Say Hello、Say Goodbye，

一次RPC调用之旅

架构演进

单一应用架构　　垂直应用架构　　分布式服务架构

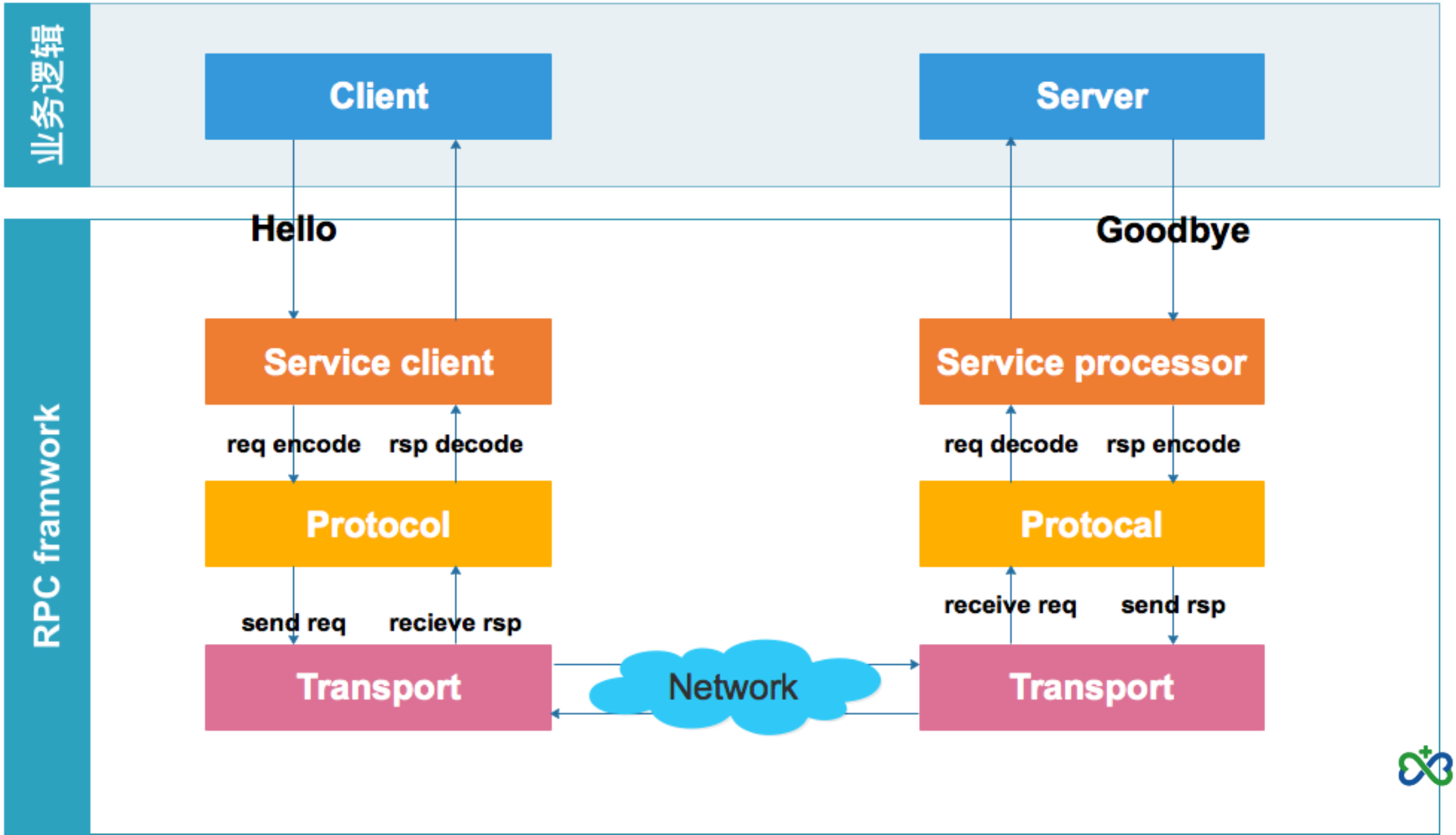
RPC (Remote Procedure Call)

让构建分布式计算（应用）更容易，
在提供强大的远程调用能力时不损失本地调用的语义简洁性。
（就像调用本地方法一样调用远程服务——调用机制透明）

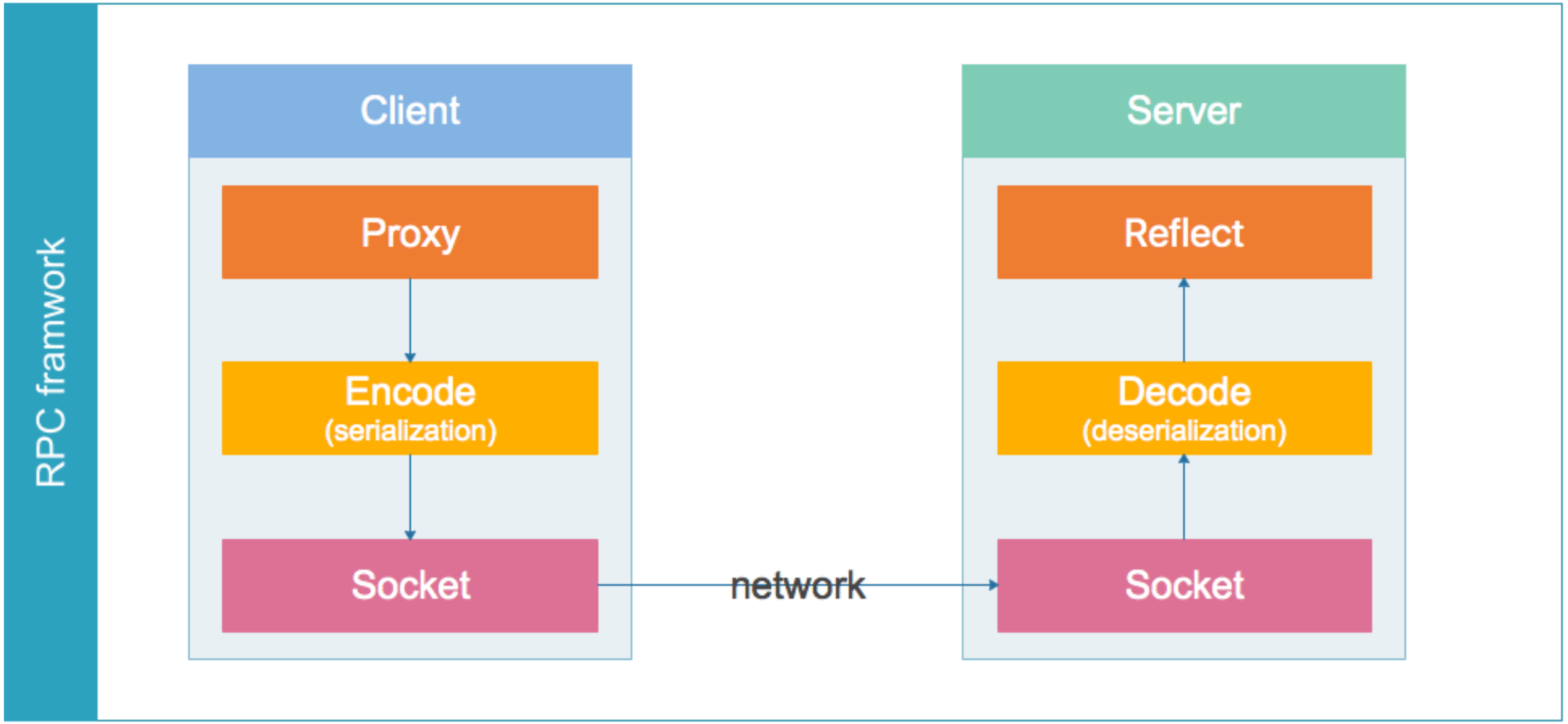
目录

一、玩具级RPC框架实现

1.1 RPC 调用流程


业务逻辑
Client
Server
Hello
Goodbye
RPC framwork
Service client
Service processor
req encode
rsp decode
req decode
rsp encode
Protocol
Protocal
send req
recieve rsp
receive req
send rsp
Transport
Network
Transport

1.2 搭建一个RPC框架



1.2 搭建一个RPC框架——ServiceClient

```
public class ServiceClient {
  public static class ServiceProxy implements InvocationHandler {
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
              //1. 构造请求参数
              model = new Protocol.ProtocolModel();
            model.setClazz(clazz.getName());
            model.setMethod(method.getName());
            model.setArgs(args);
            for (int i = 0; i < argType.length; i++) {
                argType[i] = method.getParameterTypes()[i].getName();
            }
            model.setArgTypes(argType);

            // 2. 编码(序列化)
              Protocol.protocol.encode(model);
            // 3. 方法调用
              rsp = ClientRemoter.client.getDataRemote(req);
            // 4. 解码(反序列化)
              Protocol.protocol.decode(rsp, method.getReturnType());
        }
    }
}
```

1.2 搭建一个RPC框架——Protocol

```
public class Protocol {
    public byte[] encode(Object o) {
        return JSON.toJSONBytes(o);
    }

    public <T> T decode(byte[] data, Class<T> clazz) {
        return JSON.parseObject(data, clazz);
    }

    public static class ProtocolModel {
        private String clazz;
        private String method;
        private String[] argTypes;
        private Object[] args;
    }
}
```

1.2 搭建一个RPC框架——ClientRemoter

```
public class ClientRemoter {
    public byte[] getDataRemote(byte[] requestData) {

            // 1. 建立网络连接
              socket = new Socket()
            socket.connect(new InetSocketAddress("127.0.0.1", 9999));
            // 2. 发送请求
            socket.getOutputStream().write(requestData);
            socket.getOutputStream().flush();
            // 3. 获取响应
            byte[] data = new byte[10240];
            int len = socket.getInputStream().read(data);

            return Arrays.copyOfRange(data, 0, len);
    }
}
```

1.2 搭建一个RPC框架——ServerRemoter

```
public class ServerRemoter {
  private static final ExecutorService executor = Executors.newFixedThreadPool(…);
  public void startServer(int port) throws Exception {
    // 1. 启动并接收socket请求
    socket = server.accept(); executor.execute(new MyRunnable(socket));
  }
  class MyRunnable implements Runnable {
    public void run() {
      // 2. 获取输入、输出流，用于读取请求，写入返回值
      is = socket.getInputStream(); os = socket.getOutputStream();
      // 3. 解码(反序列化)
      model = Protocol.protocol.decode(data, Protocol.ProtocolModel.class);
      // 4. 调用服务端处理器
      object = ServiceProcessor.processor.process(model);
      // 5. 编码(序列化) 返回客户端
      os.write(Protocol.protocol.encode(object));
    }
  }
}
```

1.2 搭建一个RPC框架——ServiceProcessor

```
public class ServiceProcessor {
    private static final ConcurrentMap<String, Object> PROCESSOR_INSTANCE_MAP
      = new ConcurrentHashMap<String, Object>();
    // 1. 发布服务的方法
    public boolean publish(Class clazz, Object obj) {
        return PROCESSOR_INSTANCE_MAP.putIfAbsent(clazz.getName(), obj) != null;
    }
    // 2. 处理器(反射调用)
    public Object process(Protocol.ProtocolModel model) {
            Class clazz = Class.forName(model.getClazz());
            Class[] types = new Class[model.getArgTypes().length];
            for (int i = 0; i < types.length; i++) {
                types[i] = Class.forName(model.getArgTypes()[i]);
            }
            Method method = clazz.getMethod(model.getMethod(), types);
            Object obj = PROCESSOR_INSTANCE_MAP.get(model.getClazz());

            return method.invoke(obj, model.getArgs());
    }
}
```

1.2 搭建一个RPC框架——服务

```
public interface RpcService {
    String say(String content);
}

public class RpcServiceImpl implements RpcService {
    public String say(String content) {
        System.out.println("receive: " + content);
        return "Goodbye";
    }
}
```

