1. 总则（P1）：

1、发明，是指对产品、方法或者其改进所提出的新的技术方案。

实用新型，是指对产品的形状、构造或者其结合所提出的适于实用的新的技术方案。

2、执行本单位的任务或者主要是利用本单位的物质技术条件所完成的发明创造为职务发明创造。职务发明创造申请专利的权利属于该单位。利用本单位的物质技术条件所完成的发明创造，单位与发明人或者设计人订有合同，对申请专利的权利和专利权的归属作出约定的，从其约定

1. 授予专利权的条件（P5）

1、新颖性，是指该发明或者实用新型不属于现有技术;也没有任何单位或者个人就同样的发明或者实用新型在申请日以前向国务院专利行政部门提出过申请，并记载在申请日以后公布的专利申请文件或者公告的专利文件中。

第三章 专利的申请：（P7）

1、申请发明或者实用新型专利的，应当提交请求书（写明发明或者实用新型的名称，发明人的姓名，申请人姓名或者名称、地址，以及其他事项）、说明书（对发明或者实用新型作出清楚、完整的说明，以所属技术领域的技术人员能够实现为准;必要的时候，应当有附图）及其摘要（简要说明发明或者实用新型的技术要点）和权利要求书（以说明书为依据，清楚、简要地限定要求专利保护的范围）等文件。

第五章（P11）

1、发明专利权的期限为二十年，实用新型专利权和外观设计专利权的期限为十年，均自申请日起计算。

T09:38专利 不要出现 我们 要写本专利 不要写表示程度的词 比如 一些 可能 大部分这种

T09:59 <https://mp.weixin.qq.com/s?__biz=MzIwMzg1ODcwMw==&mid=2247486539&amp;idx=1&amp;sn=18311518fa07a69bfe630b6efa6849a2&source=41#wechat_redirect>

1. 服务的本质就是行为（业务活动）的抽象。【S++】
2. S++（为解决一开始无法包含所有业务，又必须在一开始包含所有业务的矛盾，而定义一个更加抽象的服务定义）通过服务的业务与技术分离彻底将传统服务中和业务无关的技术成分剥离出去，放到服务的外延中去，让服务内涵成为纯粹的业务描述。
3. SOA：一种架构模式，是一种面向服务的思维方式。
4. 服务：一种可重复的业务活动的逻辑上的描述，是一种自包含的、可组织的“黑盒子”。【Open Group】
5. 微服务：强调应用的颗粒度足够小【微服务“微”的并不是服务，其实微的是应用】
6. S++包含两个层面的抽象：
7. 将所有与业务表达无关的技术内容剔除掉，形成一个纯粹的、与技术实现无关的、与业务细节和流程无关的、自包含的业务描述【业务与技术分离】
8. 剔除非要素的业务描述，抽象合并同类的业务要素，从而形成更加形式化的抽象业务模型。【服务多态建模】
9. 实现方法差异：SOA架构主要流行方法是WebServices；S++使用专用的S++容器承载S++服务以及处理S++的多态模型转换；微服务强调轻量化，架构上采用点对点去中心化
10. S++认为只要有需求，那么中心点是必然也必须存在的S++推荐的架构是介于SOA与微服务之间的多中心架构，根据业务需求划分不同的区域，在每一个区域中根据自己应用的特点选择不同的技术。
11. SOA架构依然使用的是面向对象的方法，**现存SOA实现方法，大抵是基于远程对象来进行服务的封装和调用的**。

T11:24 <https://mp.weixin.qq.com/s?__biz=MzIwMzg1ODcwMw==&mid=2247486536&amp;idx=1&amp;sn=68423642a657b4ee36441cf09dc1cfe1&source=41#wechat_redirect>

1. 架构：一种平衡手段，架构为特定的目标服务，通过平衡资源的分配从而达到有限的资源下实现特定的目标。【明确取舍】
2. 
3. 现实生活中大多是分布式多中心的架构
4. 微服务中“轻量级”的缺点：
5. 安全性问题【不设防的微服务，对于黑客来说，从浏览器模拟一个服务请求就可发起攻击】
6. 系统间耦合问题【会引入事务一致性等原来不存在的问题】
7. 系统可靠性问题【当某一个模块出错，会引发大规模错误和回滚请求，能做的也就是将错误的模块隔离】
8. 需要一种架构能够支撑更复杂的混合需求。

T13:48 <http://www.infoq.com/cn/articles/service-should-be-to-release>

1. 服务必须在空间和时间上具有唯一性和稳定性

时空唯一性：当任意两个服务，如果其定义中引用的所有元数据集合完全一致的时候，无论这两个服务的结构上有什么差异，这两个服务都是同一个服务。

1. 服务的多版本：

*根据S++对服务内涵的定义，服务具有空间和时间上的一致性，所以服务本身是不需要版本的。服务的变迁是因为我们对服务的抽象过程的缺陷造成的，一旦最新的抽象形成了，那么这个抽象必须完全满足时间和空间上的所有外延实现。*

