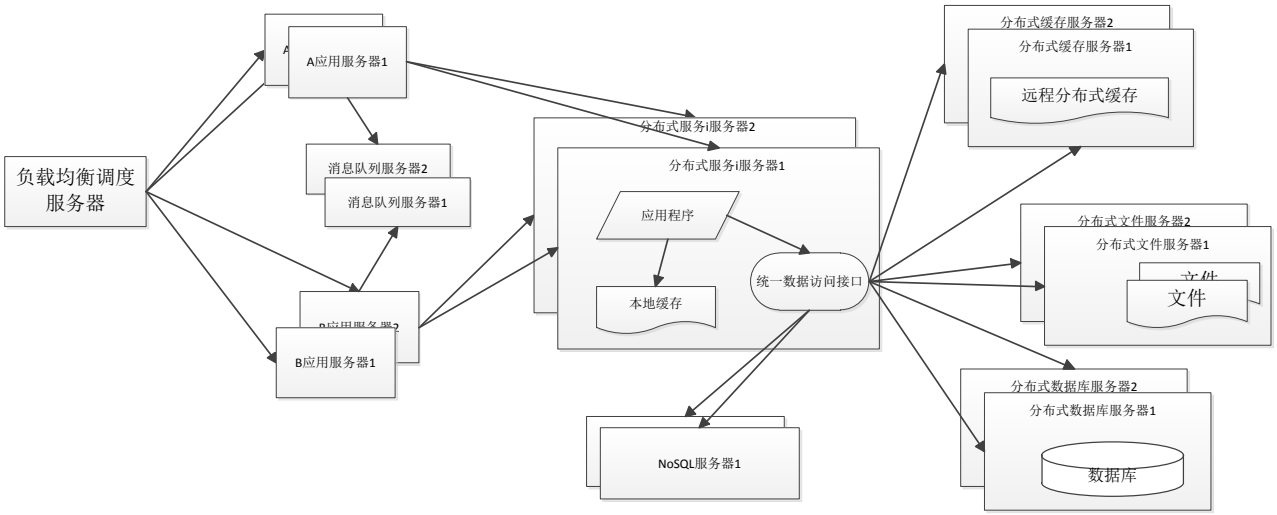
问题： 在分布式环境下如何监控服务提供者提供的服务调用情况（QPS）？

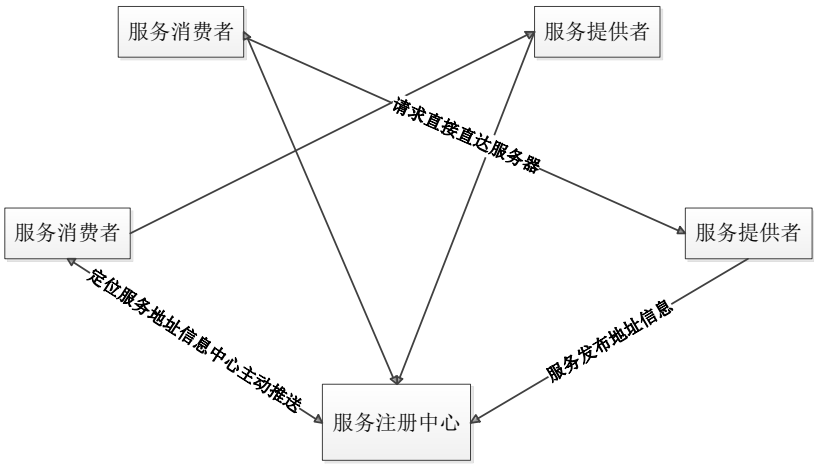
背景： 为了应对日益复杂的业务场景，通常将整个系统业务拆分成不同的子系统或者子应用，子应用之间通过消息队列进行通信，当然也可以通过访问同一个数据存储系统来构成一个关联的完整系统。在横向拆分上，将可以复用的业务子应用再拆分出来，独立部署为分布式服务，新增业务只需要调用这些分布式服务就可以完成自身内部逻辑。但是，随着服务数量的不断增多，如何有效实时监控服务调用情况成为一个必须要解决的难题。



