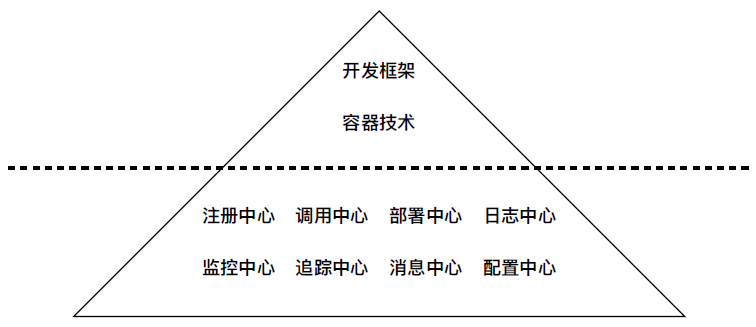
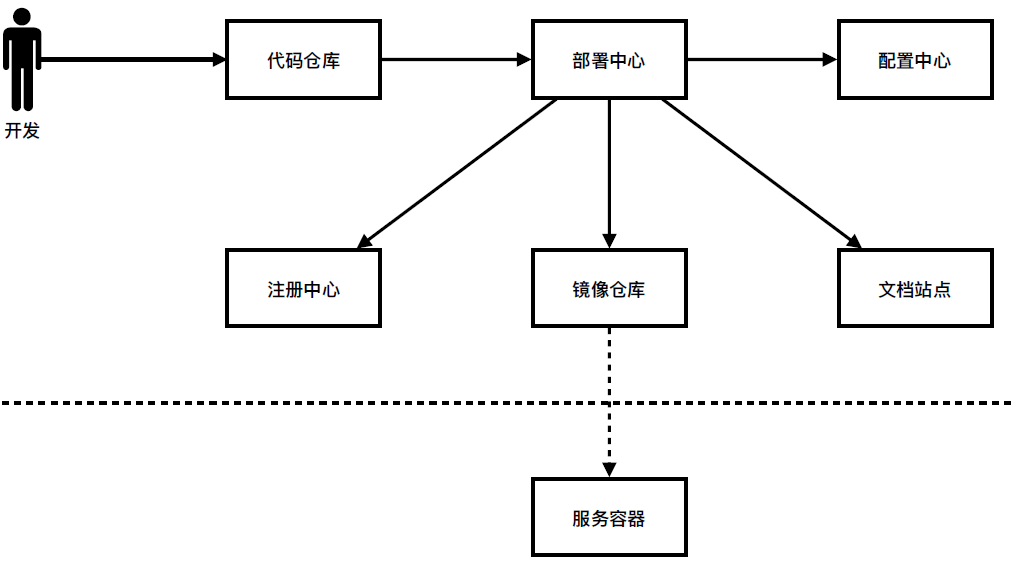
# 微服务架构冰山模型



1. 注册中心：用于注册微服务相关配置信息的中心， ZooKeeper 实现。
2. 调用中心：用于提供给前端调用的统一入口，Node.js 实现。 （Netty也可以）
3. 部署中心：用于编译并打包微服务源码并将其部署到 Docker 引擎中，我们选用 Jenkins 实现。
4. 日志中心：用于收集并管理微服务应用程序中产生的日志， Spring Boot日志框架
5. 监控中心：用于监控微服务的实时运行状况，Spring Boot监控系统。
6. 追踪中心：用于最终微服务的调用轨迹，开源调用追踪中心：Zipkin。
7. 消息中心：用于解耦微服务之间的调用关系，RabbitMQ Kafka。
8. 配置中心：用于管理微服务应用程序所需的配置参数，Ansible。
9. 开发框架为SpringBoot 容器技术为 Docker

# 微服务部署架构



当开发人员完成了微服务的细节实现后，首先要做的是确保自己所写代码的可用性，他们往往会借助单元测试工具来保证这一环节不出问题。当他们将源码提交并推送到代码仓库后，此时**部署中心**将从代码仓库中获取源码，并执行编译与打包操作。