*这就是说，我们只需要保存最新的内涵抽象就够了，为了更好记录变迁，可以对服务的变成过程做一些批注记录。*

*在服务组合流程当中，当我们认定服务在空间和时间上的唯一性后，服务的组合中调用的原子服务就具备唯一性，一旦版本被引入后破坏了服务在时间上的唯一性，那么服务组合就要决定其调用的原子服务在不同时间点的副本，这本身和业务流程毫无关系，从而破坏了业务流程的完整性造成流程不稳定。*

1. 解决外围变更问题（时空稳定性）

S++服务的时空唯一性和稳定性可以通过接口的形式来体现，当服务的外围系统（包括消费者或提供者）发生改变时，比如升级或更换系统，那么在不修改系统内涵的前提下就可以通过定义新的服务接口来适应外围变更，从而减小甚至消除变更对整体带来的影响。

1. 小结
2. 服务具有时空唯一性和稳定性
3. 服务无需维护版本
4. 在不变更服务的前提下，对同一个服务可以有无数种不同的表达（接口）
5. 服务的变更不会影响已经存在的任何一个表达（接口）

T14:47 <http://p.primeton.com/articles/59030eeda6f2a40690f0361f>

1. 代理微服务或API网关：大型企业的前端可能需要调用数十个甚至数百个HTTP请求
2. 反向代理：微服务的API接口绝对不能直接暴露（可以使用代理，如Nginx、 Apache Tomcat）
3. 最小化方法：微服务应当只包含其真正需要的东西，例如程序包（package）、库和框架等。最小化的策略同样也适用于操作系统方面（OS）
4. 配置管理：随着微服务数量的不断增加，对配置管理（CM）的需求越来越多
5. 跨功能研发小组：虽然没有规则说明，但研发团队最好是在开发具有多功能特性的应用时采用微服务方法。
6. API版本控制（versioning）

版本控制应当适用于任何API，对微服务也一样。如果有某些改动打破了API的格式，那么就应当针对该改动单独发布另外一个版本。不论是公共接口还是其他内部服务使用的接口，我们不清楚谁在使用这些接口，因此必须要保证向下兼容，或者至少要给用户足够的时间去适应。

T15:12 <https://www.cnblogs.com/onetwo/p/6269384.html>

1. 背景：ZXNP-PICT项目是一个基于容器技术，采用微服务框架实现的智能ICT融合的PaaS系统；由许多微服务化的组件项目组成的，这些组件项目团队分布在上海、成都、西安、南京等地。
2. 软件仓库流程：
3. 测试通过的组件版本会推送到临时库
4. 整系统CI从临时库取各组件最新版本，测试通过后推送到可用库
5. 交付发布

（3.1）经过联调、版本验证之后，确定版本组件清单，对内发布

（3.2）交付对内测试，项目、质量审评会通过，将组件清单里的版本从可用库同步到发布库，对外发布。

3、需要考虑的问题：

（1）保证各组件版本对应代码的可追溯性

（2）开发人员能够清楚了解组件版本在软件仓库的状态信息

（3）软件仓库的远程化

（4）避免软件仓库出现集中爆发式访问导致出现性能负荷问题

（5）软件仓库的容灾备份

（6）各地用户能快速部署所需的版本

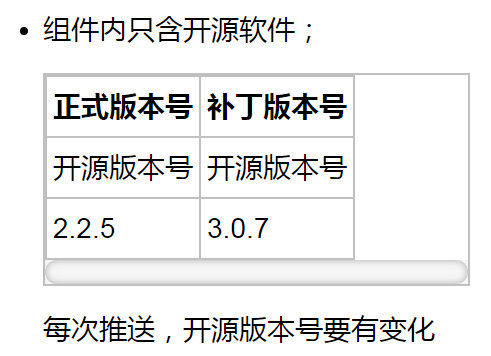
（7）软件仓库的权限控制

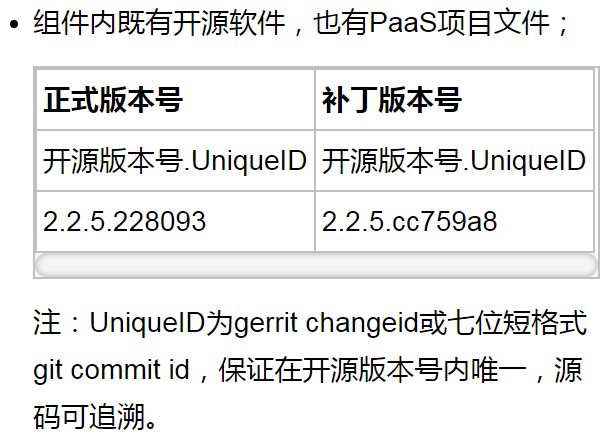
4、解决问题

（1）如果照搬老经验对代码库打标签，是不必要、不合适的，因为多组件之间的版本号并不统一，某些组件可能一个版本对应多个PaaS项目版本；某些可能是非PaaS项目下的，用的是自己项目的版本号。