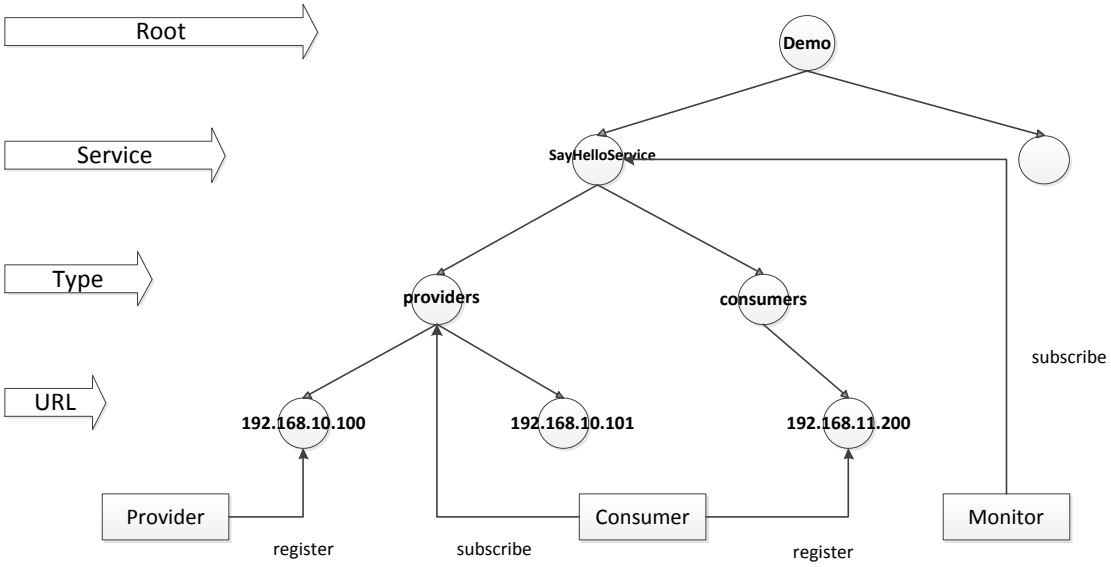
图一 分布式系统架构

总体思路： 首先在 ZooKeeper 集群中注册服务调用者和服务提供者信息以及服务监控服务，当服务调用者调用了需要监控的服务时，通过动态代理方式动态生成服务实现者的代理对象，其中代理对象中注入一个监听服务接口引用实例对象，通过回调机制起到对调用者调用某服务接口的拦截作用。而监听服务接口的实现类就是通过拦截真实服务调用无侵入式的给服务接口的调用情况进行统计和分析。下面详细介绍需要实现和注意的技术要点：

技术要点一：ZooKeeper 注册



图二 服务框架整体架构



图三 ZooKeeper 注册中心

服务注册和订阅流程：

1) 服务提供者启动时，向/Demo/SayHelloService/providers 目录下注册自己的 URL 地址；
2) 服务消费者启动时，订阅/Demo/SayHelloService/providers 目录下的提供者 URL 地址，并且向/Demo/SayHelloService/consumers 目录下注册自己的 URL 地址；
3) 监控中心启动时，订阅/Demo/SayHelloService 目录下的所有提供者和消费者 URL 地址。

技术要点二：动态代理

代理分为静态代理和动态代理，静态代理是在程序运行前代理类的.class 文件就已经存在，而动态代理在程序运行时运用反射机制动态创建而成，代理类和委托类的关系是在程序运行时确定，因此动态代理的灵活性更大。

代码片段一

```
/*暴露的服务接口*/
public interface SayHelloService {
    public void sayHello();
}
```

代码片段二

```
/*服务提供者实现类*/
Class SayHelloServiceImpl implements SayHelloService {
    public void sayHello() {
        System.out.println(“hello world”);
    }
}
```

代码片段三

```
/*功能增强的拦截服务监视类*/
public Class SayHelloServiceMonitor implements InvocationHandler {
     /*目标对象*/
     private Object target;
     /*构造方法*/
     public SayHelloServiceMonitor (Object target) {
```

```
        super();
        this.target = target;
    }
    /*拦截目标对象的服务接口方法调用*/
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

        1. 获取调用者的上下文信息
        2. 并发计数加一
        3. 记录起始时间戳
        4. 执行实际的调用 Object result = method.invoke(target, args);
        5. 采集调用信息比如计算调用耗时、获取并发数、获取实际服务名称和服务方法
        6. 并发计数减一
        7. 将收集到的信息通过任务交给监控中心处理
         return result;
    }
   /*获取目标对象的代理对象*/
  public Object getProxy() {
      return Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),
          target.getClass().getInterfaces(), this);
    }
}
```

技术要点三：消息信息格式

客户端代理在发起调用前需要对调用信息进行编码，需要考虑编码什么信息并以什么格式传输到服务端才能让服务端完成调用，出于效率考虑，编码的信息越少越好。

- 客户端请求消息

包括(1)接口名称(2)方法名(3)参数类型&参数值(4)超时时间(5)requestID 标识唯一请求 id

- 服务端响应消息

包括(1)返回值(2)状态 code(3)requestID

技术要点四：消息的编码和解码

消息格式确定之后，需要考虑将数据转化成二进制形式在网络环境下进行传输。

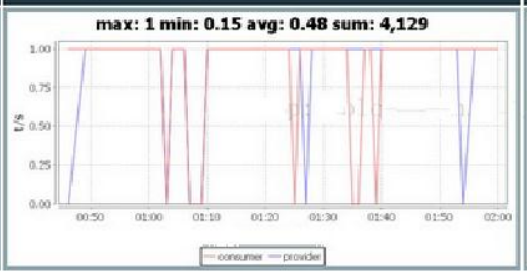
- 序列化：将数据结构或对象转换成二进制串的过程，也就是编码的过程。
- 反序列化：将在序列化过程中所生成的二进制串转换成数据结构或者对象的过程。

目前广泛使用 Protobuf、Thrift、Avro 等成熟的序列化解决方案，这些都是久经考验的解决方案。

技术要点五：网络通信

消息数据结构被序列化为二进制串后需要进行网络通信，而目前有两种常用 IO 通信模型：（1）BIO 阻塞式（2）NIO 多路复用，一般通过基于 mina 或者 netty 通信框架来实现通信。

结果： 展示监控某服务调用情况

Charts		
Method	Requests per second (QPS)	Average response time (ms)
sayHello	max: 1 min: 0.15 avg: 0.48 sum: 4,129 	max: 6,854 avg: 46.12