲突；客户和客服之间连接中断；不同来源的针对同一事务的更改需求之间产生的冲突。 |
| 制品 | C4系统 |
| 环境 | C4系统设计时、开发时、运行时、编译时、构件时、测试时 |
| 响应 | 出现冲突时几乎实时响应并高效修复故障，并且及时将冲突汇报给使用者；保持客户和客服间交流记录，中断时可以及时恢复。系统阻止一些危险用户或数据的访问； |
| 响应度量 | 出现冲突时修复的时间；修复冲突的成功率；日志记录的完整度 |

（2）性能分析

**表2： 质量需求场景2：性能**

|  |  |
| --- | --- |
| **场景的组成部分** | **可能的值** |
| 刺激源 | 系统内部或外部 |
| 刺激 | 大量客户（超过400个）同时并行使用系统 |
| 制品 | C4系统 |
| 环境 | C4系统不同操作模式：正常模式、高负荷模式、过载模式 |
| 响应 | 根据系统的使用人数切换不同的模式 |
| 响应度量 | 切换模式的时延、模式稳定程度 |

（3）可扩展性分析

**表3： 质量需求场景3：可扩展性**

|  |  |
| --- | --- |
| **场景的组成部分** | **可能的值** |
| 刺激源 | 客户、NOSS、下游系统 |
| 刺激 | 客户容量增长；新的物理设备servers的增加 |
| 制品 | C4系统的代码、数据、接口、物理组成 |
| 环境 | 系统运行时、编译时、初始化时、设计时 |
| 响应 | 保持成本和客户容量呈线性增长，能够适应较快的客户增长并最终扩展到一千五百万客户容量，能够监视并解决增长过程中的瓶颈；增加新的物理设备后能够系统能正常运行并保持性能 |
| 响应度量 | 可扩展用户容量、物理设备容量、扩展成本、扩展时间、扩展后设备的质量属性 |

（4）性能分析

**表4： 质量需求场景4：可用性**

|  |  |
| --- | --- |
| **场景的组成部分** | **可能的值** |
| 刺激源 | 最终端用户 |
| 刺激 | 客户想要高效的使用系统，能够很快的学习使用系统，减少使用时的错误情况；客户可以及时得知事务和任务是否完成、客户数据的完整性和最后被请求的配置选项的完整性 |
| 制品 | C4系统与用户交互的部分 |
| 环境 | C4系统 |
| 响应 | 系统可以及时响应用户需求，验证请求的服务内容的可用性；系统提供的服务符合用户的期待 |
| 响应度量 | 完成任务的时间；用户使用时发生错误的数量；用户完成任务的数量；用户对系统的满意程度；成功操作的比率；解决错误的时间 |

1. 约束

（1）系统前期的设备投资应该维持在一个较低的水平

（2）系统的高可用性不能通过容错硬件方式来实现

（3）系统工作站上没有持久的数据缓存来限制本地故障的影响

（4）当地工作站没有专门的管理员来维护停机时间

二、选择ASR

第一次迭代的元素选择是整个C4系统，对系统的功能性、非功能性及限制进行分析识别出ASR如下表。

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **构架驱动** | **重要性** | **难度** |
| 1 | 场景1  面对中断和冲突保持高可靠性 | 高 | 高 |
| 2 | 场景2  满足大量用户并行使用系统 | 高 | 高 |
| 3 | 场景3  可满足客户容量增长和物理设备增加，保持系统可扩展性 | 中 | 高 |
| 4 | 场景4  维护系统可用性 | 高 | 高 |
| 5 | 需求1  响应Quick Service发出的请求 | 高 | 中 |
| 6 | 需求2  能够向NOSS系统发送要求配置网络、向NOSS申请网络状态 “锁住”某些资源这三种请求 | 高 | 中 |
| 7 | 需求3  能够向Downstream Systems广播请求 | 高 | 中 |
| 8 | 需求4  对下游系统发出的重要业务事件进行及时的响应和处理 | 高 | 高 |
| 9 | 需求5  根据客户需求发送各种请求给NOSS、下游系统等系统，等待响应并完成服务 | 高 | 中 |
| 10 | 需求6  能够提供关于可供选择的产品或产品簇的建议 | 中 | 低 |
| 11 | 限制1  系统前期的设备投资应该维持在一个较低的水平 | 中 | 低 |
| 12 | 限制2  系统的高可用性不能通过容错硬件方式来实现 | 高 | 高 |
| 13 | 限制3  系统工作站上没有持久的数据缓存来限制本地故障的影响 | 高 | 中 |
| 14 | 限制4  当地工作站没有专门的管理员来维护停机时间 | 中 | 中 |

1. 选择满足ASR的设计构思
2. 识别设计理念

对于C4系统的设计理念主要有维持高可靠性、维护系统并行性能、实现功能扩展、维持高可用性。

维持高可用性包括对错误和冲突的检测、修复和预防。

维护系统并行性能包括控制资源需求和管理资源利用。

实现功能扩展包括增加内聚、减少耦合、推迟绑定时间、减少模块大小等。

维持高可用性包括帮助用户初始化及支持系统初始化。

|  |  |
| --- | --- |
| **设计理念** | **从属设计理念** |
| 可用性维护 | 错误检测 |
| 错误恢复 |
| 错误预防 |
| 并行性能 | 控制资源需求 |
| 管理资源利用 |
| 功能扩展 | 增加内聚减少耦合 |
| 减少模块大小 |
| 高可用性维持 | 帮助用户初始化 |
| 支持系统初始化 |

1. 为从属设计理念例举可选择的战略
2. 可选错误检测战略

参数识别：

系统需要能够检测与外部系统冲突和客户服务器之间连接的中断（场景1）

需要控制系统检测冲突和错误的速率及开销成本（场景1 限制1）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **#** | **模式名称** | **与外部系统冲突** | **连接中断** | **错误检测速率** | **检测成本** |
| 1 | 健康检测 | 否 | 是 | <0.2s | 高 |
| 2 | 环境检测 | 否 | 是 | <1s | 高 |
| 3 | 异常检测 | 是 | 是 | <1s | 中 |
| 4 | 自我检测 | 否 | 是 | <0.5s | 低 |
| 5 | 时间戳 | 否 | 否 | <0.2s | 中 |

理由：

异常检测可以实现对外部冲突和连接中断的检测，而其他的模式均无法同时实现系统可靠性的需求，故排除无法实现的四个模式。

异常检测的错误检测速率与检测成本可以满足需求。

决定：

选择异常检测模式

推论：

若客服与客户之间连接中断、和Billing System之间由于同时试图读写数据库产生冲突、可通过容错系统的错误报告发现。

1. 可选错误恢复战略

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **模式名称** | **错误恢复时间** | **资源占用** | **恢复率** |
| 1 | 冗余 | 小 | 占用较多空间 | >90% |
| 2 | 异常恢复 | 中 | 较少 | >90% |
| 3 | 软件升级 | 大 | 占用较多空间 | >90% |
| 4 | 重启 | 大 | 中断所有任务 | 不确定 |

理由：

由限制3没有持久缓存可排除冗余战略。重启战略恢复率无法保障，在可以容忍的范围内可以作为备选方案。软件升级成本较高且时间较长且需要投入大量人力成本，可行性较差。异常恢复机制可能需要前期投入成本，在可控范围内也可作为备选方案。

决定：

选择异常恢复模式或重启模式

推论：

系统检测到可能发生冲突时提醒客户尽快解决异常，或重启系统。

1. 可选错误预防战略

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **模式名称** | **成本** | **功能性影响** | **成功率** |
| 1 | 移除服务 | 低 | 大 | 高 |
| 2 | 预言性模型 | 高 | 小 | 中 |
| 3 | 简单预防 | 低 | 小 | 低 |

理由：

在错误检测和错误恢复效率得到保障的情况下，对可能发生的冲突有较高的容忍度。移除服务会直接影响系统功能性需求，显然不可行。而预言性模型成本过大超过可控范围。

决定：

选择简单预防模式

推论：

容错系统进行小规模的错误预防，以减小开发成本。

1. 可选控制资源需求

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **战略名称** | **可满足并行人数** | **对系统影响** |
| 1 | 资源优先级 | >400 | 小 |
| 2 | 控制时间 | >400 | 中 |
| 3 | 减少费用 | <400 | 大 |

理由：

减少费用对系统功能性需求影响较大且无法满足400人以上的并行需求（场景2），故排除此选项。资源优先级战略可较好平衡不同优先级客户的需求，以达到系统负载平衡，控制客户使用时间也可以一定程度上控制资源需求，实现并行效果。

决定：

选择资源优先级模式及控制时间模式

推论：

性能维持模块可能需要对客户的重要性程度划分优先级，根据不同的优先级提供不同的服务时长和服务功能范围。同时通过控制客户占用系统的时间达到高并行性。

1. 可选资源管理利用战略

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **模式名称** | **成本** | **可满足并行人数** | **效率** |
| 1 | 增加资源总数 | 空间时间成本高 | >400 | 高 |
| 2 | 并行运算 | 占用一定空间时间成本 | >400 | 中 |
| 3 | 数据备份 | 空间成本高 | >400 | 中 |
| 4 | 资源调度 | 占用一定空间时间成本 | >400 | 高 |