1.2 搭建一个RPC框架——Server

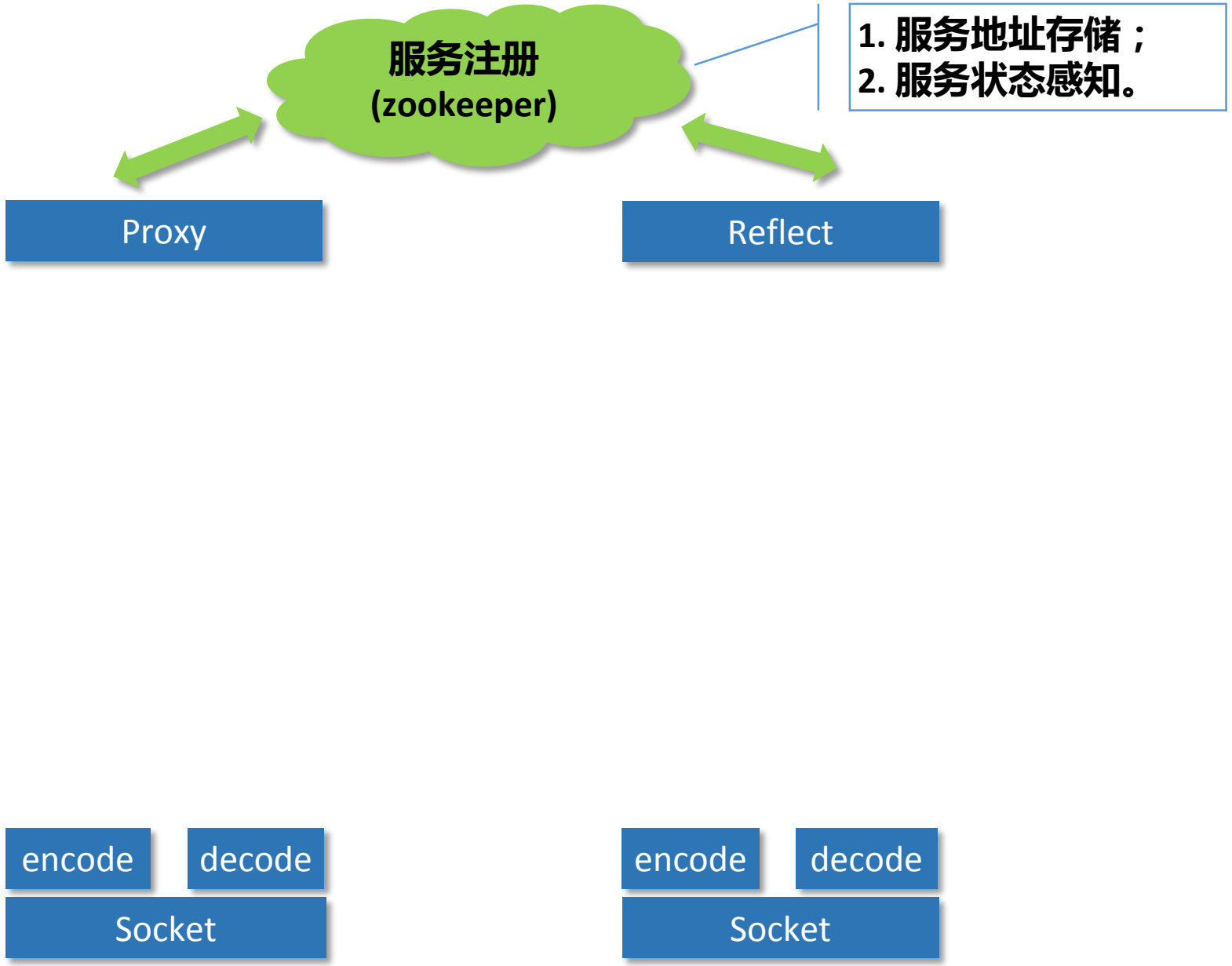
```
public class Server {

    public static void main(String[] args) throws Exception {
        // 发布服务
        ServiceProcessor.processor.publish(RpcService.class,new RpcServiceImpl());

        // 启动server
        ServerRemoter remoter = new ServerRemoter();
        remoter.startServer(9999);
    }
}
```

1.2 搭建一个RPC框架——Client

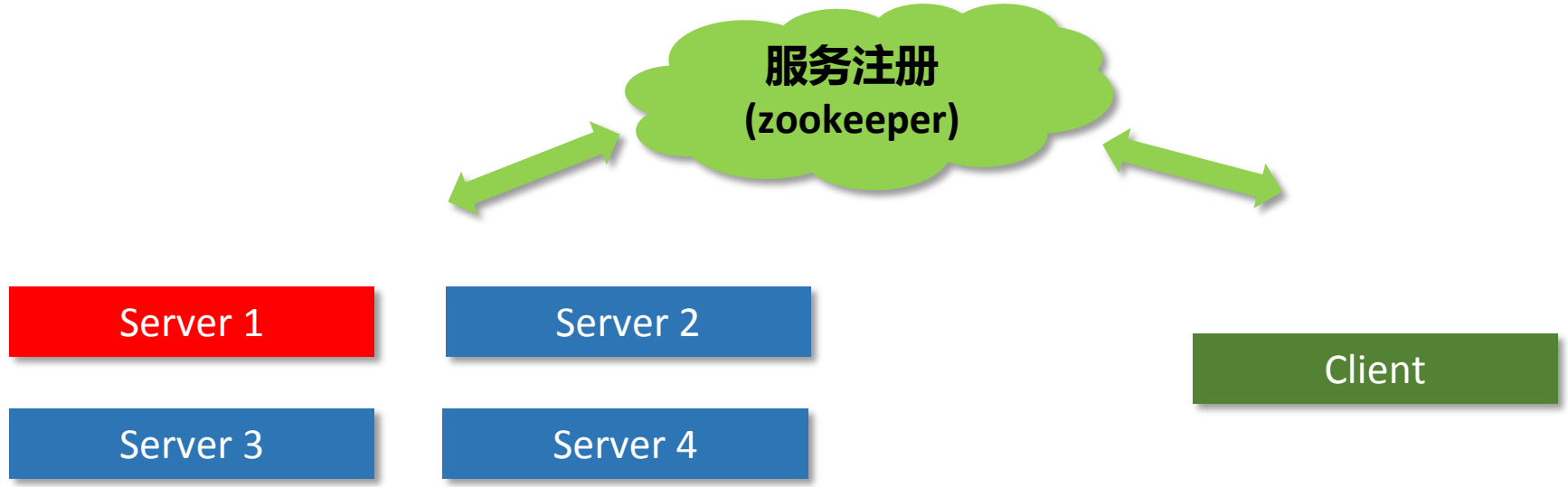
```
public class Client {

    public static void main(String[] args) {
        System.out.println("----------start invoke----------------");
        RpcService service = ServiceClient.getInstance(RpcService.class);
        System.out.println(service.say("Hello"));
        System.out.println("----------end invoke----------------");
    }
}
```

二、企业级RPC要解决的问题

2.1 服务发现


服务注册
(zookeeper)
1. 服务地址存储；
2. 服务状态感知。
Proxy
Reflect
encode
decode
Socket
encode
decode
Socket

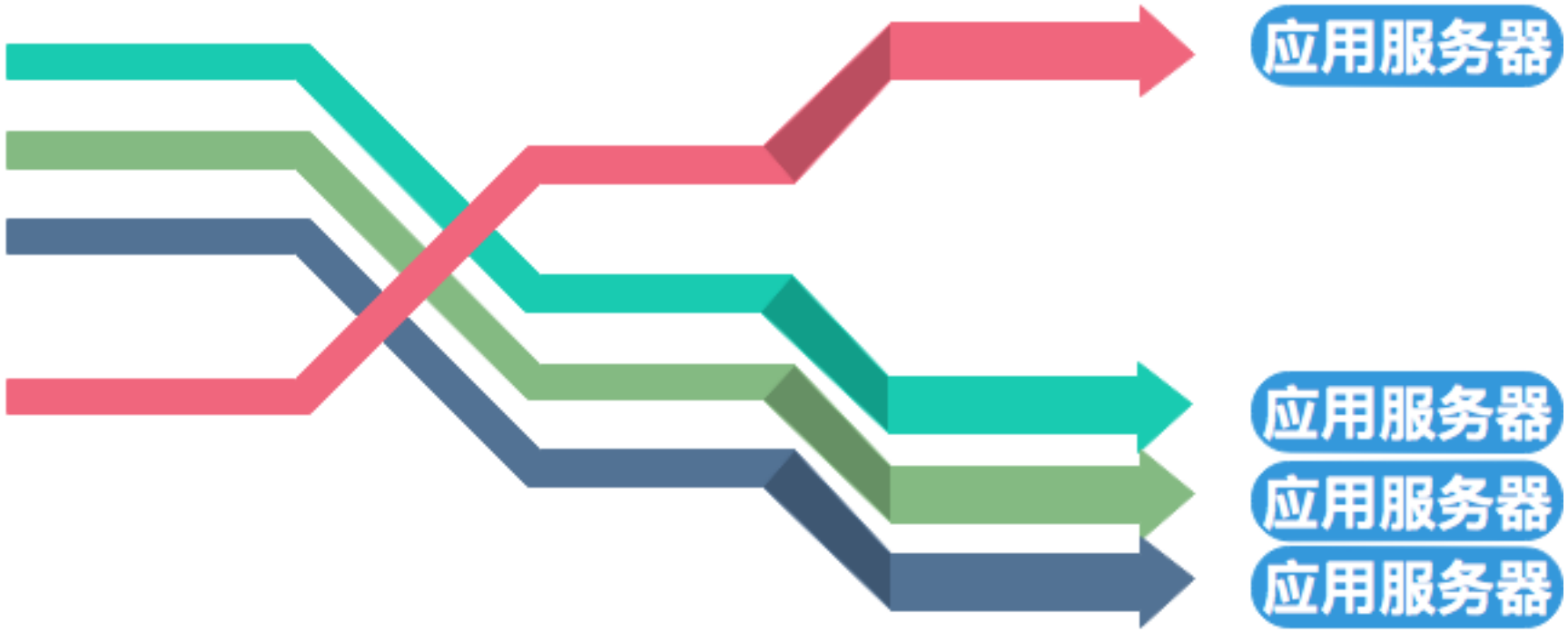
2.1 服务发现



2.1 服务发现

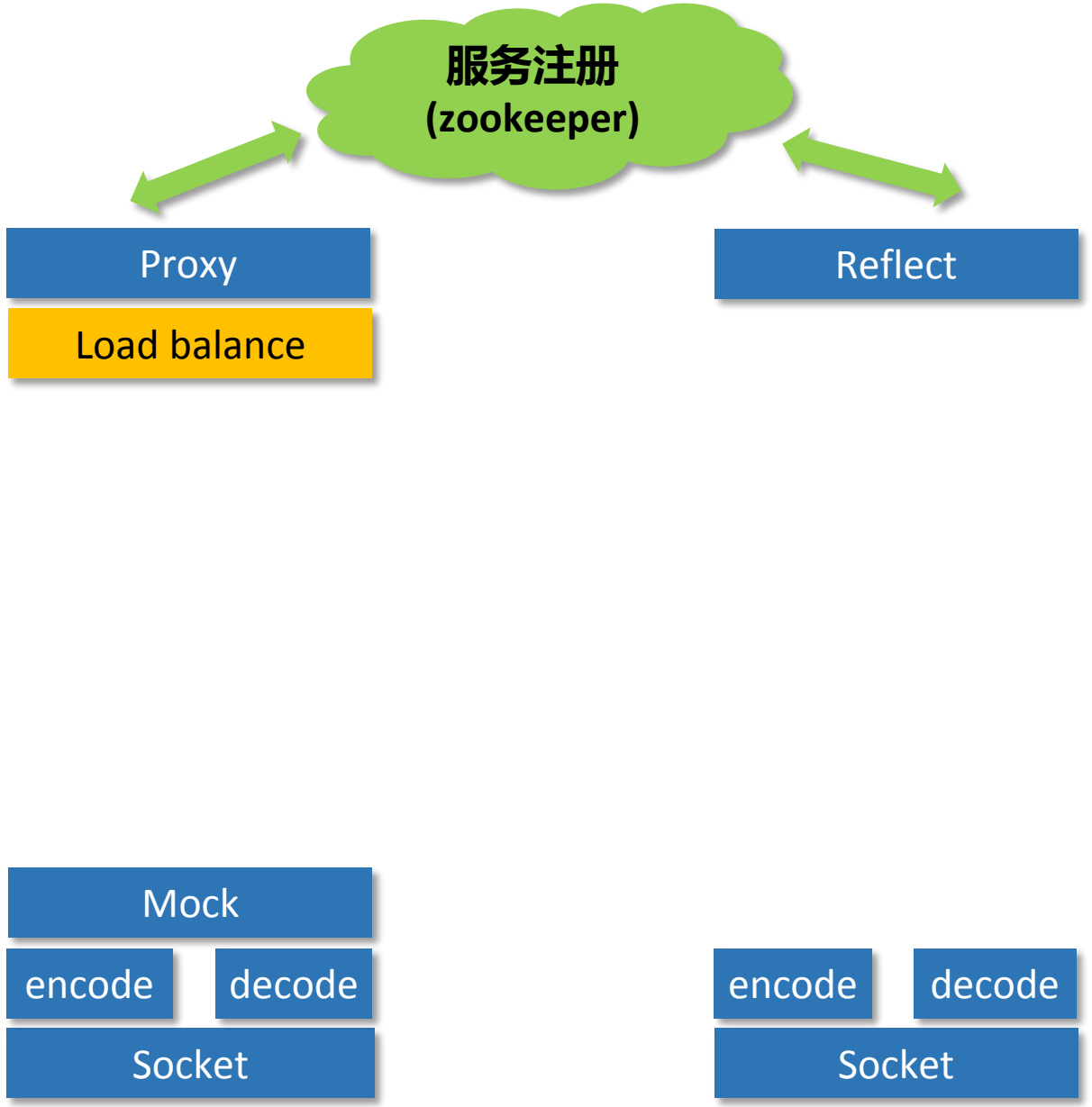
```
<!-- application 当前应用名称，用于注册中心计算应用间依赖关系 -->
<!-- owner 应用负责人 -->
<dubbo:application name="…" owner="…"/>

<!-- address 注册中心服务器地址 ——>
<!-- zookeeper://ip:port,ip:port,ip:port —>
<dubbo:registry address="…" />
```