（2）结合项目运作中遇到的问题，现已对推送组件的版本号做了一定的规范要求，这样通过组件版本号就能追溯到具体代码。

（3）每次推送组件的版本号要求都是唯一的，相同版本号上传会被仓库拒绝。





每次推送，如果开源软件有变更，则变化开源版本号；如果PaaS项目文件有变更，则变化UniqueID；如果两者都有变更，则同时变化开源版本号、UniqueID。



UniqueID为gerrit changeid或戚薇短格式git commit id，保证在PaaS项目版本号内唯一，源码可追溯。

每次推送，都需要变化UniqueID；如果切换到了下个版本迭代，则同时变化PaaS项目大版本号、UniqueID或者PaaS项目大版本号、pxx、UniqueID。



●四个使用情景的部署：整系统ci测试【临时库】、联调&版本验证【可用库】、对内发布【可用库】、对外发布【发布库】。

5、效果评价：

●组件版本变得更清晰直观，便于管理通过组件的版本号就能大致知道组件的来源归属，开源、本项目、其他项目等；已部署的PaaS系统，查看相关组件的版本号就能定位到具体代码；杜绝了以前开发人员用同一个版本号反复删除推送组件的情况，保证了组件版本来源的唯一性。

●制品库变得更加强壮，实现了远程化、自动化；镜像仓库的部署变得简单，均采用了远程化管理；随着各地镜像库数目的壮大，容灾备份点也同步增加了；异地仓库的组件版本同步，通过调用远程脚本，实现了统一维护，各地仓库只需调用即可，实现自动化

●支持多团队异地协作联调，实现节奏同步存在依赖关系的异地项目团队在提交各自组件版本入库后，可及时在本地获取到依赖对方的可用组件版本进行联调；

●快速部署，仓库共享；PaaS部署从原先的异地部署耗时半天，缩短到平均三十分钟左右就能完成一次；各地仓库组成了共享资源池，任何几处同时出现问题都不会影响到用户部署

T16:45 <https://blog.csdn.net/penker_zhao/article/details/51132514>

请求先发送到Load Balancer（负载均衡）上，通过LB上的路由算法（轮询或哈希等），将请求转发到后面具体的Web Server上，这类请求转发技术被称为Reverse Proxy（反向代理）

1. 交付流程

●设计阶段，架构师将产品功能拆分为若干微服务，为每个微服务设计API接口（例如REST API），需要给出API文档，包括API的名称、版本、请求参数、响应结果、错误代码等信息。在开发阶段，开发工程师去实现API接口，也包括完成API的单元测试工作，在此期间，前端工程师会并行开发WebUI部分，可根据API文档造出一些假数据（我们称为“mock数据”），这样一来，前端工程师就不用等待后端API全部开发完毕，才能开始自己的工作了。

●在测试阶段，前后端工程师分别将自己的代码部署到测试环境上，测试工程师将针对测试用例进行手工或自动化测试，随后产品经理将从产品功能上进行验收。在部署阶段，运维工程师将代码部署到预发环境，测试工程师再次进行一些冒烟测试，当不再发现任何问题时，经技术经理确认，运维工程师将代码部署到生产环境，这一系列的部署过程都需要做到自动化，才能提高工作效率。

2、开发模式：

●可以使用Git支持多分支代码版本管理，最稳定的代码放在master分支上（相当于SVN的trunk分支），出bug新建一个分支，修改后再推上去。

●master分出develop分支，在develop分支上分出feature分支，feature-1与feature-2等，决定发布时，在develop拉出一条release分支，测试完后，release合并到master和develop，并在master分支上打一个tag，例如v1.0.0。生产环境发现bug时，我们需要从对应的tag上拉出一条hotfix分支，并在该分支上做bug修复。

●版本号的要求：格式为：x. y. z，其中，x用于有重大重构时才会升级，y用于有的特性发布时才会升级，z用于修改了某个bug后才会升级。

3、微服务架构特点：

●粒度微小

●责任单一

●隔离性好

●管理容易

4、微服务架构带来的挑战：

●运维要求较高（部署—监控—保障系统高可用）

●分布式复杂性

●部署依赖性较强：部署可能有顺序性要求，业务上的依赖性，导致部署的依赖性。从而在某一时间点，同一微服务可能具备多个版本。

●通讯成本较高