理由：

限制3可知系统无法提供较高的空间成本。增加资源总数无法满足限制2和限制3，故排除。数据备份无法实现限制3,没有持久的本地缓存提供备份空间。并行运算和资源调度均占用一定时间空间成本，在可容忍范围内可作为备选方案。

决定：

选择资源调度模式及并行运算模式

推论：

性能维持模块需要维持一个资源调度器以实现对不同资源的调度，同时该模块可以实现通过并行运算提高并行性。

1. 增加内聚减少耦合模块

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **#** | **模式名称** | **支持客户扩容数** | **支持物理设备增加** | **成本** | **开始时间** |
| 1 | 封装 | >15million | >100 | 中 | 详细设计时 |
| 2 | 限制依赖 | >15million | >100 | 高 | 体系结构设计时 |
| 3 | 增加语义凝聚 | >15million | >100 | 高 | 详细设计时 |

理由：

以上模式均可以定程度上提高系统的可扩展性，限制依赖和增加语义凝聚成本较高，限制依赖需要再体系结构设计时提出，封装和增加语义凝聚在详细审核及是实现。为实现可扩展性，以上的策略都可以为考虑因素，但需要考虑人力和时间成本。

决定：

选择封装模式、限制依赖战略和增加语义凝聚模式

推论：

在功能扩展模块代码需要实现封装，并在体系结构设计和详细设计时注重限制模块间依赖和增加语义凝聚。

1. 减少模块大小战略

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **战略名称** | **支持客户扩容数** | **支持物理设备增加** | **成本** |
| 1 | 模块划分 | >15million | >100 | 中 |
| 2 | 控制模块资源 | >15million | >100 | 汇总 |

理由：

例举的模式均可实现15million客户的扩展数，且均可一定程度上支持新的物理设备的添加。在控制成本的情况下（限制2）均可以作为可选方案。

决定：

选择模块划分及控制模块资源模式

推论：

系统针对NOSS划分出NOSS Server，对于网络操作的交互由NOSS Server进行控制、分配和管理，以直接请求-响应和建立连接-请求-响应两种方式向NOSS系统发送要求NOSS重新配置网络、向NOSS申请获取现有网络的状态或能力、要求NOSS“锁住”某些资源这三种请求；针对Billing模块划分出Billing Server，与Corps DB交互，向Billing系统广播客户提出的请求并接受对方对系统设置提出的更改需求，将需求反馈给broker。针对下游系统划分出Downstream Server，通过P/S component向Downstream Systems广播请求。

1. 可选帮助用户初始化战略

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **模式名称** | **实现容易程度** | **成本** |
| 1 | 取消 | 容易 | 低 |
| 2 | 整合 | 中 | 中 |

理由：

对于用户而言，取消任务较为简单，但无法满足用户功能性需求。整合耗费一定成本但在可以容忍范围内，故选择整合模式。

决定：

选择整合模式

推论：

通过整合方式帮助用户初始化。

1. 可选支持系统初始化模块

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **模式名称** | **实现容易程度** | **时间成本** | **空间成本** |
| 1 | 维护任务模型 | 中 | 高 | 中 |
| 2 | 维护用户模型 | 中 | 中 | 中 |
| 3 | 维护系统模型 | 高 | 高 | 高 |

理由：

为满足系统可用性需求（场景4），且由限制4可知高可用性不能通过硬件方式来实现，则维护系统模型战略排除。而维护任务模型和用户模型实现容易程度较好，时间成本和空间成本较为可控，则均可选择为方案。

决定：

选择维护用户模型模式和维护任务模型模式

推论：

可用性模块需要维护任务模型和用户模型。

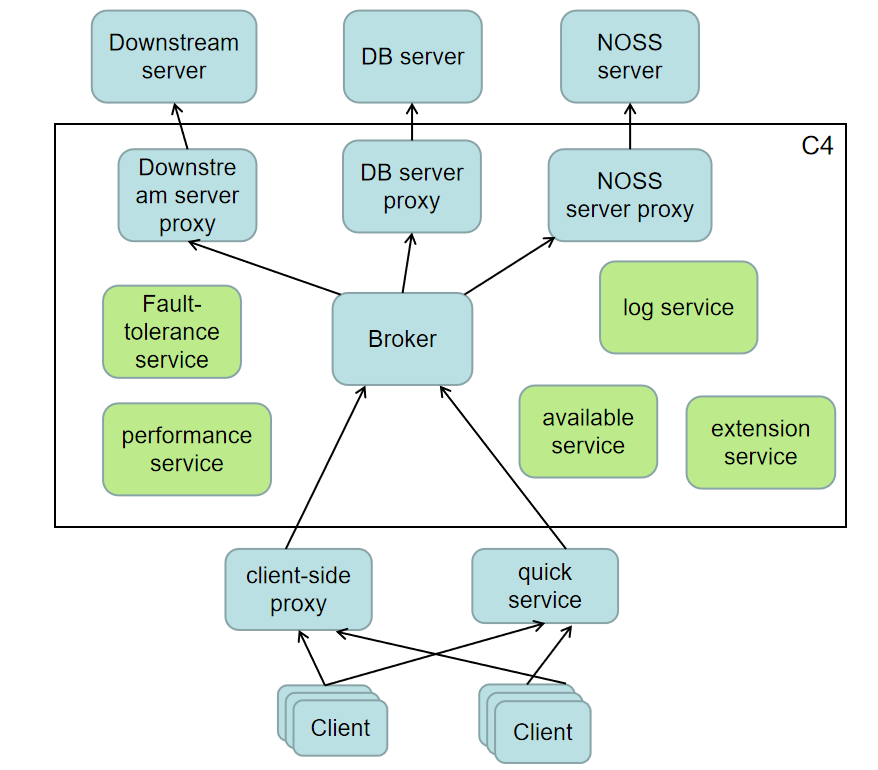
1. 决定所选战略和ASR之间的关系
2. 战略和ASR的映射关系

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **模式类型** | **模式选择** | **结构驱动** |
| 0 | #Replicas | Two Replicas | Two Replicas(DC#3) |
| 1 | 错误检测 | 异常检测模式 | 面对中断和冲突保持高可靠性（SC1） |
| 2 | 错误恢复 | 异常恢复模式  重启模式 | 面对中断和冲突保持高可靠性（SC1） |
| 3 | 错误预防 | 简单预防模式 | 系统前期的设备投资应该维持在一个较低的水平(DC1) |
| 4 | 控制资源需求 | 资源优先级模式  控制时间模式 | 满足大量用户并行使用系统（SC2）  系统前期的设备投资应该维持在一个较低的水平(DC1) |
| 5 | 管理资源利用 | 资源调度模式  并行运算模式 | 满足大量用户并行使用系统（SC2） |
| 6 | 增加内聚减少耦合 | 封装模式  限制依赖模式  增加语义凝聚模式 | 可满足客户容量增长和物理设备增加，保持系统可扩展性(SC3) |
| 7 | 减少模块大小 | 模块划分  控制模块资源模式 | 可满足客户容量增长和物理设备增加，保持系统可扩展性(SC3)  系统前期的设备投资应该维持在一个较低的水平(DC1)  系统的高可用性不能通过容错硬件方式来实现(DC2) |
| 8 | 帮助用户初始化 | 整合模式 | 维护系统可用性(SC4)  系统的高可用性不能通过容错硬件方式来实现(DC2) |
| 9 | 支持系统初始化 | 用户模型模式  维护任务模型模式 | 维护系统可用性(SC4)  系统的高可用性不能通过容错硬件方式来实现(DC2)  系统工作站上没有持久的数据缓存来限制本地故障的影响(DC3) |

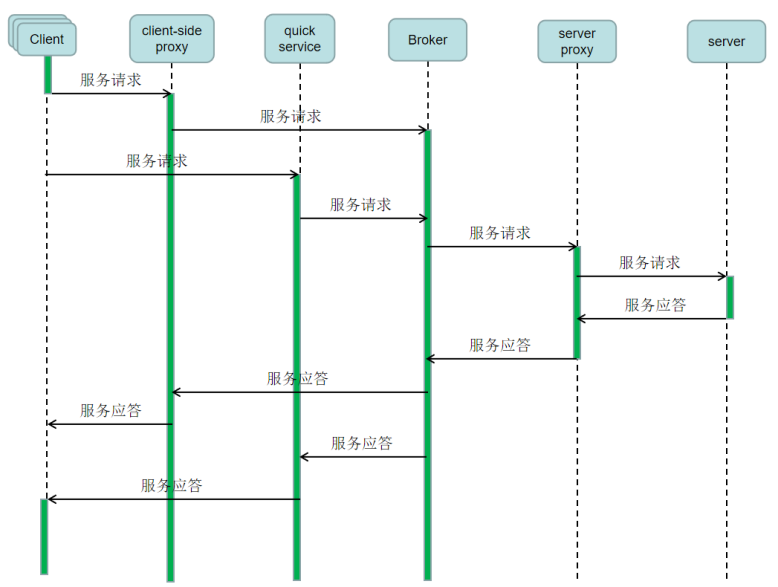
1. 元素表

|  |  |  |
| --- | --- | --- |
| **#** | **元素** | **是否在ADD迭代中涉及** |
| 1 | Broker | 1 |
| 2 | NOSS Server | 1 |
| 3 | Billing Server | 1 |
| 4 | Downstream Server | 1 |
| 5 | Fault-tolerance service | 1 |
| 6 | performance service | 1 |
| 7 | Log service | 1 |
| 8 | Available service | 1 |
| 9 | Extension service | 1 |