2.2 负载均衡



2.2 负载均衡

随机(Random)
轮询(RoundRobin)
最少活跃(LeastActive)
一致性Hash(ConsistentHash)



2.2 负载均衡

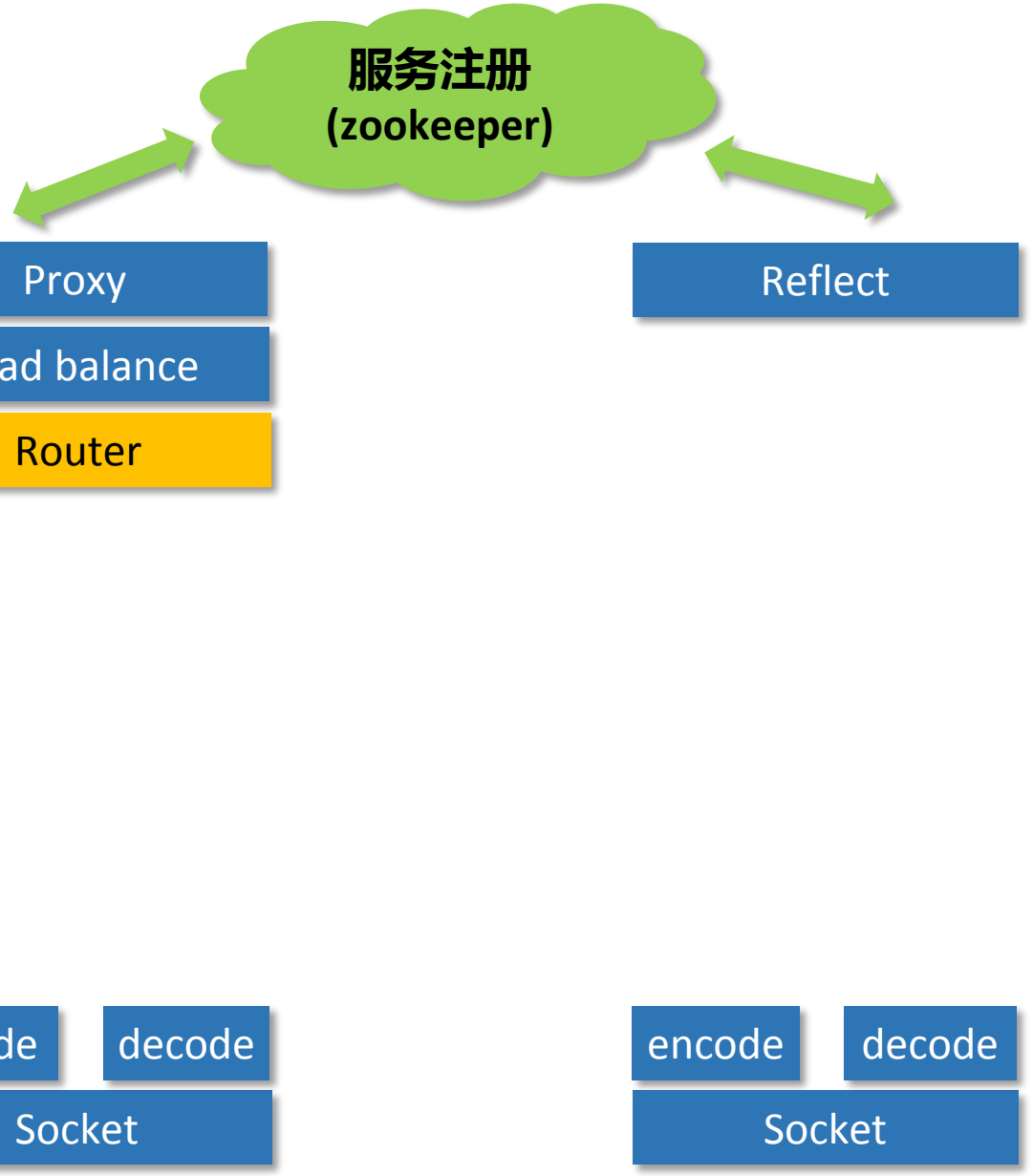
```
<!-- loadbalance 负载均衡策略 -->
<!-- random,随机(默认) -->
<!-- roundrobin,轮循 -->
<!-- leastactive,最少活跃调用 -->
<!-- consistenthash,一致性hash -->
<dubbo:provider loadbalance=“random” />

<dubbo:service interface=“…Service” ref=“…Service”
  loadbalance=”random”/>
```

2.2.1 路由

负载均衡策略是基于整个集群中所有机器的普适策略
路由策略则是单个机器根据自身特点做出的“服务方选择”策略

场景：单元化——优先调用本机房的服务

2.2.1 路由

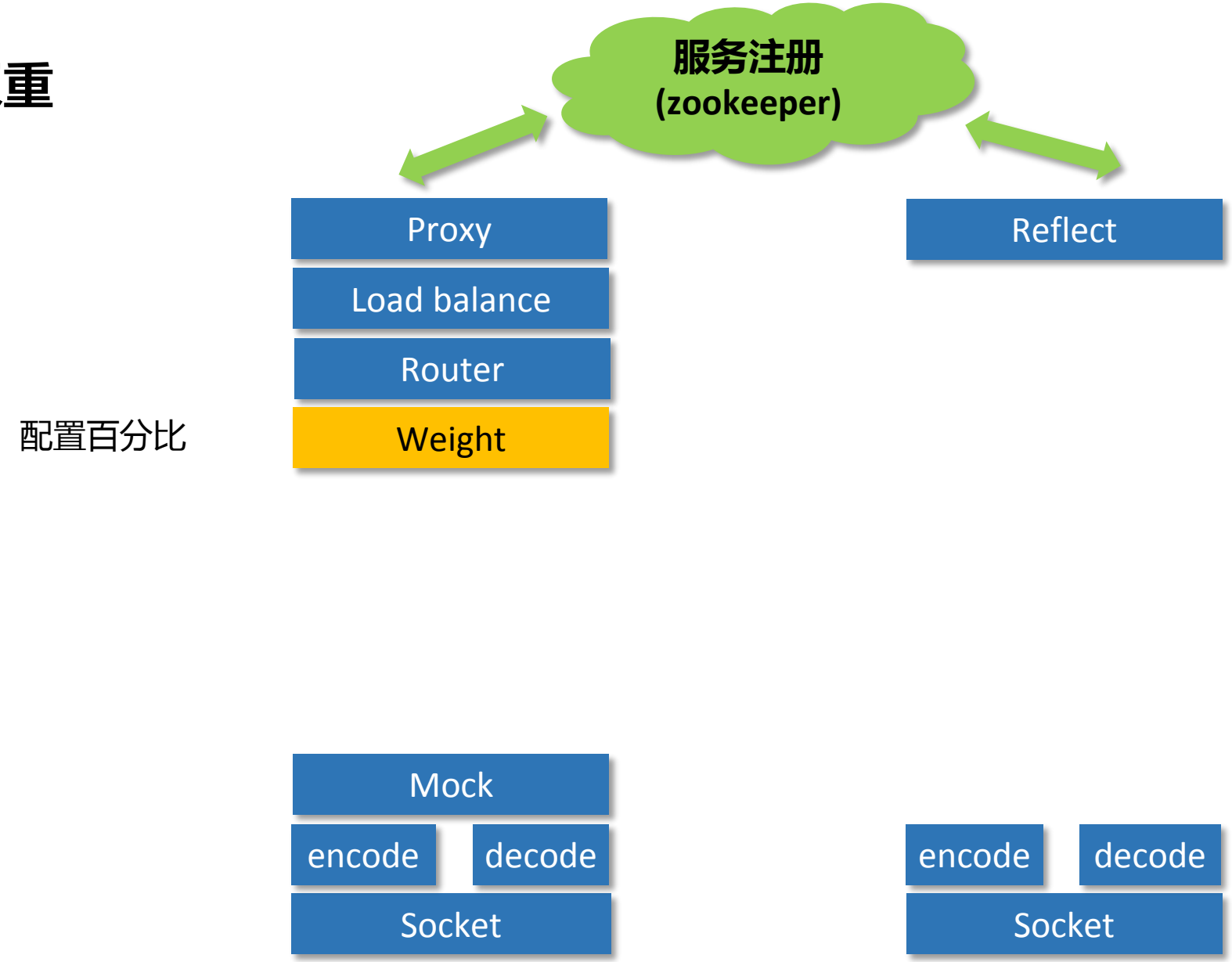


环境隔离(机房、预发)
黑白名单

2.2.2 权重

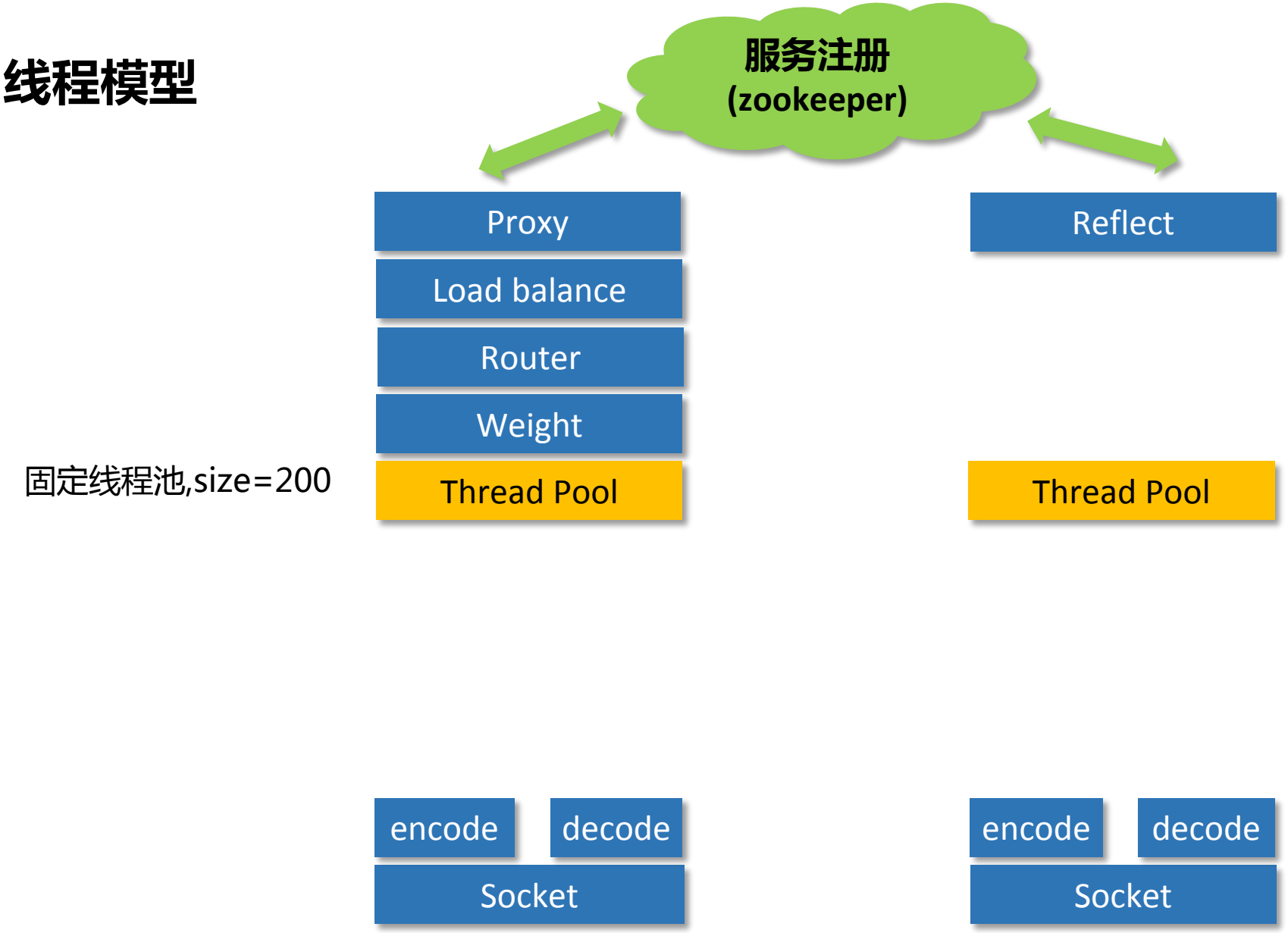
应用集群中机器性能差异很大

2.2.2 权重


服务注册
(zookeeper)
Proxy
Load balance
Router
Weight
Reflect
Mock
encode
decode
Socket
encode
decode
Socket

