微服务架构落地实施初步设想

gochant（guochant@gmail.com）

1 总体工作思路

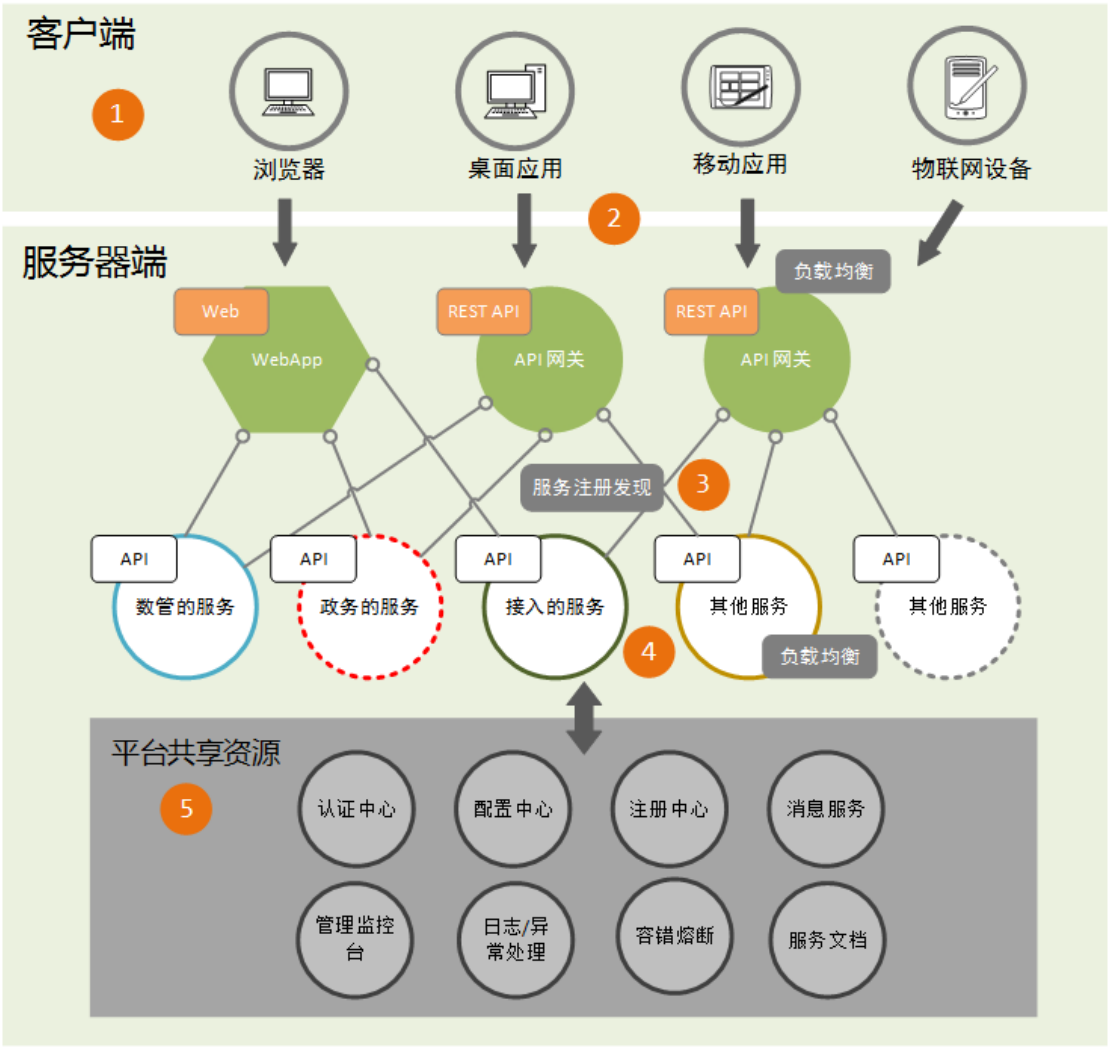
为了在采用不同语言（框架）开发，部署在不同环境下的各个系统间进行协同，在各系统采用面向服务的架构基础上，使用一个公共的服务管理平台。虽然最终目标是要搭建一个比较完备的服务管理（云）平台，但初期的目标还是集中在如何实现各系统间服务的集成这个问题上面来，因此，基于近期或远期的考虑，需要提出一种可行的架构，在实施过程中小步快跑，不断演进。