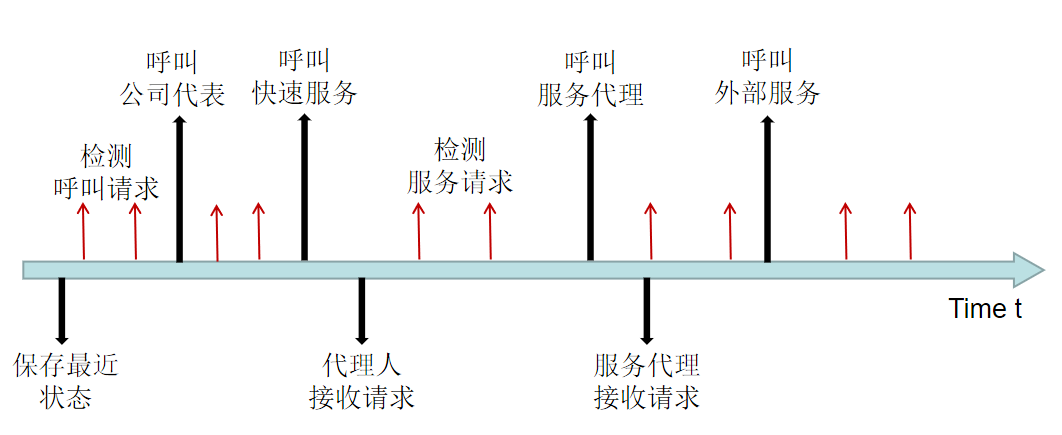
1. 体系结构元素视图



1. 时序图



五、评估和解决不一致



1. 事件发生顺序
2. 客户1呼叫公司代表请求服务。
3. 客户2呼叫快速服务。
4. 公司代表向服务代理请求NOSS等具体服务。
5. 代理呼叫NOSS等服务代理。
6. NOSS服务代理负责执行NOSS服务。
7. 快速服务系统向代理人发出服务请求。
8. 代理呼叫NOSS等服务代理。
9. 代理呼叫NOSS等服务代理。
10. 服务完毕，系统恢复到初始状态。
11. 时间评估

Ta：用户1呼叫公司代表，与公司代表建立连接（2 seconds）

Tb：公司代表向代理人发出服务请求（0.5 second）

Tc：代理人向服务代理请求服务（1 second）

Td：服务代理执行并完成服务（0.8 second）

Te：用户获取到服务完成的反馈，断开连接（0.2 second）

Tf：用户2请求快速服务（0.5 second）

Tg：快速服务系统向服务代理请求服务（0.3 second）

T1=Ta+Tb+Tc+Td+Te=2+0.5+1+0.8+0.2=4.5(seconds)

T2=Tf+Tg+Tc+Td+Te=0.5+0.3+1+0.8+0.2=2.8(seconds)

T=T1+T2=4.5+2.8=7.3(seconds)

1. 可能的时间解决方案
2. 使用完全的串行方式耗时巨大，用户体验非常差，尤其是用户拨打的高峰期，耗时过多。
3. 用户1向公司代表请求服务的同时，用户2可以同时请求快速服务，若来自公司代表或来自快速服务的请求同时到达代理人时，将事件添加进事件队列，按照先到先得或者根据优先级先后调用服务代理，执行服务。当队列将要溢出时，暂时拒绝用户的服务请求。
4. 在NOSS服务尚未执行完成时，下游系统、计费系统对应的公司数据库其实处于空闲状态，可以充分利用这段时间，实现任务的并行执行。
5. 时间决策

|  |  |  |  |
| --- | --- | --- | --- |
| # | 描述 | 初始时间 | 最终时间 |
| Ta | 用户1与公司代表建立连接 | 2.0 | 2.0 |
| Tb | 公司代表向代理人发出服务请求 | 0.5 | 0.5 |
| Tc | 代理人向服务代理请求服务 | 1.0 | 1.0 |
| Td | 服务代理执行并完成服务 | 0.8 | 0.8 |
| Te | 用户获取服务反馈并断开连接 | 0.2 | 0.2 |
| Tf | 用户2请求快速服务 | 0.5 | 0（与Ta并行） |
| Tg | 快速服务系统向服务代理请求服务 | 0.3 | 0（与Ta并行） |

六、为元素分配职责

1. Broker：
2. 接收来自客户端代理和来自快速服务系统的请求，将接收到的请求存入先进先出队列或者优先级队列，队列将要溢出时，给客户端代理和快速服务系统发送反馈，暂时拒绝接收客户请求。
3. 并将收到的服务请求按先进先出的顺序或者一定的优先级转发给各个服务代理。
4. 等待服务代理服务的同时，将不冲突的其它服务请求继续发送给其它的服务代理，并等待服务代理执行任务后给出反馈。
5. 接收服务代理给出的反馈，将反馈发送给原来的请求者，即对应的客户端代理或快速服务系统。
6. 下游服务代理：接收来自Broker对下游系统服务的请求，与下游系统建立连接并完成服务，并给Broker发送反馈。
7. 公司数据库服务代理：接收来自Broker对计费系统服务的请求，建立与公司数据库的连接，通过公司数据库系统间接与计费系统交互，完成服务后给Broker发送反馈。
8. NOSS服务代理：接收来自Broker对网络操作支持系统的服务请求，与NOSS系统建立连接并完成服务，并给Broker发送反馈。

七、接口总结

|  |  |  |  |
| --- | --- | --- | --- |
| 来源元素 | 目的元素 | 接口 | 时间条件 |
| 客户端代理 | Broker | 服务请求 | 0.4 second |
| 快速服务系统 | Broker | 服务请求 | 0.2 second |
| Broker | 下游服务代理 | 服务请求 | 0.8 second |
| Broker | 公司数据库服务代理 | 服务请求 | 0.8 second |
| Broker | NOSS服务代理 | 服务请求 | 0.8 second |
| Broker | 客户端代理 | 服务应答 | 0.1 second |
| Broker | 快速服务系统 | 服务应答 | 0.1 second |
| 下游服务代理 | Broker | 服务应答 | 0.2 second |
| 公司数据库服务代理 | Broker | 服务应答 | 0.2 second |
| NOSS服务代理 | Broker | 服务应答 | 0.2 second |