配置百分比

2.3 线程模型



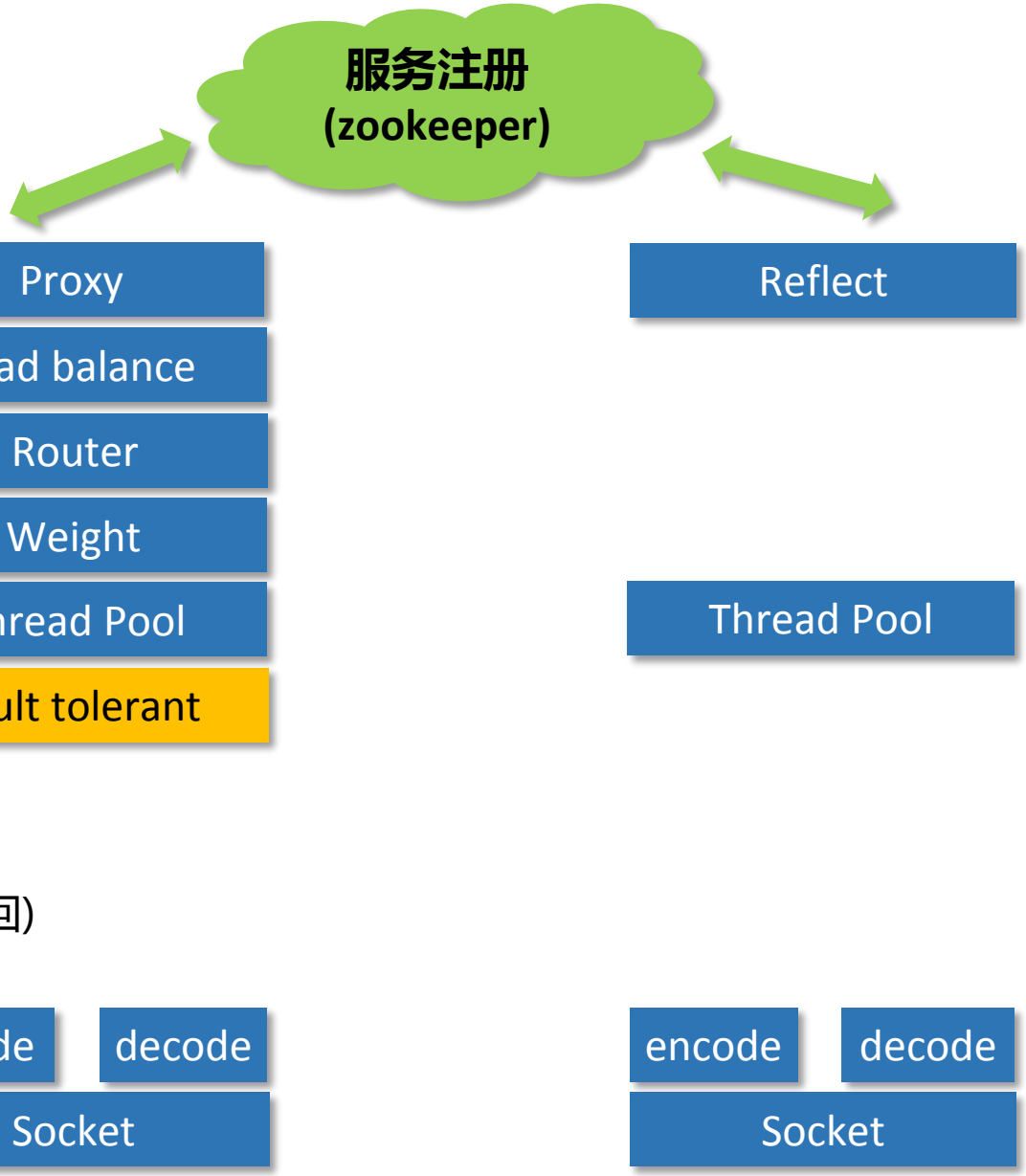
固定线程池,size=200

2.3 线程模型

- fixed 固定大小线程池，启动时建立线程，不关闭，一直持有。(缺省)
- cached 缓存线程池，空闲一分钟自动删除，需要时重建。
- limited 可伸缩线程池，但池中的线程数只会增长不会收缩。(为避免收缩时突然来了大流量引起的性能问题)。

```
<dubbo:protocol name="dubbo" threadpool="fixed" threads="200" />
```

2.4 集群容错

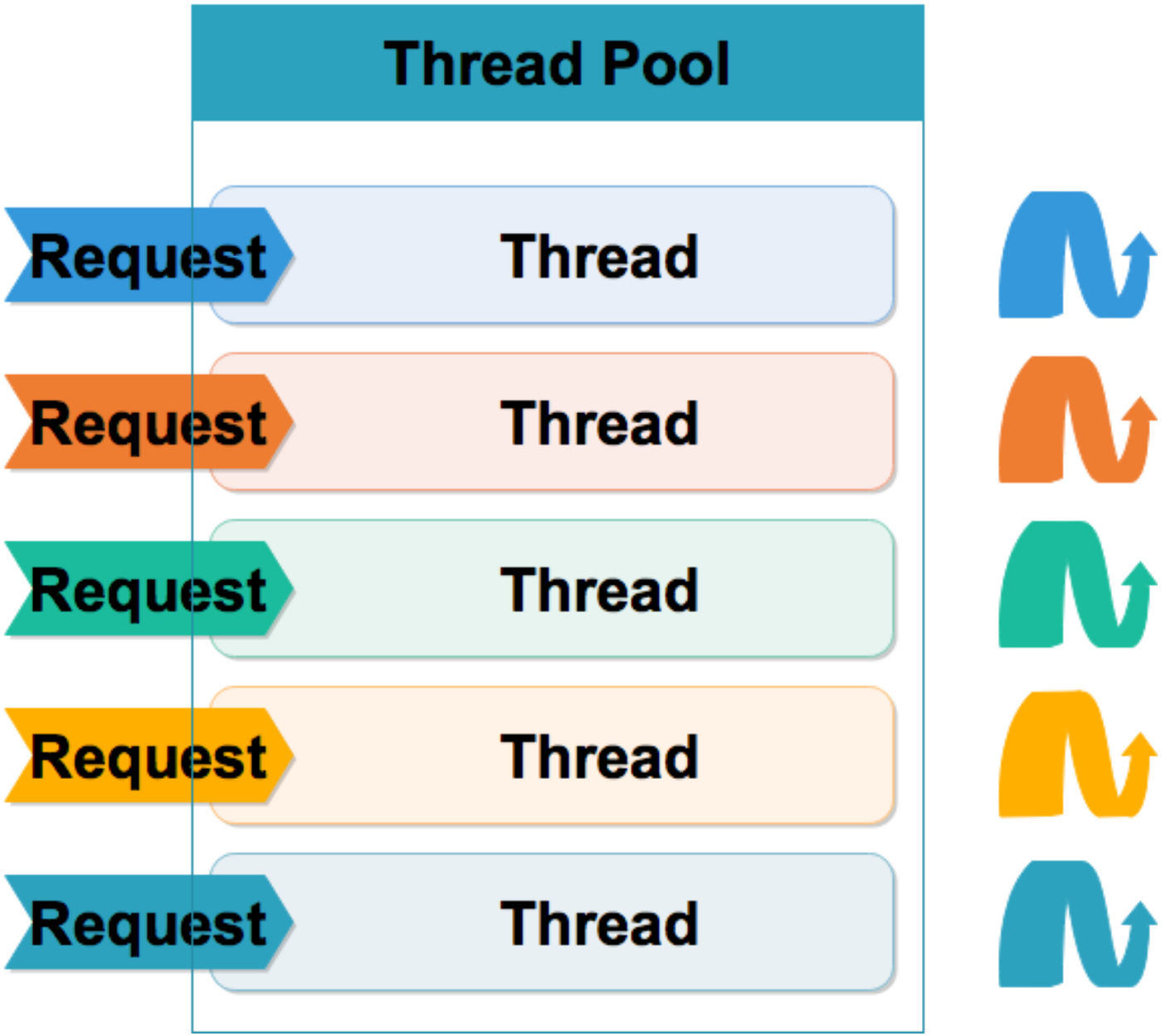


Failover(自动切换,失败后重试)
Failfast(快速失败,只发起一次调用)
Failsafe(失败安全,出现异常时,直接忽略)
Failback(失败自动恢复,定时重发)
Forking(并行调用多个服务器,只要一个成功即返回)
Broadcast(广播调用所有提供者,任一报错则报错)

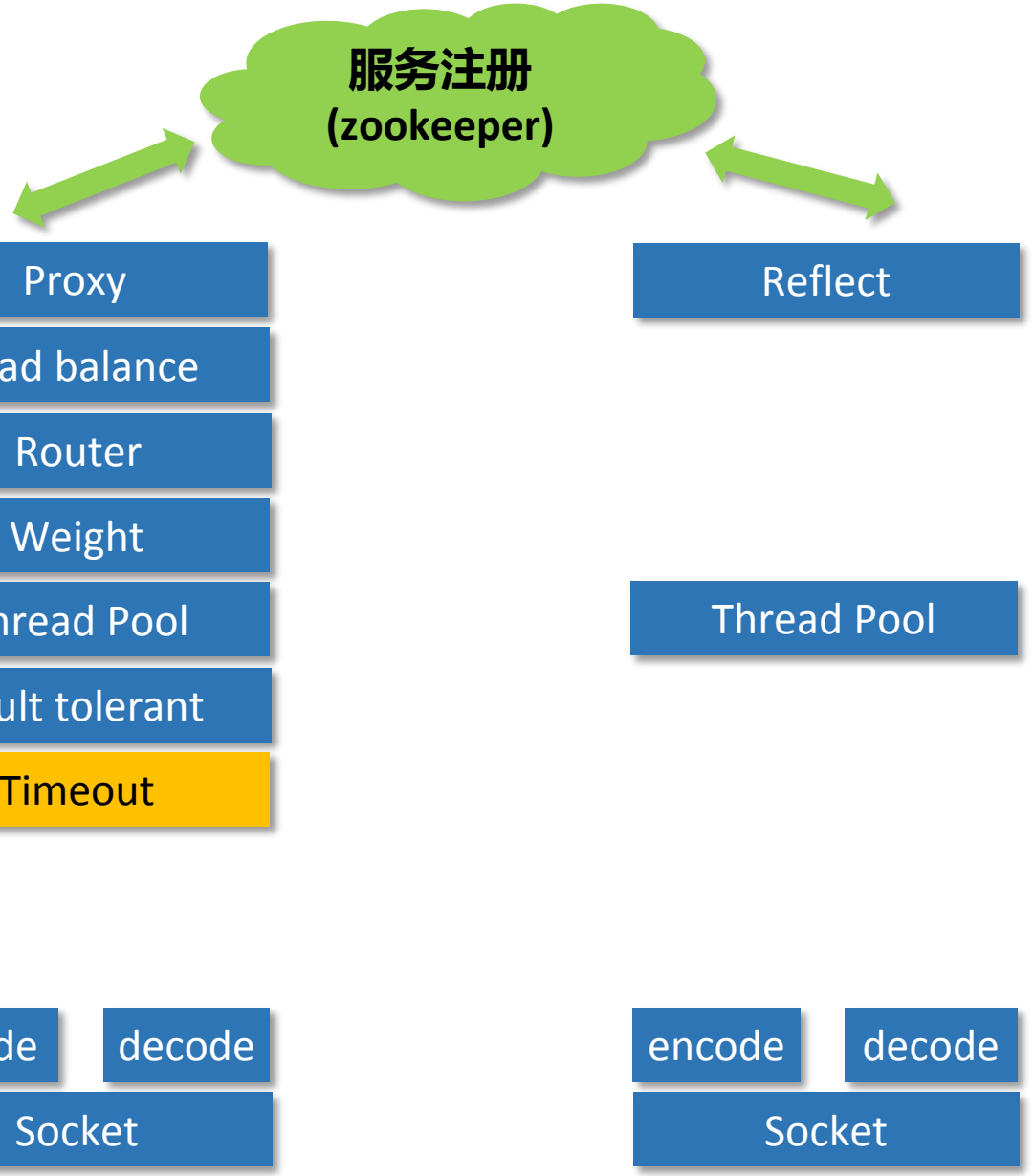
2.4 集群容错

```
<!-- cluster 集群容错策略。默认：failover -->
<dubbo:provider cluster=“failover” />

<dubbo:service cluster=“failover” />
```