首先，从架构层面，采用现在比较流行的微服务架构。微服务架构在其他行业（例如互联网）有比较多的实践，但在具体进行行业落地以及结合现在具体实施的现状来看，需要做一些更改和裁剪，把焦点集中在各系统间协同和数据共享这个层面上来。

其次，从实施层面，整个微服务体系搭建比较复杂，不仅是说这些框架和平台本身的搭建，而是分布式系统带来的固有复杂度，包括通讯和测试的开销，设计层面根据独立业务进行的彻底组件化和服务化，以及基础设施管理和运维的成本。因此，在标准的制定和具体实施上，需分清轻重缓急，在某些场合不非拘泥于微服务的最佳实践，根据本项目具体情况，因时因地因情况制宜，在标准（或方案）制定和开发实施上，保持小步快跑，形成一种不断演进的体系。

2 服务框架架构

2.1 总体介绍



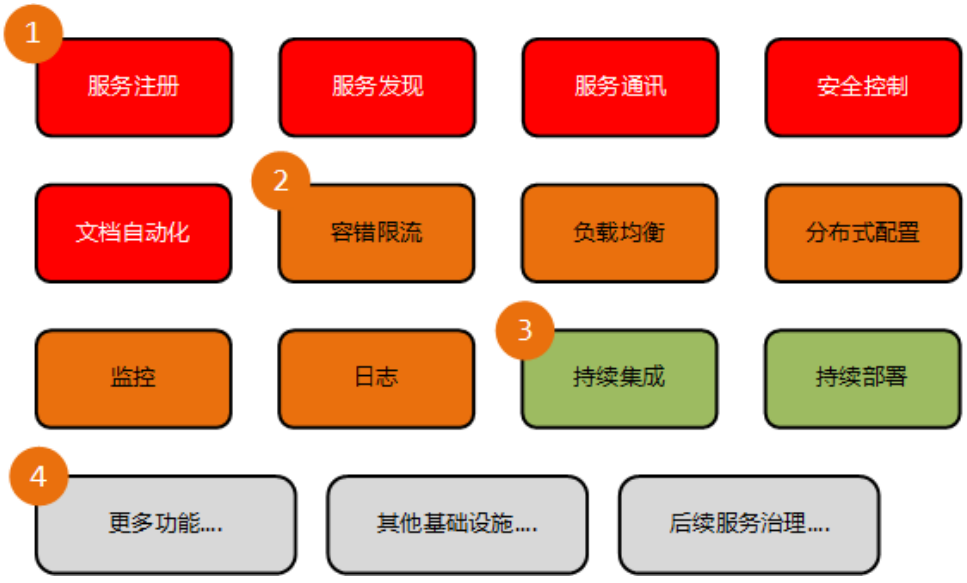
系统架构图说明：

1. 客户端主要分两种：瘦客户端（浏览器）、胖客户端（CS 系统），浏览器后面接的是一个 Web 应用，承载在 Web 服务器上，它可能具备大部分 API 网关的功能，但更多得是需要作为 Web 静态资源的承载器；桌面（本地）应用资源都在本地，因此只需要一个 API 网关作为数据（或业务）的提供方就行了
2. 这里服务器端与客户端交互基本上建议通过 HTTP（S）作为传输协议或其他 Web 友好的协议，由于客户端的调用不在我们的控制范围，因此可能需要做一个负载均衡
3. API 网关和 WebApp 都是服务的消费者，而服务与服务间需要通过统一的注册中心进行注册和管理，这里面要禁止消费者和提供者绕过注册中心点对点直连，而可能仅有在开发测试时才允许这么做