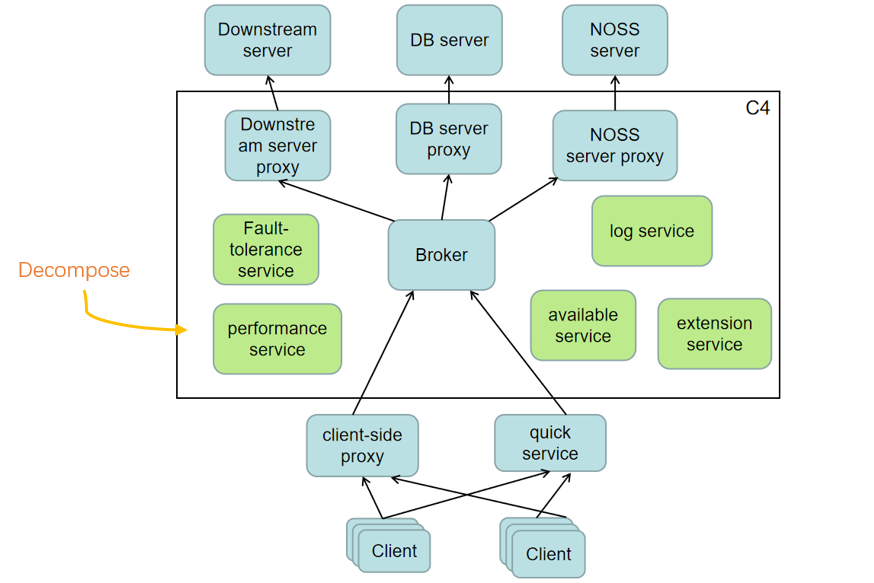
八、重要的ASRs

|  |  |  |
| --- | --- | --- |
| **#** | **构架驱动** | **选择的模式** |
| 1 | 场景1：面对中断和冲突保持高可靠性 | 异常检测模式、异常恢复模式、重启模式、简单预防模式 |
| 2 | 场景2：满足大量用户并行使用系统 | 资源优先级模式、控制时间模式、资源调度模式、并行运算模式 |
| 3 | 场景3：可满足客户容量增长和物理设备增加，保持系统可扩展性 | 封装模式、限制依赖战略、增加语义凝聚模式、模块划分、控制模块资源模式 |
| 4 | 场景4：维护系统可用性 | 整合模式、维护用户模型、维护任务模型 |
| 5 | 需求1：响应Quick Service发出的请求 | 资源优先级模式、资源调度模式 |
| 6 | 需求2：能够向NOSS系统发送要求配置网络、向NOSS申请网络状态 “锁住”某些资源这三种请求 | 控制时间模式、简单预防模式 |
| 7 | 需求3：能够向Downstream Systems广播请求 | 控制时间模式、资源调度模式、并行运算模式 |
| 8 | 需求4：对下游系统发出的重要业务事件进行及时的响应和处理 | 资源优先级模式、控制时间模式 |
| 9 | 需求5：根据客户需求发送各种请求给NOSS、下游系统等系统，等待响应并完成服务 | 资源调度模式、并行运算模式 |
| 10 | 需求6：能够提供关于可供选择的产品或产品簇的建议 | 资源调度模式、资源优先级模式 |
| 11 | 限制1：系统前期的设备投资应该维持在一个较低的水平 | 限制依赖战略、资源优先级模式、控制模块资源模式 |
| 12 | 限制2：系统的高可用性不能通过容错硬件方式来实现 | 整合模式、维护用户模型、维护任务模型 |
| 13 | 限制3：系统工作站上没有持久的数据缓存来限制本地故障的影响 | 维护用户模型，维护任务模型 |
| 14 | 限制4：当地工作站没有专门的管理员来维护停机时间 | 整合模式、维护任务模型 |

**第二次迭代**

1. 选择用于分解系统的元素

选择performance service作为分解系统的元素。



1. 选择ASR

第一次迭代的元素选择是整个C4系统，对系统的功能性、非功能性及限制进行分析识别出ASR如下表。

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **构架驱动** | **重要性** | **难度** |
| 1 | 场景1  满足大量用户并行使用系统 | 高 | 中 |
| 2 | 需求1：  能够同时进行大量的并行任务 | 高 | 高 |
| 3 | 需求2：  为完成任务提供完善的支持和良好的性能支持 | 中 | 低 |
| 4 | 限制1  系统前期的设备投资应该维持在一个较低的水平 | 中 | 中 |

1. 选择满足ASR的设计构思
2. 识别设计理念

|  |  |
| --- | --- |
| **设计理念** | **从属设计理念** |
| 控制资源需求 | 管理事件率 |
| 控制事件响应 |
| 控制资源的使用 |
| 资源仲裁 |
| 管理资源利用 | 增加可用资源 |
| 保持数据或计算的多个副本 |

1. 为从属设计理念列举可选择的战略并选择（step4.2&4.3）
2. 可选管理事件率战略

参数识别：系统需要满足大量用户并行使用系统（场景1）

系统能够同时进行大量的并行任务（需求1）

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **模式名称** | **性能提高程度** | **对系统影响** |
| 1 | 周期性更新DB数据 | 中 | 中 |
| 2 | 降低记录日志的频率 | 中 | 高 |
| 3 | 降低向NOSS查询现有网络状态或能力的频率 | 高 | 中 |

理由：

降低日志记录频率尽管可以降低事件率，但是很有可能影响可用性，破坏容错服务。而且它对易用性有着较大的影响，因为日志频率的降低会使系统对于持续时间长、断断续续的会话的支持不完全，在一定程度上会带来客户的不满。

尽管周期性更新DB数据会使得database中的数据有一定的延迟，不能保持完全的同步，但是这对效率的提高有着比较好的效果，而且可以选择一个适中的周期长度，在提高效率和保持同步中间做一个妥协。

降低向NOSS查询现有网络状态或能力的频率可以显著提高broker的性能，因为broker会是一个负载较大的节点，除了必要的任务，应该尽量少地发起请求。而这一行为对可用性的影响可以由其他错误检测战术代替，所以不会对系统造成过多影响。

决定：

选择周期性更新DB数据和降低向NOSS查询现有网络状态或能力的频率

推论：

性能服务模块可能需要由一个可以暂存DB已更新数据的存储区域，如果可以的话，对已更新不久的数据的查询应该首先考虑性能服务模块中的暂存更新区域。

(2)可选控制事件响应战略

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **模式名称** | **性能提高程度** | **对系统影响** |
| 1 | 接受下游系统消息时C4合并发送回复 | 中 | 低 |
| 2 | 控制NOSS问题报告数量 | 中 | 中 |
| 3 | 减少通知和广播的数量 | 高 | 中 |
| 4 | 采取所需通讯量较少的错误检测方法 | 中 | 中 |

理由：

接受下游系统消息时C4合并发送回复可以避免一条消息对应一条回复的情况，这减少了网络通讯量的同时也确保了能够收到下游系统的消息。可以对性能提高产生帮助，对系统影响较小。

控制NOSS问题报告的数量可以对性能提高带来有益的影响，但是如果没有合适的筛选问题报告的方法，可能会破坏系统的可用性。

在减少通知和广播的数量这一模式中，尤其是减少广播这一策略可以有效地提高系统性能。而且，可以用替代方法传达点对点的通知，不会对系统产生较大的影响。

如果采取所需通讯量较少的错误检测方法，那么尽管通讯量的减少可以带来性能提升、占用网络资源的减少，但是选取其他错误检测方法对系统带来的影响是个未知数。

决定：

选择接受下游系统消息时合并发送回复和减少通知和广播数量的方法

推论：

由于下游系统是通过发送异步消息来通知C4重要的商业活动信息，那么C4的合并消息回复可以在间隔一段合适的时间后没有收到下游系统再次通知时发送，此条消息包括此次收到消息的ID，这可以做到减少通讯量和错误检测。

（3）可选控制资源的使用战略

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **模式名称** | **成本** | **性能提高程度** | **对系统影响** |
| 1 | 使用中介来减小开销 | 中 | 中 | 低 |
| 2 | 限制执行时间 | 低 | 高 | 低 |
| 3 | 限制队列大小 | 低 | 高 | 低 |

理由：

使用中介可以增加用于处理流事件的资源，从而减少开销，做到对资源使用的控制。由于他需要新增中间部件，因此会带来一定成本。

限制执行时间可以规定某一操作将可以花费的最长时间，比如应用在C4对NOSS系统发起重新配置服务请求时，如果执行时间过长超过规定时间，那么可以认定任务执行产生了问题，就会终止任务并重新发起，而不是浪费时间继续等待。

限制队列大小是指控制可以处理事件的数量，比如限制消息队列的大小，不接受过多事件的涌入，可以提高系统性能。

决定：

采用限制执行时间或限制队列大小的方法

推论：

控制资源的使用可以使系统做到对用户需求的响应做出适当的调整，由此达到性能的一定提升。

（4）可选资源仲裁战略

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **模式名称** | **性能提高程度** | **对系统影响** |
| 1 | 先进先出 | 低 | 低 |
| 2 | 动态优先级 | 高 | 中 |
| 3 | 固定优先级 | 中 | 低 |

理由：

先进先出是一种同等看待每个资源的资源仲裁策略，这种方法简单而实用，但是在性能提高方面作用不大。

为事件选定优先级可以依据需求的紧迫程度来处理事件，比起先进先出有着更为灵活的优势。而优先级的选择可以分为动态优先级和固态优先级。

但是动态优先级可能需要额外的开销，因此在这里选择折中的固定优先级用以在较少开销的情况下达到灵活的资源处理顺序的目的。

决定：

采用固定优先级

推论：

固定优先级是指事先为事件分配优先级，分配的策略可以采用语义重要性、时限时间单调、速率单调等多种方法。

（5）可选增加可用资源战略

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **#** | **模式名称** | **成本** | **对并发的支持** | **性能提高程度** | **对系统影响** |
| 1 | 增加服务器资源 | 高 | 中 | 高 | 低 |
| 2 | 增加系统内存 | 高 | 中 | 中 | 低 |
| 3 | 新增secondarybroker | 中 | 高 | 高 | 中 |

理由：

为提高性能最直接的方法就是增加可用的资源，但是增加服务器资源或增加系统内存都会显著增加成本。考虑到系统的初期成本不能过高，所以增加服务器资源或系统内存不应是首先考虑的选项。

而在性能服务中新增组件secondarybroker，可以通过并发缓解当任务并行量过多时，broker压力过大的问题。

决定：

选择新增secondarybroker组件

推论：

secondarybroker也可以为broker处理一些低优先级的日常事务，如记录日志等，这可以让broker专注于优先级更高的事件，同时避免单点故障。

（6）可选保持数据或计算的多个副本战略

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **模式名称** | **成本** | **性能提高程度** | **对系统影响** |
| 1 | 新增cache区 | 中 | 较高 | 低 |
| 2 | 新增数据库副本 | 高 | 高 | 中 |
| 3 | 新增NOSS副本 | 高 | 高 | 低 |