2.5 超时管理


Thread Pool
Request
Thread
Request
Thread
Request
Thread
Request
Thread
Request
Thread

2.5 超时管理

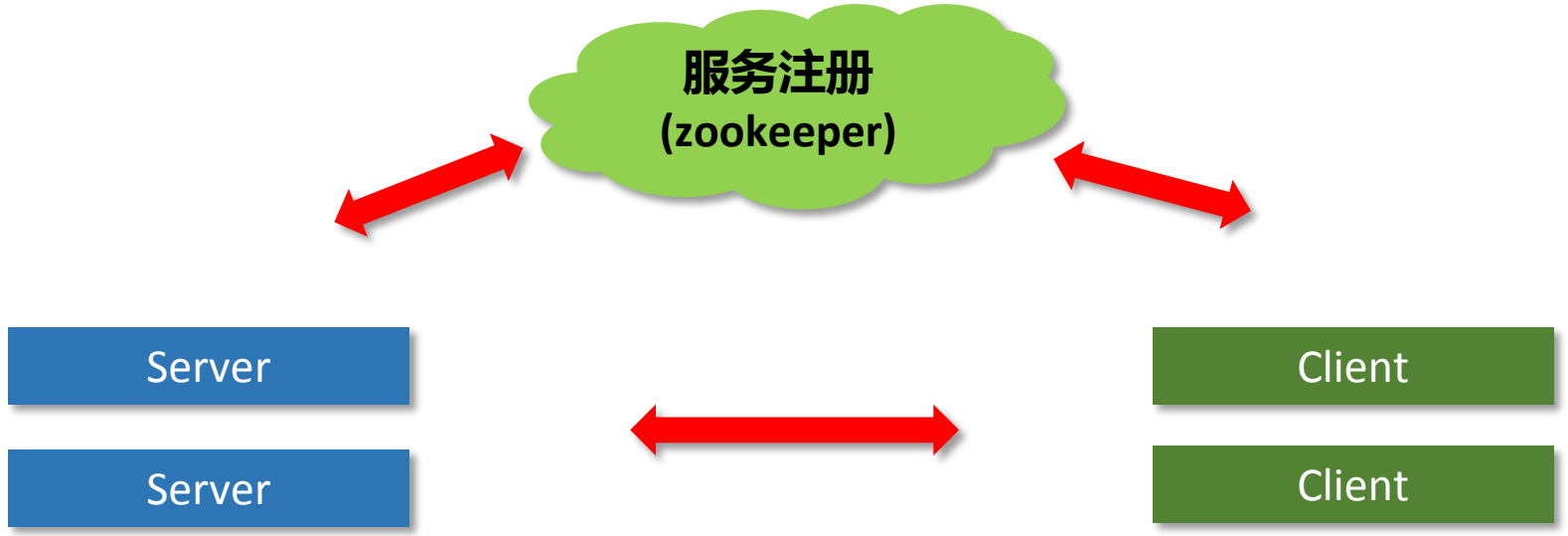


默认重试2次，超时为1000ms

2.5 超时管理

```
<!-- timeout 远程服务调用超时时间。单位：毫秒，默认：1000 -->
<!-- retries 远程服务调用重试次数，不包括第一次调用。默认：2 -->
<dubbo:service interface=“…Service” ref=“…Service”
 timeout=”1000” retries=“0” />

<dubbo:reference id=“…Service” interface=“…Service”
 timeout=”1000” retries=”0” />
```

2.6 容灾

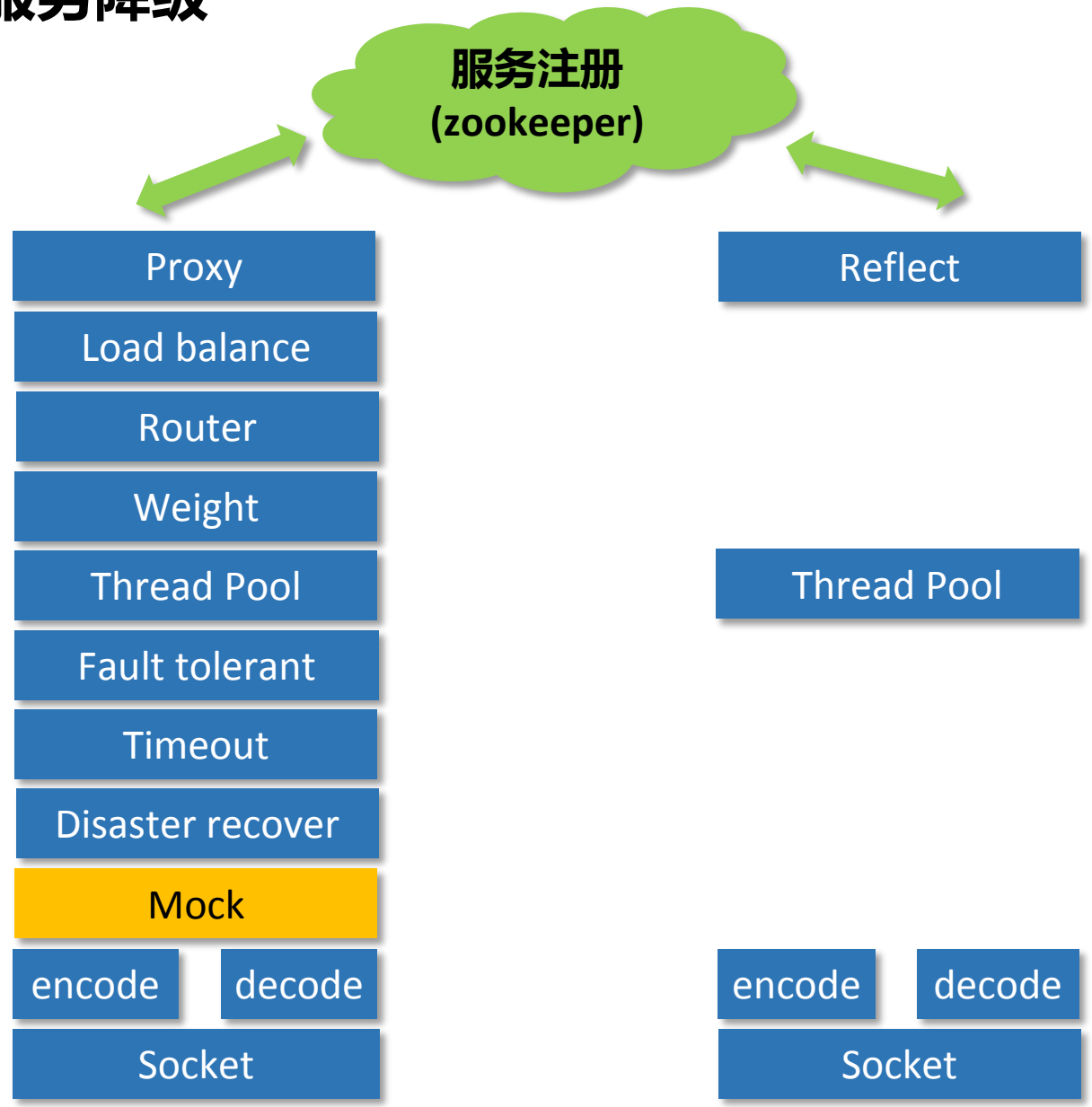


- 注册中心宕机
- 服务端宕机

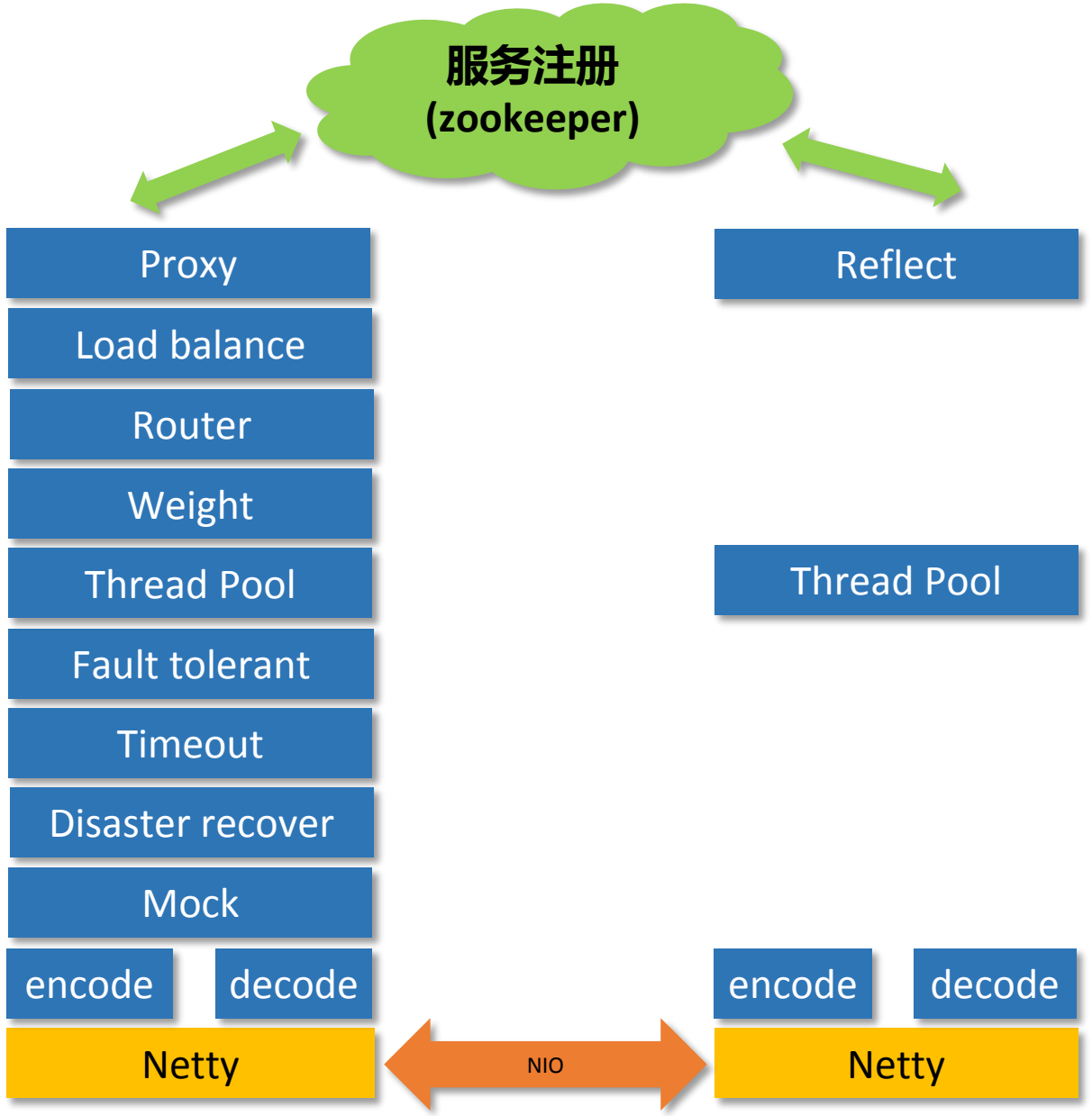
2.6.1 注册中心容灾——本地缓存

```
<!-- check 注册中心不存在时，是否报错。默认：true -->
<!-- file 使用文件缓存注册中心地址列表及服务提供者列表，  -->
<!-- 应用重启时将基于此文件恢复 -->
<dubbo:registry address=“${dubbo.registry.address}”
 check="false" file="cache/dubbomm.cache"/>
```