4. 微服务是个笼统的概念，一个系统可能会暴露很多个服务，而具体内部实现没有去定义，你可以采用 N 层架构、事件驱动架构、插件式架构等，也可以在微服务内部共享存储。同时部署方式也不去定义，可以部署实体机，也可以部署成虚拟机、容器（虚线表示）等，仅需要把要共享的功能或数据暴露成服务，并且符合一定的标准就可以了，在这个服务体系中，对于地图服务以外的服务，建议都使用 HTTP + JSON 的 REST 服务（假设使用 REST，那么这个 REST 服务的数据标准需要指定或定义）

5. 把那些在服务管理过程中各系统可能用到的公共资源（组件）大概罗列在了这里，可能不规范，这大体也涵盖了接下来公共的服务管理平台需要做的内容，具体的实现没有去定义，有可能业界已有的开源框架或方案就能涵盖一大部分，也有可能会基于不同的库（框架）去进行定制开发，在有的实现里，有可能很多组件都是放在一起的，因此后续可根据实际情况进行划分。另外，这个架构是一个抽象的表示，并没有把底层的软硬件设施画出来，等最终方案敲定后，在具体实施之前，这些可能才能进行进一步明晰。

2.1.1 功能规划



功能规划图说明：

1. 首先，红色部分是表示当前应优先完成的功能，主要是包含系统间协作，进行数据服务共享、对外服务发布的最基础的一些功能

2. 橙色部分是后面需要进行引入或配置的功能，主要包含对服务的记录监控和对分布式、集群模式下服务健壮性方面的处理，比如可能引入的熔断器、分布式配置管理工具、监控工具等
3. 持续集成（CI）、持续部署（CD）对于整个平台建设来说优先级不高，它不是功能，只是一种自动化开发的模式，可能各系统内部自己也在使用，假如后期各平台协作增多，服务器（节点）过多，迭代频繁，可以考虑在服务平台间引入这些模式
4. 整个架构是不断演进的架构，现在考虑的只是应对当前的情况，还有一些可能的情况都没有考虑，例如如何确保分布式更新的数据一致性，性能监测等，后面应根据实际情况进行不断调整和增加新的组件功能、新的基础设施以及进行更深层次的服务治理

2.1.2 整体评价

	能力	评价
整体灵活性	高	使用这种模式构建的应用往往是松耦合的，也有助于促进改变，能够快速响应不断变化的环境
可伸缩性	高	每个服务组件可以单独扩展，并允许对应用程序进行扩展调整
易于开发性	高	当整个服务体系建立起来后，由于开发范围更小且被隔离，单个应用开发维护变得更简单，同时服务间采用比较简洁的调用方式，相比于传统的 SOA 方案，能减少开发复杂度
易于部署性	分情况	由于该模式的解耦特性和事件处理组件使得部署变得相对简单，但需要配套设施和自动化机制的完善
可测试性	分情况	由于业务功能被分离成独立的应用模块,可以在局部范围内进行测试，这样测试工作就更有针对性，但整个业务链路多个服务间端到端的测试可能比单体架构的测试要复杂
性能	分情况	由于微服务架构模式的分布式特性，通常并不适用于高性能的应用程序，但通常可根据具体场景做性能优化，有可能取决于你所采用的协议，有些协议性能较差，有的可能与本地调用差距不大

2.2 框架基础要件

下面对一些系统架构图上所体现的要件做一些说明。

2.2.1 API（服务）网关

API 网关是一个服务器，也是一个服务，可以说是进入系统的唯一节点。这跟面向对象设计模式中的 Facade 模式很像。API Gateway 封装内部系统的架构，并且提供 API 给各个客户端。它还可能有其他功能，如授权、监控、负载均衡、缓存、静态响应处理等。