理由：

保持数据的多个副本可以提供更高的查询效率，从而提高性能。

在可选的两种方案中，新增数据库副本可以有效提高数据查询效率，因为可以选择更近的、空闲的副本进行查询，但是这种方式成本较高。

而新增cache区是指将一部分的客户数据放在性能服务的cache区中，在进行查询时，可以首先访问cache区，丢失之后再去访问数据库。，因此划出一个cache区可以对性能带来较好的改善，也能将成本限制在一定范围里。

新增NOSS副本可以让broker选择空闲的NOSS发送请求，达到负载均衡，提升性能的效果。但是这一战略成本很高。

选择：

新增cache区

推论：

Cache区可以存储对数据查询功能使用频率较高的客户的数据或者更新服务不久的客户的数据，这部分客户在可见的未来对于数据库查询的需求程度会高于其他客户。

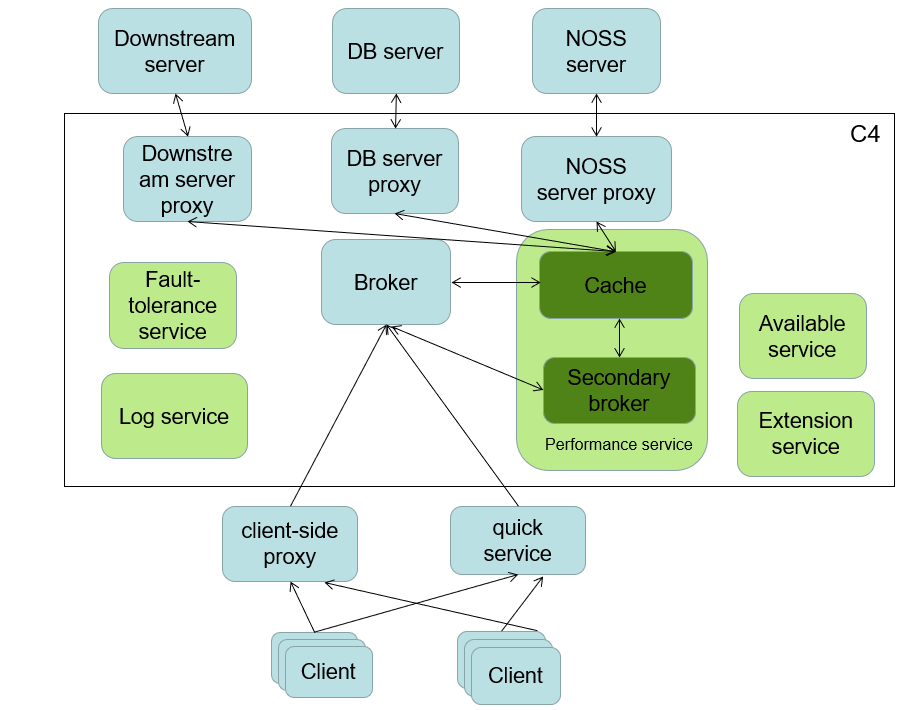
1. 决定所选战略和ASR之间的关系

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **模式类型** | **模式选择** | **结构驱动** |
| 1 | 管理事件率 | 周期性更新DB数据；  降低向NOSS查询现有网络状态或能力的频率 | 满足大量用户并行使用系统（SC1）  系统前期的设备投资应该维持在一个较低的水平（DC1) |
| 2 | 控制事件响应 | 接受下游系统消息时合并发送回复；  减少通知和广播数量的方法 | 满足大量用户并行使用系统（SC1） |
| 3 | 控制资源的使用 | 限制执行时间；  限制队列大小 | 满足大量用户并行使用系统（SC1）  系统前期的设备投资应该维持在一个较低的水平(DC1) |
| 4 | 资源仲裁 | 固定优先级 | 满足大量用户并行使用系统（SC1 |
| 5 | 增加可用资源 | 新增secondarybroker组件 | 满足大量用户并行使用系统（SC1）  系统前期的设备投资应该维持在一个较低的水平（DC1) |
| 6 | 保持数据或计算的多个副本 | 新增cache区 | 满足大量用户并行使用系统（SC1）  系统前期的设备投资应该维持在一个较低的水平（DC1) |

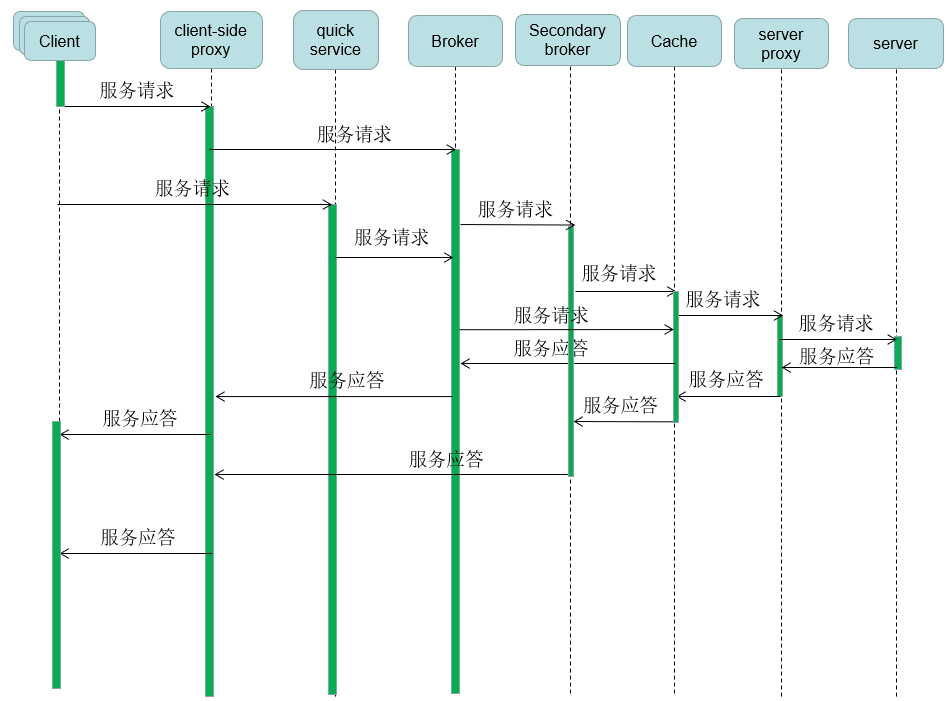
1. 捕捉初步架构视图
2. 元素表

|  |  |  |
| --- | --- | --- |
| **#** | **元素** | **是否在ADD迭代中涉及** |
| 1 | Broker | Requirement |
| 2 | NOSS Server | Requirement |
| 3 | DB Server | Requirement |
| 4 | Downstream Server | Requirement |
| 5 | Fault-tolerance service | 1 |
| 6 | Performance service | 1 |
| 7 | Log service | 1 |
| 8 | Extension service | 1 |
| 9 | Availability service | 1 |
| 10 | Secondary broker | 2 |
| 11 | Cache | 2 |

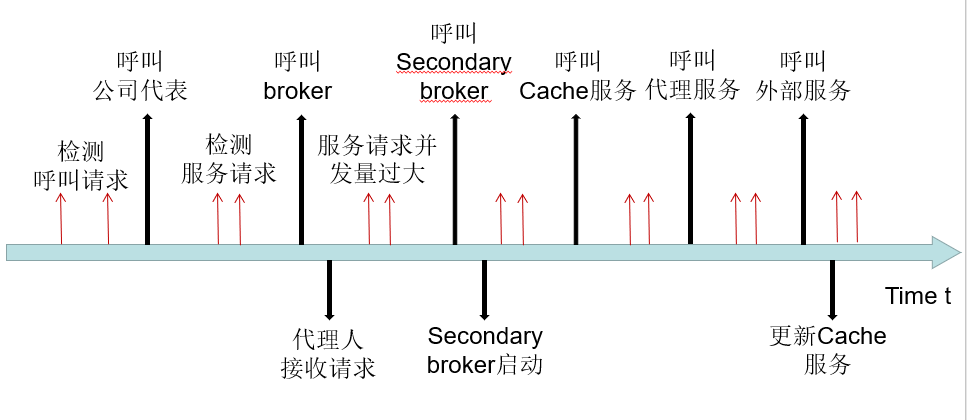
1. 体系结构元素视图



1. 时序图



1. 评估和解决不一致



* + - 1. 事件发生顺序

客户向公司代表请求服务

公司代表向broker呼叫服务

Broker接受服务请求

有pb的概率broker负载过大，请求秘书服务

pb的概率下Secondary broker启动

broker或Secondary broker向Cache请求服务

有pc的概率Cache中已有所需数据，直接返回响应

（1-pc）的概率下，Cache向代理服务发送请求

（1-pc）的概率下，代理服务呼叫外部服务

（1-pc）的概率下，外部服务返回响应

完成服务，保存日志，恢复初始状态

* + - 1. 时间评估

Tlink：客户呼叫公司代表，与公司代表建立连接（2 seconds）

Tar：公司代表向Broker发出服务请求（0.5 second）

Tsr：Broker向Secondary broker发送服务请求（0.1 second）

Tsb：Secondary broker启动并开始工作（0.5 second）

Tca：Broker或Secondary broker请求Cache服务（0.1 second）

Tcp：Cache中存在所需数据，直接响应（0.2 second）

Tcr：Cache中无所需数据，向服务代理发送请求（0.1 second）

Tsr：服务代理执行并完成服务（0.8 second）

Tcrf：Cache更新数据（0.1 second）

Te：用户获取到服务完成的反馈，断开连接（0.2 second）

T = Tlink + Tar + (Tsr + Tsb) \* pb + Tca + Tcp \* pc + (Tcr + Tsr) \* (1 - pc) + Tcrf +Te