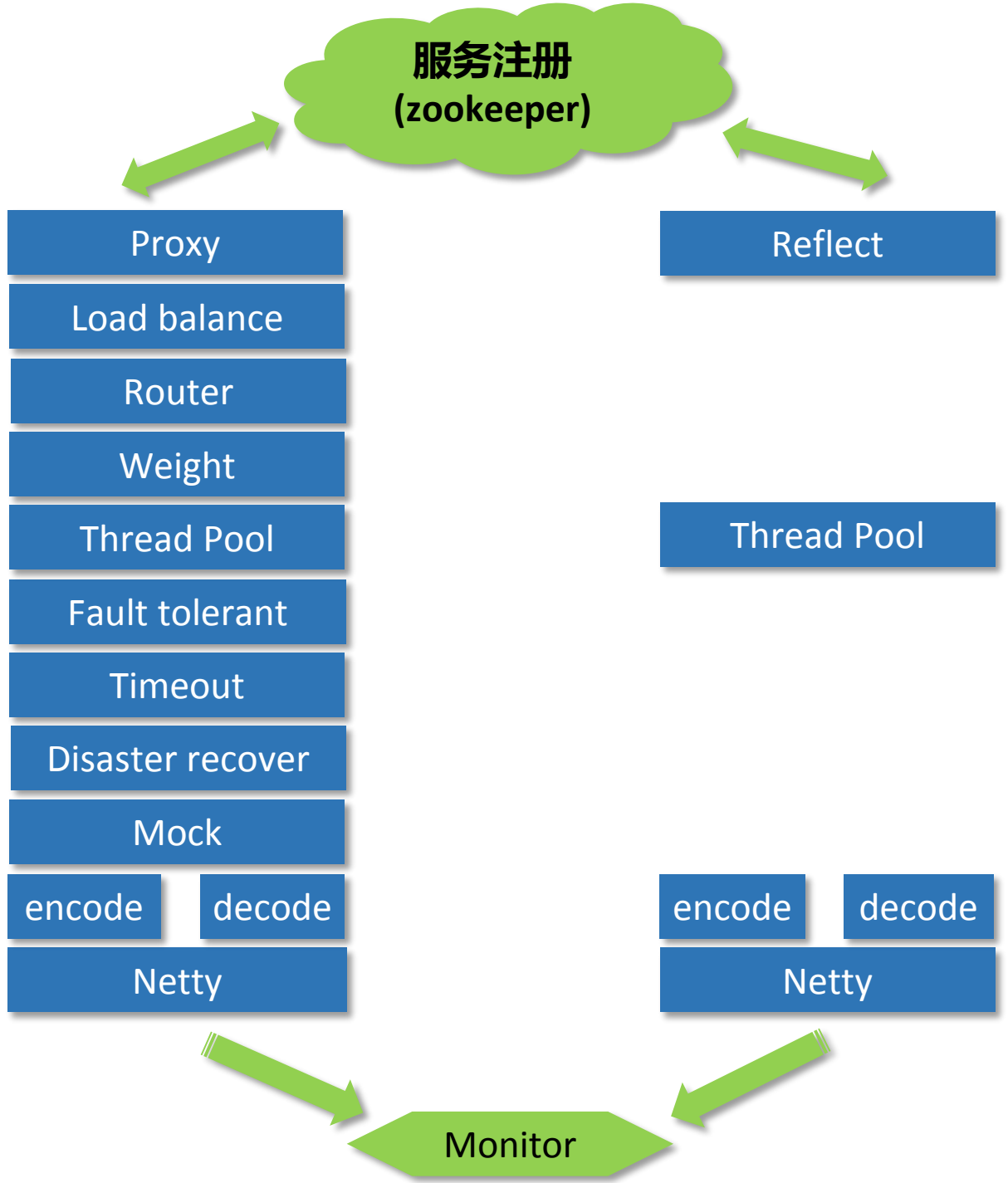
2.6.2 服务容灾——服务降级



2.7 通信模型


服务注册
(zookeeper)
Proxy
Load balance
Router
Weight
Thread Pool
Fault tolerant
Timeout
Disaster recover
Mock
encode
decode
Netty
NIO
Reflect
Thread Pool
encode
decode
Netty

2.8 监控


服务注册
(zookeeper)
Proxy
Load balance
Router
Weight
Thread Pool
Fault tolerant
Timeout
Disaster recover
Mock
encode
decode
Netty
Reflect
Thread Pool
encode
decode
Netty
Monitor

2.8 监控

```
<!-- protocol 监控中心协议，如果为protocol="registry"，表示从注册中心发现监控中心地址，否则直连监控中心。 -->
<dubbo:monitor protocol="registry" />
```

2.9 团队协作

- 直连提供者
- Mock测试
- 接口兼容
- 服务隔离
- 只订阅
- 只发布

2.9 团队协作——直连提供者

```
<dubbo:reference id="xxxService" interface="com.XxxService"
 url="dubbo://localhost:20890" />
```

2.9.1 团队协作——Mock测试

```
<dubbo:service interface= “com.foo.BarService”
 mock=”return null”/>
```

2.9.2 团队协作——接口兼容

```
<dubbo:service interface="com.XxxService" version="1.0.0" />
<dubbo:service interface="com.XxxService" version="2.0.0" />

<dubbo:reference id="xxxService" interface="com.XxxService"
 version="1.0.0" />
<dubbo:reference id="xxxService" interface="com.XxxService"
 version="2.0.0" />
```

2.9.3 团队协作——服务隔离

```
<dubbo:service interface="com.XxxService" group="pre" />
<dubbo:reference id="xxxService" interface="com.XxxService"
 group="pre" />
```

2.9.4 团队协作——只订阅

```
<dubbo:registry address="ip:port" register="false" />
```

2.9.5 团队协作——只发布

```
<dubbo:registry id="xyRegistry" address="ip:port" />
<dubbo:registry id="baRegistry" address="ip:port"
 subscribe="false" />
```

2.10 配置覆盖策略

三、为什么选择dubbo

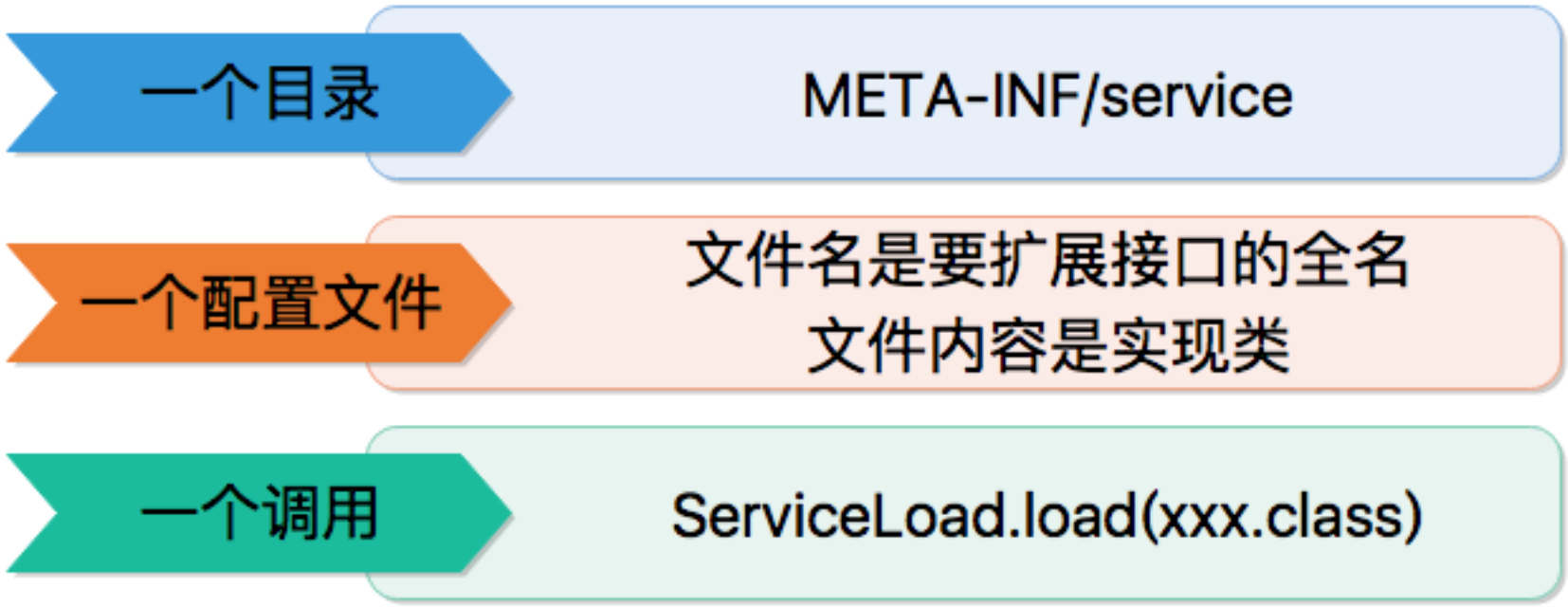
良好的设计
良好的扩展

3 SPI

Service Provider Interface.

基于同一个接口实现的可插拔

3 SPI



3 SPI 示例——接口与实现

```
public interface SpiDemo {
    void say();
}

public class SpiDemoSayHelloImpl implements SpiDemo {
    @Override
    public void say() {
        System.out.println("Hello!");
    }
}

public class SpiDemoSayGoodbyeImpl implements SpiDemo {
    @Override
    public void say() {
        System.out.println("Goodbye!");
    }
}
```

3 SPI 示例——工厂

```
public class SpiDemoFactory {
    public static Iterator<SpiDemo> getSpiDemo() {
        ServiceLoader<SpiDemo> spiDemos = ServiceLoader.load(SpiDemo.class);
        return spiDemos.iterator();
    }
}
```

3 SPI 示例——配置文件

META-INF下创建 services目录

新建 com.demo.spi.SpiDemo 文件

内容为：

com.demo.spi.SpiDemoSayGoodbyeImpl

3 SPI 示例——测试

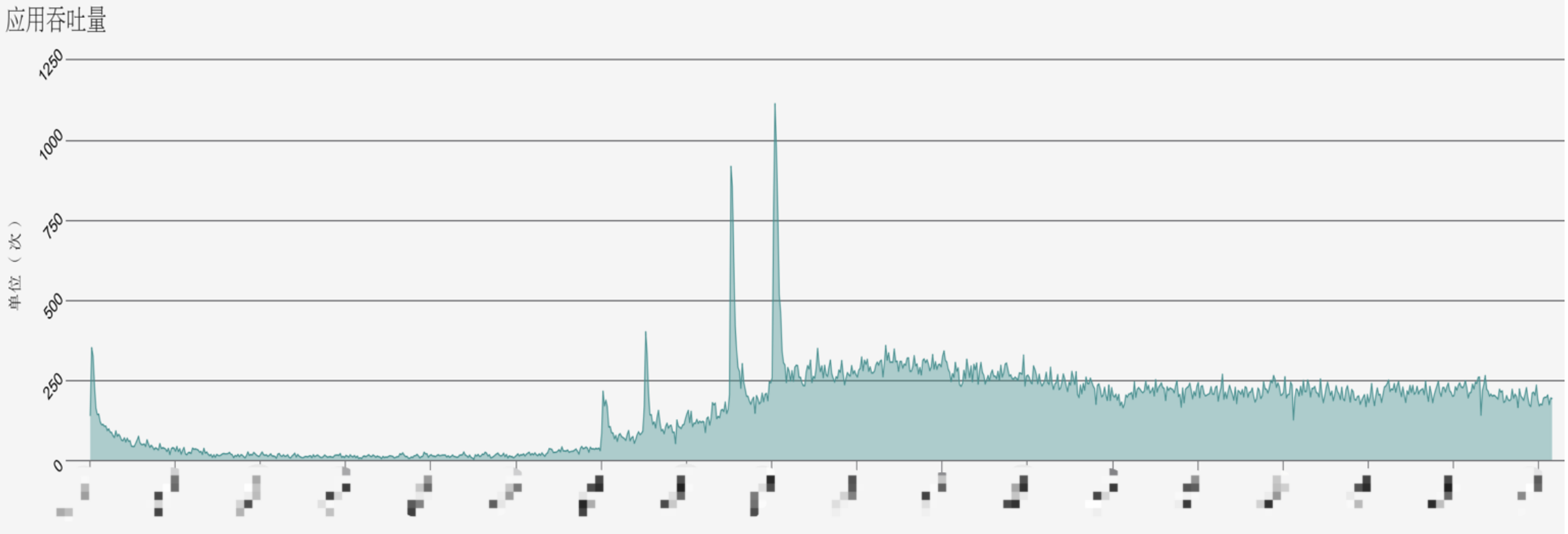
```
public class SpiDemoTest {
    public static void main(String[] args) {

        Iterator<SpiDemo> spiDemos = SpiDemoFactory.getSpiDemo();
        while (spiDemos.hasNext()) {
            SpiDemo spiDemo = spiDemos.next();
            spiDemo.say();
        }

    }
}
```