使用它主要有以下好处：

- 避免将内部信息泄露给外部客户
- 为微服务添加额外的安全层
- 可支持混合通信协议
- 降低微服务复杂性
- 微服务模拟与虚拟化

2.2.2 服务构建（通讯）框架

这个框架主要解决服务创建、发布、通讯（调用）的问题，严格说来，这个框架属于整个服务体系的范畴，但是不属于服务平台的范畴，因为各个系统完全可以根据自己的实际业务情况，不同的开发语言，选择适合于自己的服务通讯框架。这里仅从可用性角度对该框架应满足的需求提一点建议：

- 适用多种通讯场景
- 支持多种协议
- 兼容多种语言（如果是使用私有通讯方式），例如 Java、C# 等
- 具备安全性方面的设计（应自己去验证权限）
- 性能不能太差，在易用性和性能之间做一个权衡

关于多种通讯场景的说明

	一对一	**一对多**
同步	请求/响应	

异步	通知	发布/订阅

对于大部分场景（可能 80%以上）服务间调用都是同步的请求/响应模式，而一些长时任务或需要多点派发的任务则可能需要使用异步或一对多的通信模式

因此，在同步模式下，可以使用采用 HTTP(S)/JSON 的类 RESTful API 或 SOAP/WSDL 服务，在追求高性能的场合可使用一些二进制协议（库），例如 thrift 等。异步条件下可使用消息协议，例如 AMQP、JMS 等

2.2.3 服务注册中心

服务注册中心的设计是为了更方便的进行服务的管理，使用它能自动通过服务的名称去发现服务，而不必了解这个服务提供的终结点到底是哪台主机，同时增加或移除某个服务的终结点（endpoint）时，对于服务的消费者来说是透明的。它最核心的功能是维护服务注册表，因为这是服务发现的基础（不管是使用服务器端发现还是客户端发现）。从更高层面来说，他应该具有以下功能：

- 具备服务注册、服务订阅功能
- 具备健康检查、状态监测等功能
- 如果需要集群式部署服务，应支持服务的负载均衡（load balance）和故障转移（failover）
- 支持多种语言，例如 Java、C# 或 Python 等
- 如果需要集群式支持分布式集群部署（避免单点故障），需维护服务注册表的强一致性