= 2 + 0.5 + (0.1 + 0.5) \* 0.25 + 0.1 + 0.2 \* 0.5 + (0.1 + 0.8) \* 0.5 + 0.1 + 0.2 = 3.6(seconds)

* + - 1. 可能的时间解决方案

使用完全的串行方式耗时巨大，用户体验非常差，尤其是用户拨打的高峰期，耗时过多

Cache服务的命中率是提高Cache服务质量的关键所在，一个高的命中率可以大幅提高broker与外部服务的交互效率，但同时一个低的命中率会使得Cache服务的存在反而使整个系统变得低效，因此应根据不同的现实状况，设计多种合理的Cache算法以保证其命中率

Cache服务更新的同时，可以将获取到的数据同时传回给broker以再次传回给公司代表和客户使用， 此部分可以采取并行的方式以减少时间提高效率

Secondary broker启动后应直接在一段时间内和broker共同工作成为broker的工作助手，而不应在完成单次的任务分配后就进入休眠状态，这是效率极低的一种安排。取而代之的，应该是一种根据当前客户并发量、任务并发量估计计算得出的一个检查周期，检查周期内Secondary broker不休眠继续工作，检查点检查并发量变化，若并发量减小至动态阀值，则使Secondary broker休眠，否则继续工作至下一周期结束进行下一次检查

* + - 1. 时间决策

|  |  |  |  |
| --- | --- | --- | --- |
| # | 描述 | 初始时间 | 最终时间 |
| Tlink | 客户呼叫公司代表，与公司代表建立连接 | 2.0 | 2.0 |
| Tar | 公司代表向Broker发出服务请求 | 0.5 | 0.5 |
| Tsr | Broker向Secondary broker发送服务请求 | 0.1 | 0.1 |
| Tsb | Secondary broker启动并开始工作 | 0.5 | 0/0.5（周期内为0） |
| Tca | Broker或Secondary broker请求Cache服务 | 0.1 | 0.1 |
| Tcp | Cache中存在所需数据，直接响应 | 0.2 | 0.2 |
| Tcr | Cache中无所需数据，向服务代理发送请求 | 0.1 | 0.1 |
| Tsr | 服务代理执行并完成服务 | 0.8 | 0.8 |
| Tcrf | Cache更新数据 | 0.1 | 0（与Te并行） |
| Te | 用户获取到服务完成的反馈，断开连接 | 0.2 | 0.2 |

1. 为元素分配职责
   * + 1. Cache：

根据合适的算法保存最近使用过的一些来自外部服务的数据，提高查询效率

当Cache中找不到对应数据时，Cache需要充当一个数据搬运者的角色，将数据取出，进行自身更新并把数据并行传回给broker

* + - 1. Secondary broker：

不是Broker的备份，而是助手和秘书的职责，在Broker负载过高的时候与Broker并行处理请求

应根据当前客户并发量、任务并发量估计计算得出一个检查周期，检查周期内Secondary broker不休眠继续工作，检查点检查并发量变化，若并发量减小至动态阀值，则使Secondary broker休眠，否则继续工作至下一周期结束进行下一次检查，即自我检查外部环境并决定工作时间

1. 接口总结

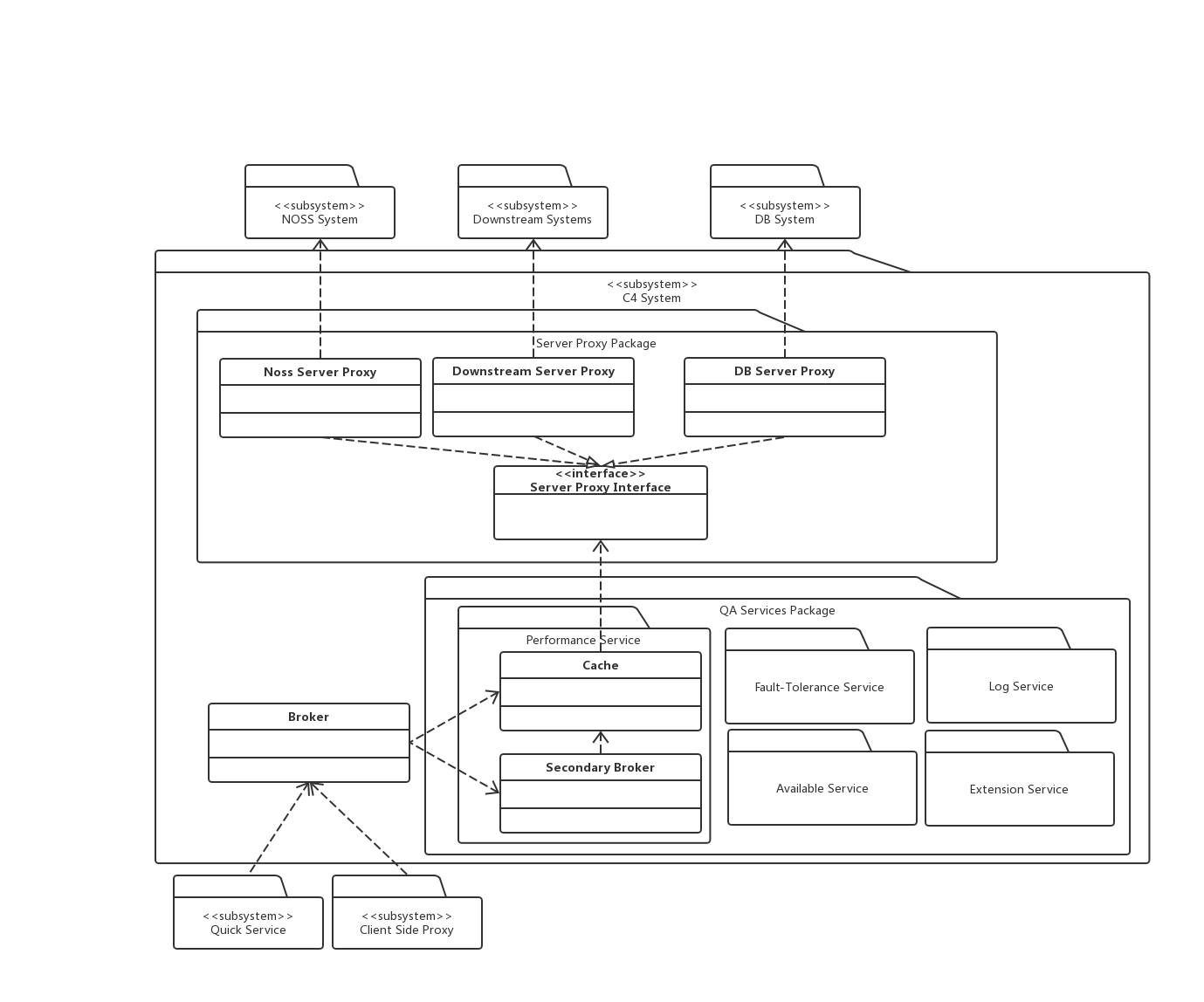
|  |  |  |  |
| --- | --- | --- | --- |
| 来源元素 | 目的元素 | 接口 | 时间条件 |
| Broker | Secondary broker | 服务请求 | 0.1 second |
| （Secondary）Broker | Cache | 服务请求 | 0.1 second |
| Cache | NOSS服务代理 | 服务请求 | 0.1 second |
| Cache | 公司数据库服务代理 | 服务请求 | 0.1 second |
| Cache | 下游系统服务代理 | 服务请求 | 0.1 second |
| Cache | （Secondary）Broker | 服务应答 | 0.2/0.9 second |
| 下游服务代理 | Cache | 服务应答 | 0.8 second |
| 公司数据库服务代理 | Cache | 服务应答 | 0.8 second |
| NOSS服务代理 | Cache | 服务应答 | 0.8 second |

七、 重要的ASRs

|  |  |  |
| --- | --- | --- |
| **#** | **构架驱动** | **选择的模式** |
| 1 | 场景1  满足大量用户并行使用系统 | 周期性更新DB数据、降低向NOSS查询现有网络状态或能力的频率、限制执行时间或限制队列大小、新增Secondary Broker |
| 2 | 需求1：  能够同时进行大量的并行任务 | 周期性更新DB数据、降低向NOSS查询现有网络状态或能力的频率、接受下游系统消息时合并发送回复、减少通知和广播数量、固定优先级、新增Cache服务 |
| 3 | 需求2：  为完成任务提供完善的支持和良好的性能支持 | 限制执行时间或限制队列大小、固定优先级、新增Secondary Broker、新增Cache服务 |
| 4 | 限制1  系统前期的设备投资应该维持在一个较低的水平 | 新增Secondary Broker、新增Cache服务 |