3 dubbo的扩展

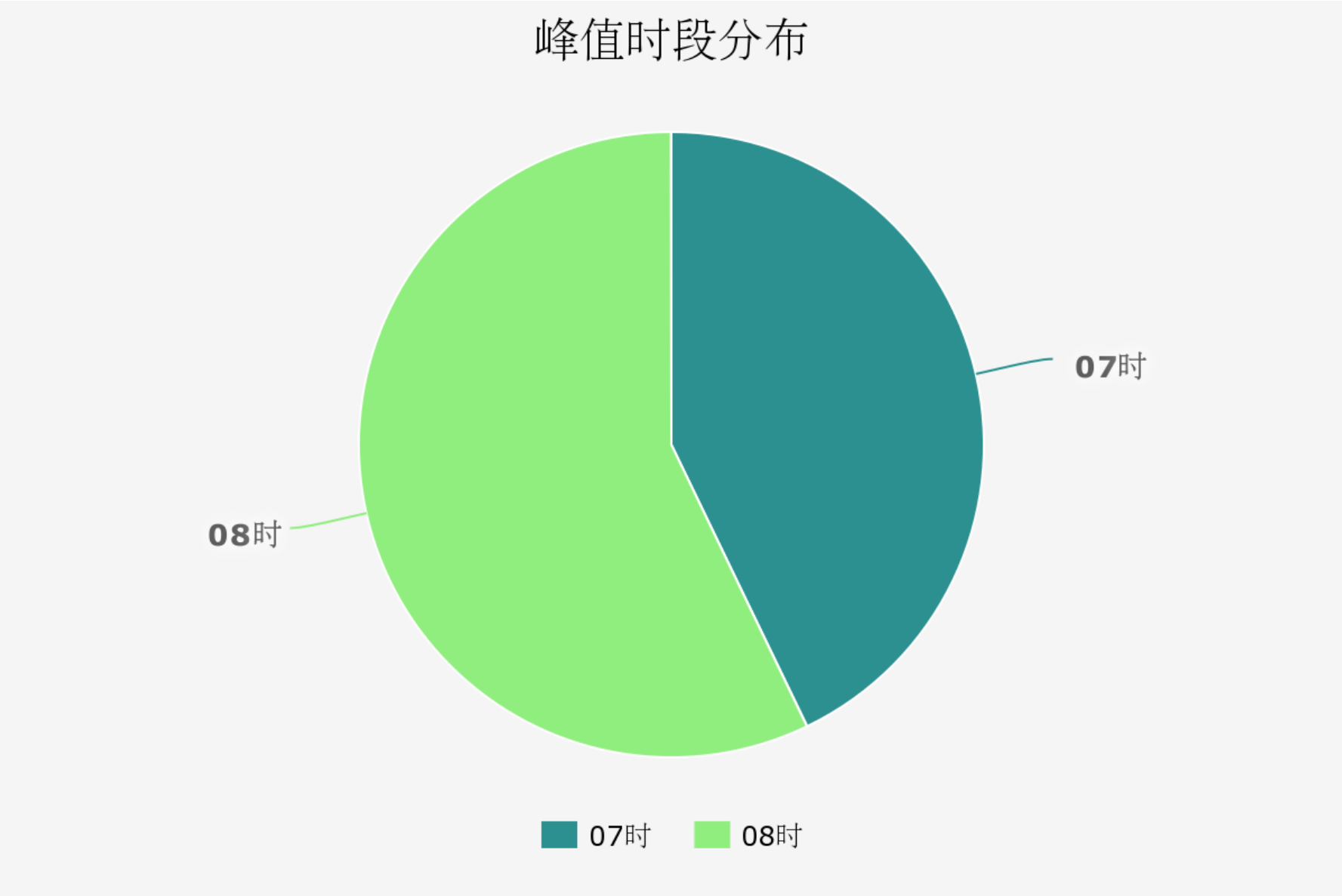
Dubbo开发手册：
https://dubbo.gitbooks.io/dubbo-dev-book/

