配置和注册中心一般nacos就行。

业务中心，就是saas能够提供的具体软件服务。

同时需要区分几个环境：开发、测试、预上线、线上

基础服务

大体方向如上面步骤，我们来优化下细节。

个性化

saas系统之所以如此复杂，就是因为个性化。接入的供应商越多，需求越多样。

配置中心

第一个问题就是个性化配置，针对每个客户，某种功能，有的客户需要，有的不需要，每个客户展示内容也不相同。

所以需要有一个配置中心，而一个灵活的客户配置中心应该是怎样的呢?

首先，从作用域范围分类，分别是系统层级、客户层级、账号层级。

系统层级就是最顶层配置，基本放的都是默认值之类的，比如需要给某些客户增加特定功能，默认的系统层级配置为未生效，指定客户层级配置为生效，

这样就能区分出需要使用该功能的客户和不需要使用该功能的客户。然后是账号层级，为什么需要账号层级?

账号层级应该是最细粒度的层级，多数情况是一个客户可能在系统开通多个账号，比如子公司等业务，针对每个账号也需要不同的配置

另外，配置需要区分应用，每个应用的配置肯定是不一样的。

然后从传递方式可以分为向上传递或向下传递，所谓的向上向下传递，就是指三个层级相同的配置key同时配置时，应该取哪一层级的配置。

一般情况下都是取粒度更细的配置，当然，也不排除向上取。

举两个例子：

1.向上传递（取粒度更小的配置），比如有这么一个需求，系统可以配置个性化的logo，

大部分小客户（个体工商户、小门店等）可能没有这种需求，他们只想要一个可以操作的系统，没有心思去搞logo之类的，那么系统就需要给一个默认的图片。

根据客户类型，也需要配置不同的图片样式等。但是中大型客户有自己完善的信息，他们能自己配置logo图片等。

而针对这类客户，还可能存在子公司等情况，而子公司也会有自己的个性化配置，但子公司如果未配置会默认为上层公司的配置。

所以这种配置就需要向上传递，取细粒度最小的那个配置。

2.向下传递（取细粒度更大的配置），比如说业务中需要发送短信功能，会根据客户的付费情况分配每月能够发送的短信条数。

最底层的账号配置成100w。但它的上一层级为了控制资源，想要限制他的短信条数，给它配置为50w，那么发送短信时就需要获取层级最上面的那个配置50w去做限制。

好了，从这里的需求就能看出，非saas系统和saas系统在配置上的差异化。

而这里只是列举了三层结构，但真实的情况是，子公司下面还会有子公司，所以真实的业务配置是一个无限向下的树形结构。

那么配置中心的数据结构在saas系统的演化过程就是：

简单配置 -》 固定层级配置（两层、三层等）-》多层配置（无限延展的树形结构）

如果一上来就设计成最复杂的多层结构，应该怎么去定义我们的数据结构呢?

首先，我们需要暴露出一个接口，支持修改配置，需要的字段有组织机构id、配置的key、对应的值、传递方式（每个key可以写死）

新增和保存比较简单，只需要将上面的信息入库即可，但是获取配置信息的时候就会比较麻烦。

这里的查询需要结合组织机构树来查询，根据组织id、层级关系和传递方式获取对应的配置信息。

而组织机构树的结构一般记录的信息就是机构id、上层机构id两个信息。但是

，为了提高查询效率，也可以采用空间换时间的思路，冗余存储每层关系。

比如以下简单的一个组织机构：

! ql提供的拦截器，在执行sql时动态的将租户筛选条件带上。类比其他的中间件入mongodb、es、redis也是类似。

简单的数据隔离策略只适用于数据量比较小，且业务不是很复杂的场景，一般常见于saas系统开发初期，用的资源少、逻辑简单。

一般数据隔离

一般的数据隔离方式就是数据分表，业务表后缀带上分割标识，一般以租户code区分。数据库上同样可以利用拦截器，替换指定表表名执行sql。

当然也可以直接引入分库分表工具，如shardingJDBC等。分库分表工具不仅可以动态修改sql，也能更好的对表进行新增、修改等操作，手动运维有时候容易出错。

这种方式，就比较适合有一定数据规模的saas系统了。

较为完善的数据隔离

大部分saas系统可能在第二种数据隔离方式下就已经算是到头了，但也不乏一些比较大型的saas系统。

他们的数据隔离方式更加复杂，基本是分库 + 分表两种方式结合使用。

如何分库?

针对一个比较大的客户，他们的业务量可能是几十甚至上百家中小客户的总和，这种客户单纯的分表已经满足不了需求，一般就会分库。

多个中小客户分在一个库里，大客户单独成库。

当然，在不断前进的时间轴里，个别中小客户也会成为大客户，这时候如果我们有幸在这个变化中，就会经历分表到分库的数据迁移过程，以及服务迁移过程。

这时，光站在分库分表策略角度，你就会发现，分库分表策略将变得非常复杂，而且复杂的程度是呈指数式增长。

因为这种策略上，需要包含每个客户的分库逻辑和分表逻辑、单个大客户的分

表逻辑和分库逻辑（大客户就算分了库，也可能再分表）。不管是维护还是开发，都会有一定的难度。

个人认为比较好的处理方式是流量数据隔离，既然从分库分表策略上比较复杂，可以直接选择在流量接入点就做流量切分。

比如存在一个大客户和几十个小客户，在请求到达网关和内部feign调用时，动态的修改请求需要访问的应用服务，将相同应用分成两部分，

一部分应用叫大客户应用，一部分叫集群中小应用，提供的功能完全一样，但访问的节点永远隔离。

这样，不管哪个服务，只需要分表策略，分库策略在流量上就已经动态隔离。

同时，这样也带来以下一些优点：

1.资源细分，更方便做资源优化。大客户本来的并发和数据量就比较大，可以分配更多的资源给他们。

小客户可能需要的资源微乎其微，如果绑在一起，将节约不少成本。

2.内测，如果一次大版本更新，就算经历好几轮测试，估计也没人会笃定一定不会出现大的纰漏。

如果是在比较大的公司，一个小bug就可能会导致很大的损失。

这时候如果是上线一部分企业，流量和数据又是完全隔离的，即使出现问题也影响不到大部分其他客户。这样做能将损失尽量减少。

公共数据

说了数据隔离，一般的业务中也会有不需要数据隔离的部分。

比如登录的时候，在没登陆之前是不知道当前用户应该归属于哪个租户的。

针对这种情况，如果是数据，就会将这些部分单独抽离出，比如用户、短信、支付、审批等。

由此我们就看到了各种集合的中台或者领域模型。

其他

saas系统和一般的微服务应用最大的区别就是数据隔离、个性化配置以及对资源的掌控。

其他的什么高并发、链路跟踪、ELK、限流熔断等等都和一般的微服务没什么区别，这里就不详细说了。

原文链接: https://juejin.cn/post/7366967246374109203