**第二部分：最终软件架构文档**

1. 模块视图

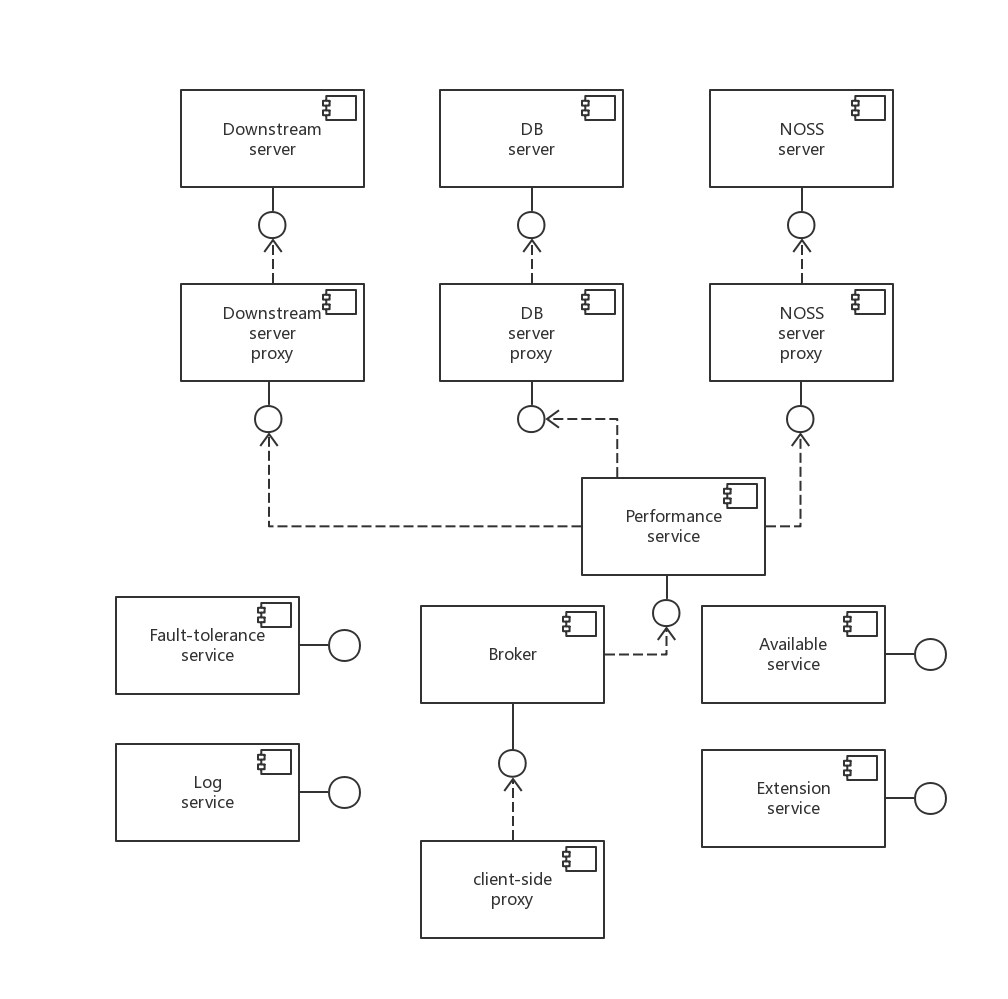


总体上采用Broker的架构模式，以C4系统中的Broker类作为Broker，负责分配任务、调用其他系统，这与整个系统中C4所扮演的任务协调者、中心控制者的角色十分符合。NOSS、Downstream systems、DB system作为子系统接受C4的任务调度和分配。

对于不同的子系统调用，使用不同的具体化的服务代理，这些代理统一实现Server Proxy Interface接口，实现C4 Broker和子系统之间的低耦合。抽象接口，使得在子系统发生改变时只需改变对应服务代理的代码而不影响其他部分代码，新的子系统的添加也成为可能。

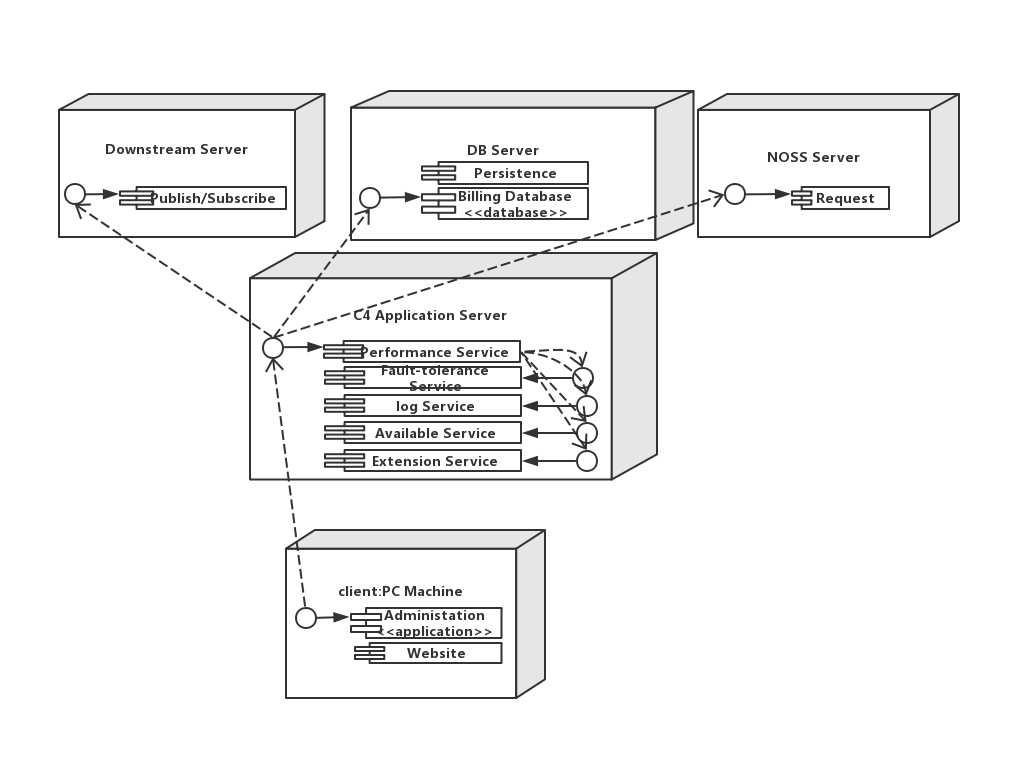
将所有QA Services放入一个软件包，表示其在逻辑上属于一个类别的服务，提高抽象化程度，但不为它们设置统一的接口或父类——不同服务的功能不同、内部方法和成员变量不同、仅是在逻辑上都属于QA Services，但并没有足够的相似性。