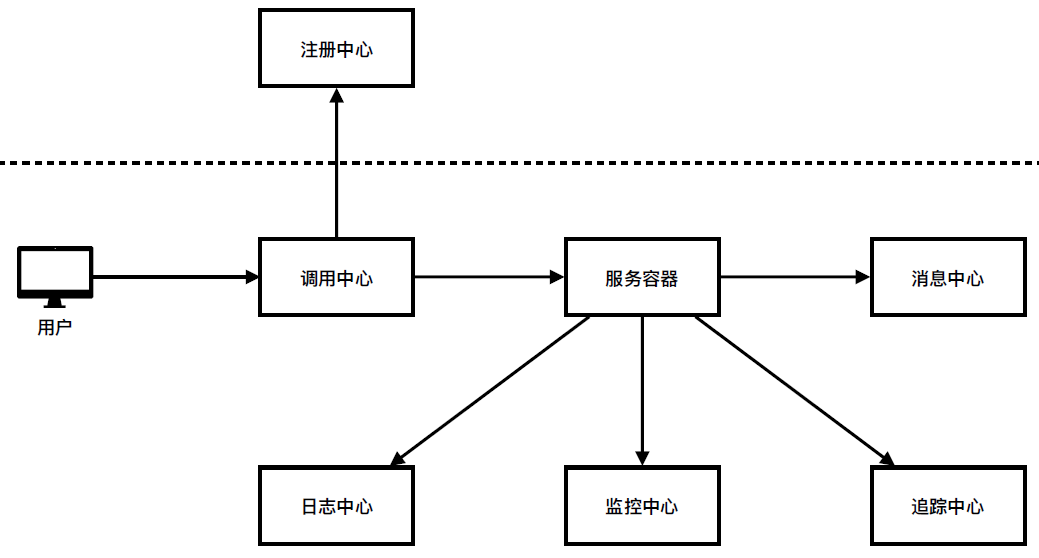
不仅如此，**部署中心**还需从配置中心获取对应运行环境的配置参数，并生成相应的配置文件，并将这些配置文件与应用程序一同复制到 Docker 镜像中，最后还需将此镜像上传到镜像仓库，以便后续可从镜像仓库中下载指定的镜像，从而运行相应的 Docker 容器。

此外，部署中心还可扫描源码并自动生成 API **文档站点**，以便其他技术人员可随时从文档站点中查看最新部署服务所包含的 API 文档。当然，我们还可以对所需完成的微服务仅提供简单的代码实现，这样就能将微服务文档的输出时间尽可能提前，以便其他技术人员能够更早地了解到微服务的相关 API 信息，随后再去完成更加细节的代码实现，这是我们推荐的工作方法。

当 Docker 镜像上传到镜像仓库后，部署中心可在不同的运行环境下根据特定的镜像来启动相应的 Docker 容器。为了便于描述，我们将该容器称为 “**服务容器**”，它包含了承载微服务的应用程序及其配置文件。

当服务容器启动后，会自动将其配置信息写入**注册中心**，与此同时，部署中心也会连接到注册中心，并设置服务的版本号，以便于在后续调用服务时可根据版本号来识别当前可用的服务。

# 微服务运行架构



当用户通过浏览器或移动端访问应用系统时，请求首先将进入服务网关，因为它是所有请求调用的中心，我们也将其称为 “调用中心”。它虽然是不带任何业务的中心，但我们需要确保它所做的事情足够少，让它不会成为整个应用系统的调用瓶颈。

随后**调用中心**将连接**注册中心**，并通过服务名称从注册中心中获取服务所在的 IP 地址与端口号（即服务地址），该过程称为 **“服务发现”**，进而调用中心可根据服务地址以反向代理的方式来调用具体的服务容器，该过程称为 “服务调用”。

在服务容器中可能会触发一些事件，这些事件将以消息的方式写入**消息中心**，以便其他服务可监听消息中心并从中获取相应的消息。该方案可解决服务之间的耦合问题，同时能将同步调用转为异步调用，提高整个应用系统的吞吐率。

在服务容器运行时会产生大量的日志，我们可将这些日志统一写入日志中心，并能在**日志中心**所提供的控制台上查询具体的日志信息。此外，日志中心也能帮助我们快速地定位并分析系统出现的异常状况。

为了观察服务容器是否运行正常，我们可借助**监控中心**所输出的图形化数据来判断。监控中心将不断地收集服务容器中的运行状态，包括 CPU、内存、硬盘、网络，以及应用程序的 JVM 内存使用情况。

由于微服务很难切得干净，服务之间难免会出现少量的调用关系，我们可将每次调用所产生的相关信息写入**追踪中心**，并通过追踪中心提供的图形化界面来查看服务之间的调用轨迹以及所产生的调用延时，从而可分析出服务调用所产生的性能瓶颈。

# 四、总体架构图