四、dubbo管控利器——dubbomm

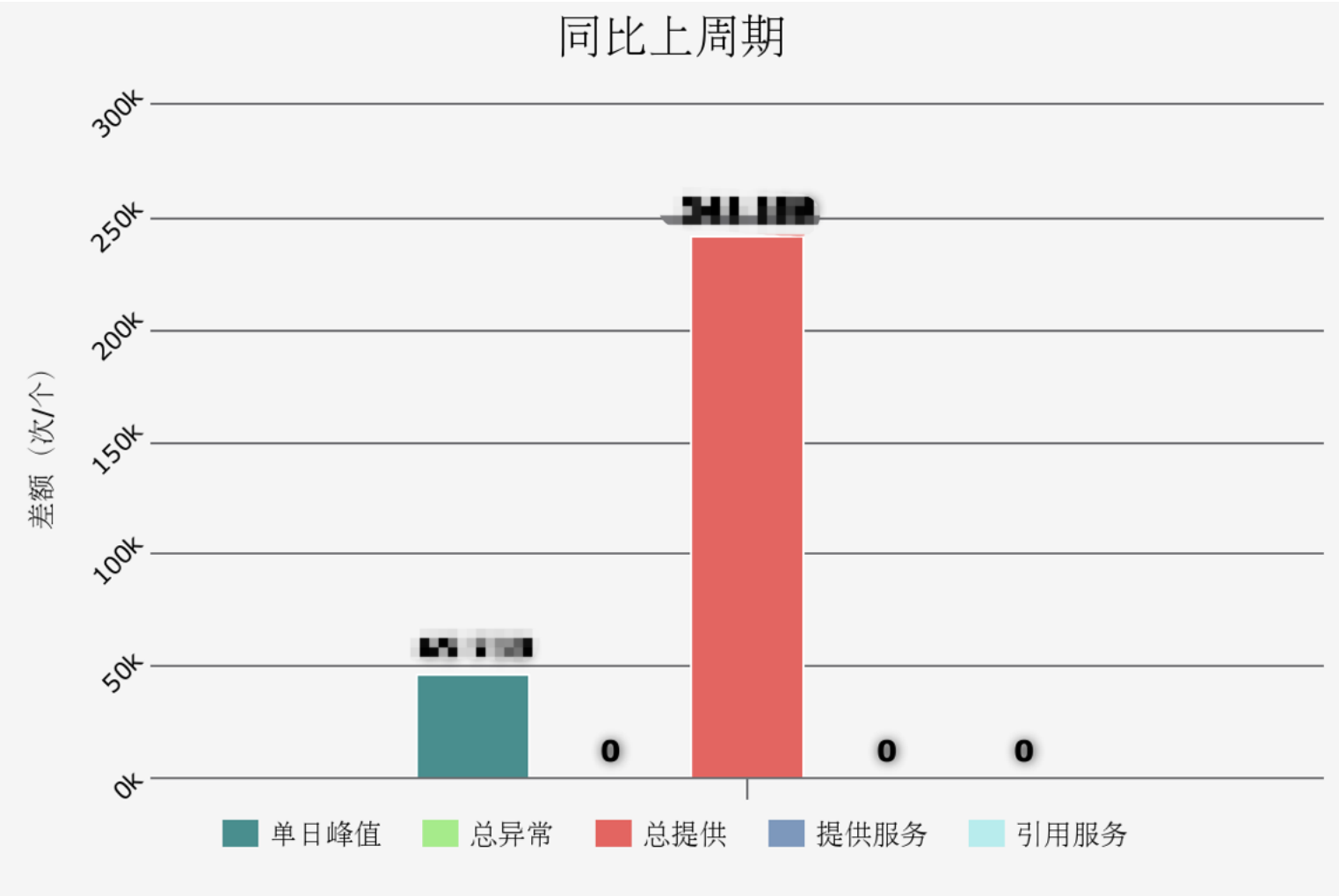
4.1.1 应用监控——吞吐量



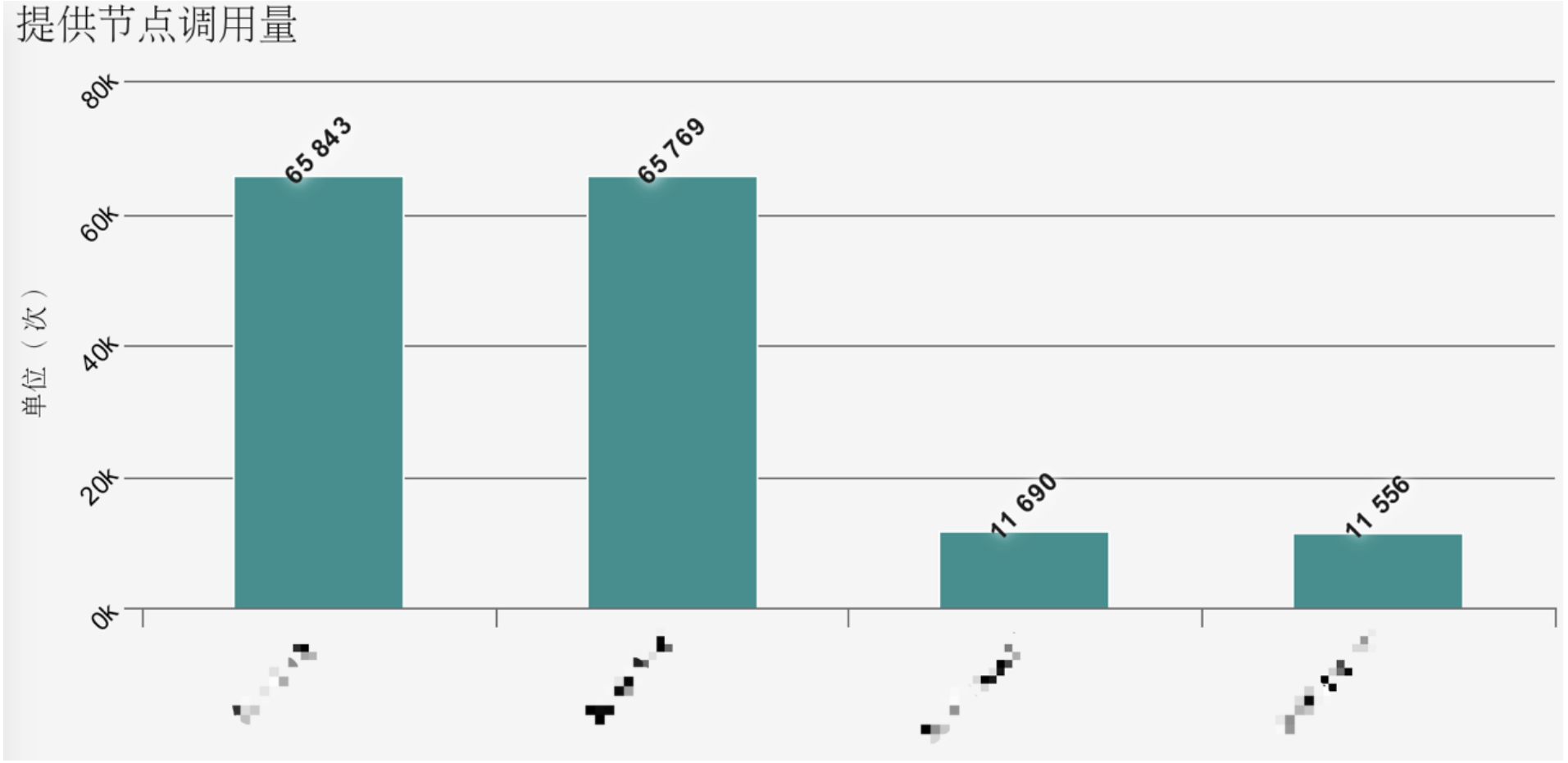
4.1.2 应用监控——峰值时段



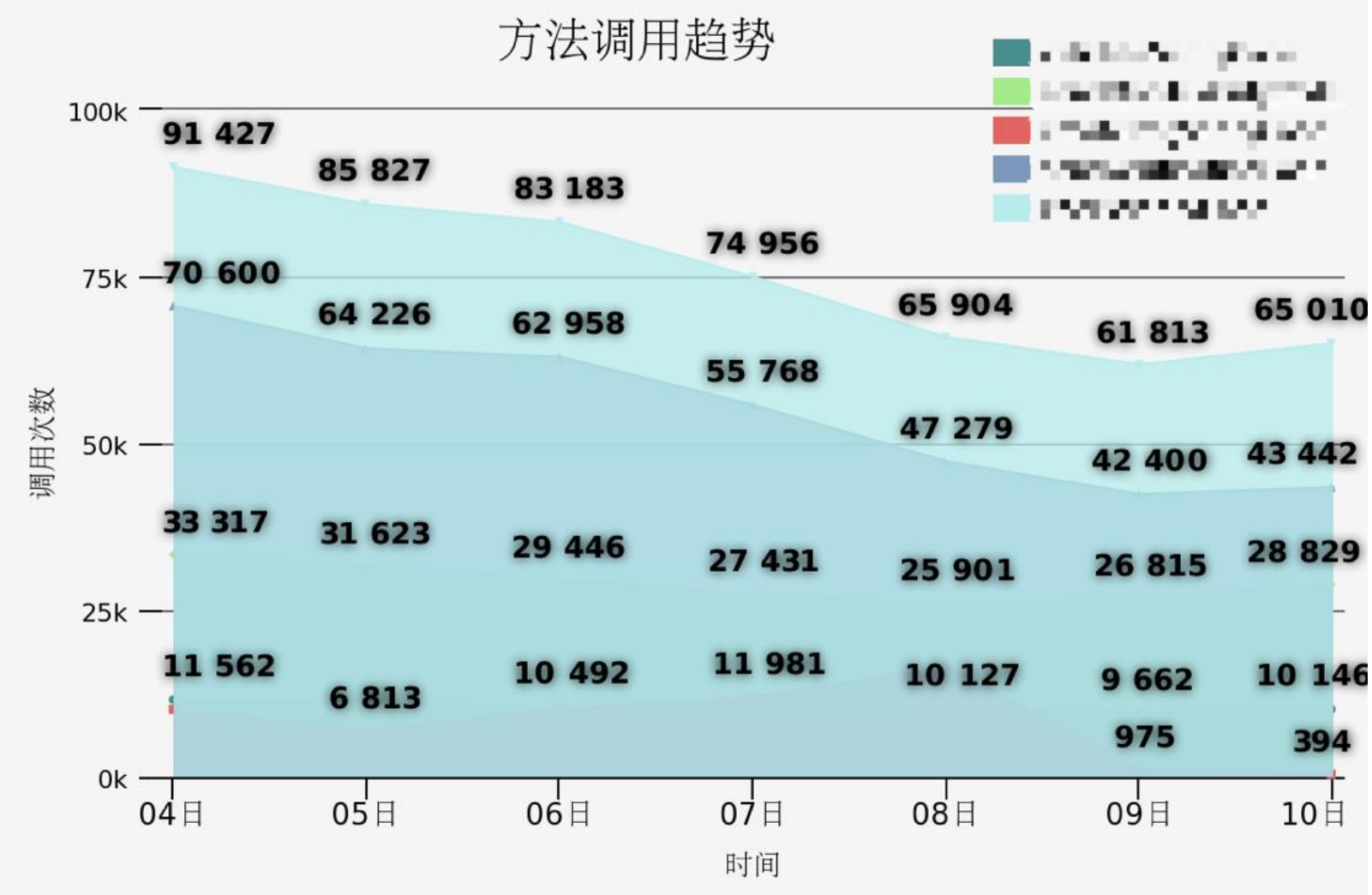
4.1.3 应用监控——同比上周期



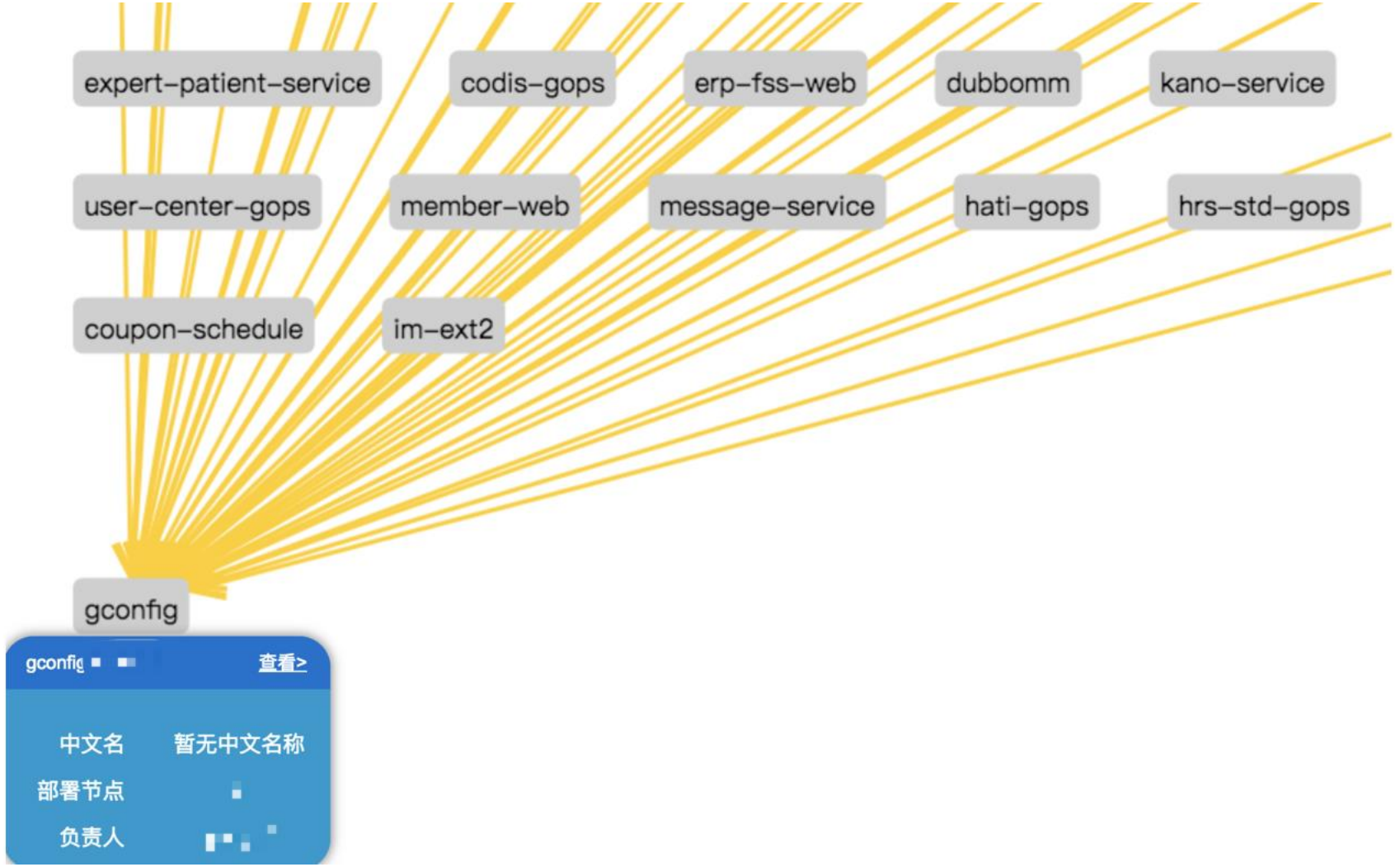
4.1.4 服务监控——节点调用量



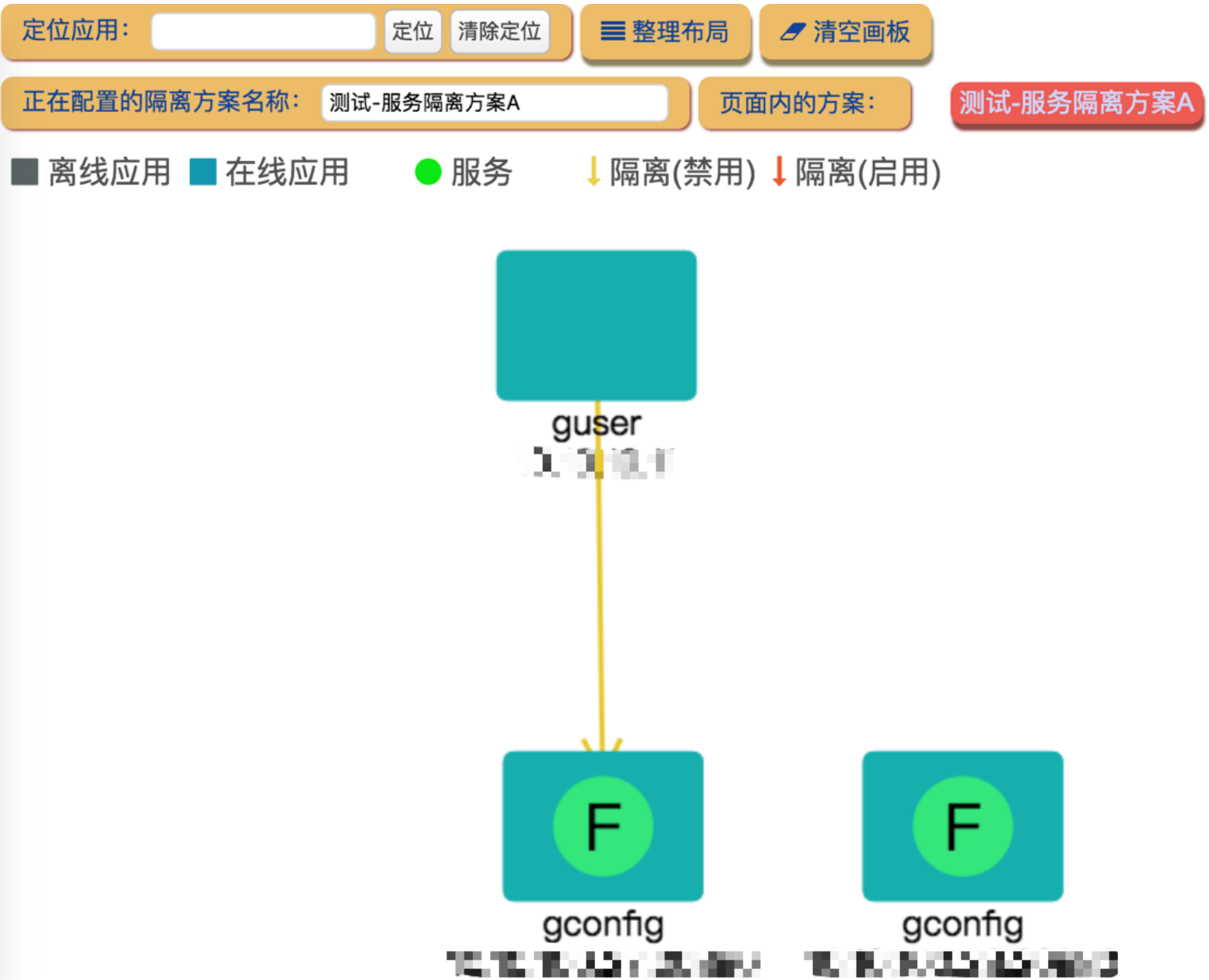
4.1.5 服务监控——方法调用趋势



4.2 应用、服务拓扑


expert-patient-service
codis-gops
erp-fss-web
dubbomm
kano-service
user-center-gops
member-web
message-service
hati-gops
hrs-std-gops
coupon-schedule
im-ext2
gconfig
gconfig
查看>
中文名
暂无中文名称
部署节点
负责人

4.3 应用、服务隔离



机房： 滨安 兴议

应用： gconfig

服务： FileService

分组：

版本：

FileService

com.greenline.gconfig.service.FileService

分组：*

版本：*

应用：gconfig

添加 定位

微医 WE DOCTOR

Dubbomm:
http://dubbomm.guahao-test.com/
负责人：张佳鉴、杨晓飞

Dubbo：
User's Guide
https://dubbo.gitbooks.io/dubbo-user-book/

Developer's Guide
https://dubbo.gitbooks.io/dubbo-dev-book/

Admin's Guide
https://dubbo.gitbooks.io/dubbo-admin-book/