2、组件-连接器视图



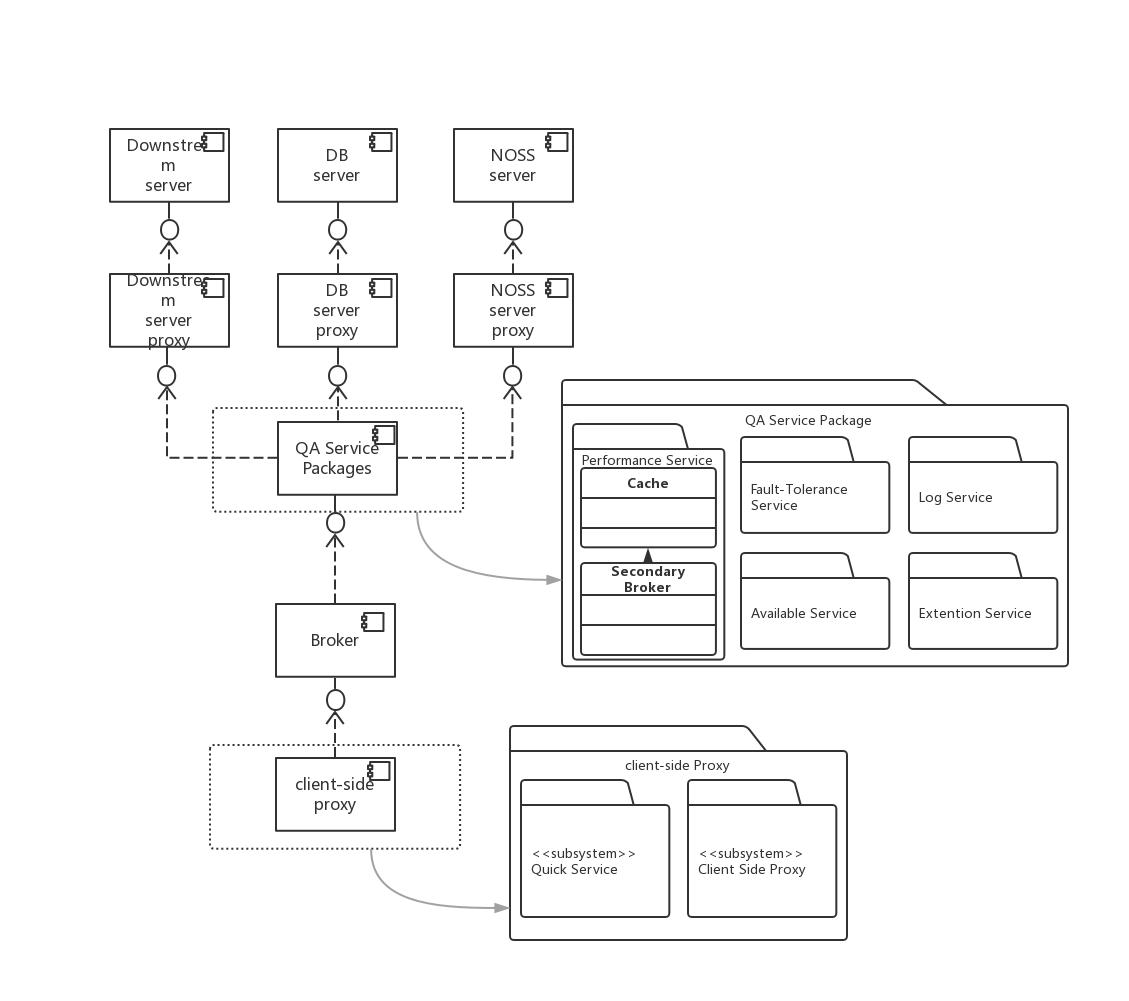
将每个小模块视作一个组件，组件之间通过连接件相连。构件包含了配置组件、工作产品组件和执行组件，接口包含了供接口和需接口，连接器包含代理连接器、组装连接器。每个service都对外提供服务，客户端需要Broker提供的服务，Broker向客户端提供连接服务，通过连接件进行客户端的认证并接入系统，从而实现外部的客户端与C4系统内部的互联。同时Broker请求Perfomance service中cache和Secondary broker的服务。由Perfomance service统一获取来自server proxy的服务。Server proxy与相对应的server建立连接，并获取可能远程的C4系统外部的服务。

3、分配视图



C4的部署图中C4 Application Server面向使用系统的用户，并与Downstream Server、DB Server、NOSS Server通信，分配任务、调用其他Server的服务。C4 Application Server的部件中包含5个Service，其中Performance Service中包括一个Cache和一个Secondary Broker，Cache承担broker任务其他的Service进行交互，传递数据信息。DownStream Server负责和下游的系统进行联系，通过Publish/Subscribe部件和C4系统联系。NOSS Server是个网络提供者，通过named-tag 形式的消息发送网络需要的请求。DB Server负责和Billing系统联系，包含账单的数据库等元素。Client Server需要有C4系统的程序和网络，登录进程序后可以和C4 Application Server交互，发送请求，得到服务。

4、cross-view



Cross-view图将组件-连接器视图和模块视图组合起来，这提高了整个视图的抽象程度，更方便了读者的理解，而在细节方面，cross-view图对QA Service Packages和client-side proxy做了进一步的模块分解，展示了performance service、fault-tolerance service等多种质量服务，也对关键的performance service做了分解。Cross-view整体的高抽象程度和重要部分的模块分解使得它更加清晰完整，利于理解。

**第三部分：个人总结**

1. 顾明政

在本次的小组作业中，除了参与讨论等活动之外，我主要负责了ADD迭代前功能性及非功能性需求、约束的初步总结，ADD迭代过程中第二次迭代的Step4.4及以后的所有Steps，和文档部分的Module view部分。在使用ADD方法进行设计的过程中，我得到以下经验和总结：

1. ADD方法对于那些之前毫无设计参考的工程而言似乎很难发挥其优势，而更适用于初步架构的完善和具体化。具体来讲，对于一个“green field”项目，我们很难通过ADD的思考方式来确定出其中的功能性模块以及功能性的架构。相反，将ADD的设计方法、思维方式与其他设计方法结合，先得出最基本的系统的功能性架构，再根据ADD对这些架构进行添加、完善、更改，则似乎能够起到卓越的效果。
2. ADD设计方法作为一种多次迭代演化的设计方法，对于那些复杂的系统来说，不失为一种非常好的选择。具体来讲，使用ADD方法可以区分功能性模块和质量属性模块，先对系统整体架构有一个抽象的初步的判断，然后围绕ASRs，选择不同的elements细节，细化设计、根据变化重构设计，但又不至于打破原有总体架构。通过一次次迭代、由质量属性驱动架构的方式，大大简化了系统的复杂度和设计工作的难度。
3. 使用ADD的过程中必须对QA的基础知识十分了解。具体来说，QA的刺激-响应表、应用场景、常用tactics在ADD设计方法中扮演了非常重要的角色，它们常常是设计思考的出发点或重要决策的判断依赖，因此如果对QA基础知识不甚了解，想要通过ADD设计方法来实现较好的设计是困难的。
4. 王青雅

在本次的小组作业中，我主要负责第一轮迭代的后半部分，包括体系结构元素视图、时序图、评估和解决不一致、为元素分配职责、接口总结、重要的ASRs部分以及文档部分的组件连接件视图。同时我还参与了小组作业的讨论活动和任务分配。在使用ADD方法的过程中，我有以下体会：

1. 工程设计需要多次迭代、循序渐进、逐步求精。设计不是一件一蹴而就的事情，特别是对于一些复杂系统和复杂模块，无法在设计的初期就做到尽善尽美，需要在设计的过程中不断深入，不断精细化，以求达到设计的最优化。
2. 属性是设计的重要驱动力，在进行软件设计前，首先应该对希望被架构的软件系统有一个全面的认识，充分认识系统的质量属性、非质量属性以及约束条件等，并从中选择最为重要的属性，设计策略以期满足。
3. 为选择出的属性设计策略的时候，每个策略都有利有弊，可能对不同的属性利弊作用不太一样，这时候需要我们评估利弊的大小和属性的重要程度，实现一种权衡，使得系统尽可能最大化地满足需求。
4. ADD设计将质量作为重点关注对象，在满足质量的前提下合理分配功能。在设计的全过程中都以质量为重心，可以保证软件产品的高质量要求。
5. 宗琳

本次小组作业中，我主要负责了ADD第一次迭代中step1-step4的部分，即最初系统的情景设计、功能性非功能性需求的分析、ASR的选择和元素分解的战略选择部分。文档中部署图的部分，同时参与了功能型需求、非功能性需求和约束的小组讨论，总结了第一次迭代中的ASR。

我在ADD方法使用中得到以下经验：

1. 在使用ADD进行迭代前需要对系统的功能性、非功能性需求和限制还有使用场景有一个全面的认识，并通过讨论得出对系统结构的一个统一的认识。
2. ADD步骤中Step1-Step3是后面进行分解的基础，如果ASR的判断不准确，对后面元素分解会有比较大的影响，甚至可能使得设计偏离了中心方向，所以对需求重要程度和实现难度的判断以及最终ASR的选择需要谨慎。
3. 在Step4元素分解的过程中要讲设计重点放在ASR上，一些非功能性需求的实现可能存在矛盾甚至对立的情况，当非功能性需求出现矛盾的时候，战略的选择需要以更重要的ASR作为出发点进行判断。
4. ADD对于最初步的分解较为困难，需要考虑的内容较多，所以我认为ADD适合有一个初步架构的，相对复杂的系统中一个部件的分解和迭代。
5. 陆梦葶

本次设计中，我们组首先经历了较长时间的讨论，确定了迭代的主要思路和任务，经过组内讨论决定分解系统的元素之后，我主要负责两次迭代中第二次迭代的前半部分，选择ASR，为从属设计理念列举可选择的战略，选择最优战略并陈述理由。在架构编档部分，我负责画组合视图的图和解释说明。

在这次迭代中，我对ADD方法有了更加深入的理解。

1. ADD方法多次迭代，每次关注一个元素的方法符合循序渐进，精益求精的思想。每次战略的选择也是个取舍的过程，都要求设计者考虑周全，慎重决定。
2. 在实践这次属性驱动的设计时，一开始我对整个系统的了解仅限于C4本身的功能和其他组件的功能、限制，很难在头脑中构想出整个系统的交互和工作流程，对大致的设计结果也感到迷茫。
3. 但是ADD分步进行，逐层推进的方法让我逐步清楚了设计过程中每个阶段要关注的内容，它分解设计的想法能够让我在每个阶段只关注设计的一小部分，这降低了复杂度，明确了每一小个步骤的工作，能让我专注于当下的设计工作。
4. 而且ADD方法不急着得出结果，他总是在梳理完各个影响因素和关注点之后，给出可能的解决方法，根据ADD设计的架构可能还会故意推迟某种决定，以免某个中间件给之后的架构带来限制，从而给系统带来更多的灵活性。