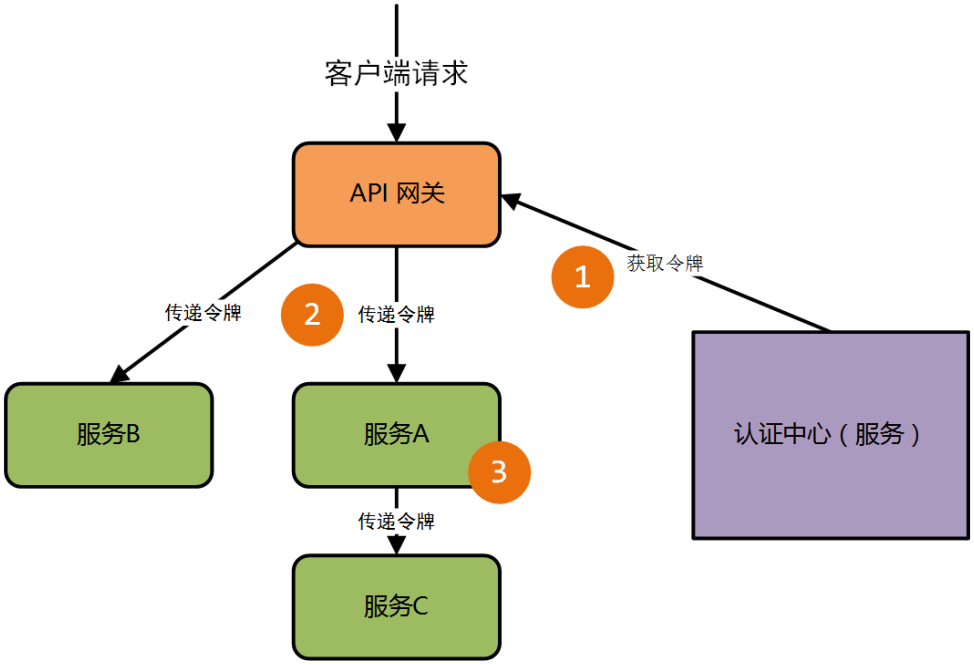
2.2.4 认证中心

服务调用存在一个服务授权的概念，这个认证中心可以和应用级别的单点登录使用一个系统。服务授权包含两个部分：认证和授权。主要的授权模式大概有三种：简单授权（主要通过 IP 地址、域名等过滤）、协议授权（主要通过事先约定的密钥，适合点对点传输）、中央授权。

建议采用中央授权，它的工作模式是引入独立的授权中心，服务调用方每次发起服务调用请求时，先从授权中心获取一个授权码，然后附在原始请求上一起发给服务提供方，提供方收到请求后，先通过授权中心将授权码还原成调用方身份信息和相应的权限列表，

然后决定是否授权此次调用。使用中央授权，简化了服务提供方的实现，让提供方专注于权限设计而非实现，并且提供了一套独立于服务提供方和服务调用方的授权机制，无需重新发布服务，只要在授权中心修改服务授权规则，就可以影响后续的服务调用。

在具体实施中，服务的安全控制主要是放在 API 网关层去解决的，服务间主要是通过令牌（token）去鉴权，而令牌取得和验证则是由统一认证中心去管理，具体如下图：



关于安全控制的说明：

1. 客户端请求，如果没有 token（令牌），则根据某种方式（如用户名密码）去获取 token
2. 在服务间调用传递 token
3. 服务提供方需要验证 token，至于验证的方法就取决具体的实现

2.2.5 负载均衡器

负载均衡需要在多个地方考虑，包括可能的软负载（客户端负载或服务器端负载），这个要结合具体的框架使用情况。如果有必要，也可能考虑硬件负载均衡器。

2.2.6 分布式配置管理

在分布式环境中，出于负载、容错等种种原因，几乎所有的服务都需要在不同的机器节点上部署多个实例。当然，业务项目中总少不了各种类型的配置文件，我们常常会遇到

这样的问题，有时仅仅是一个配置内容的修改，便需要重新进行代码提交 Git/SVN，打包，分发上线的流程。当部署的机器有很多时，分发上线本身也是一个很繁杂的工作。而配置文件的修改频率又远远大于代码本身。分布式配置管理框架的目的就是将配置内容从代码中完全分离出来，及时可靠高效地提供配置访问和更新服务。

2.2.7 分布式消息中间件

这个消息中间件建议采用轻量级的消息队列组件，并且功能只是用在服务间异步通信的场景，不涉及到服务整合、编排之类的业务逻辑，当然异步通信也可以采用其他方式，那么这个组件就是可选的。

2.2.8 API 文档管理

协作需要一套自动化的 API 文档生成、管理工具，避免口头沟通的风险和无约束，以及通过 Word 等文档工具手工管理的低效性和延迟性。在所有服务采用统一标准的基础上，使用语言无关的接口描述语言（IDL），例如 wsdl、swagger（yaml）等，生成在线实时的 API 文档，同时可以有相应配套的测试工具。

2.3 框架选型参考

这里简单的列了一些开源的框架，仅供参考

功能组件	库、框架
服务注册	Eureka, Zookeeper, Etcd, Consul, SmartStack, …
服务构建，通讯框架	REST, Dubbo, Thrift …（太多）
服务网关	Zuul, …
分布式配置	Spring Cloud Config, QConf, Diamond, …
消息队列	RabbitMQ, Kafka, …
熔断器	Hystrix, …
API 文档	Springfox, Swagger, RAML, Blueprint, …
……	……