

深入RPC原理（上）

学习目标

目标1：深入掌握RPC原理

目标2：掌握RPC的高级运用

目标3：Dubbo RPC的实现与Spring集成运用

1. RPC特点

1.1 RPC概述

RPC 的主要功能目标是让构建分布式计算（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。为实现该目标，RPC 框架需提供一种透明调用机制，让使用者不必显式的区分本地调用和远程调用。

RPC的优点：

- 分布式设计
- 部署灵活
- 解耦服务
- 扩展性强

1.3 RPC框架

- Dubbo：国内最早开源的 RPC 框架，由阿里巴巴公司开发并于 2011 年末对外开源，仅支持 Java 语言。
- Motan：微博内部使用的 RPC 框架，于 2016 年对外开源，仅支持 Java 语言。
- Tars：腾讯内部使用的 RPC 框架，于 2017 年对外开源，仅支持 C++ 语言。
- Spring Cloud：国外 Pivotal 公司 2014 年对外开源的 RPC 框架，提供了丰富的生态组件。
- gRPC：Google 于 2015 年对外开源的跨语言 RPC 框架，支持多种语言。
- Thrift：最初是由 Facebook 开发的内部系统跨语言的 RPC 框架，2007 年贡献给了 Apache 基金，成为 Apache 开源项目之一，支持多种语言。

详情：<https://juejin.im/post/5d5bfab6f265da03d2114215>

1.4 RPC框架优点

- RPC框架一般使用长链接，不必每次通信都要3次握手，减少网络开销。
- RPC框架一般都有注册中心，有丰富的监控管理
发布、下线接口、动态扩展等，对调用方来说是无感知、统一化的操作
协议私密，安全性较高
- RPC 协议更简单内容更小，效率更高，服务化架构、服务化治理，RPC框架是一个强力的支撑。

1.5 应用场景

特征：

- 长链接通讯。

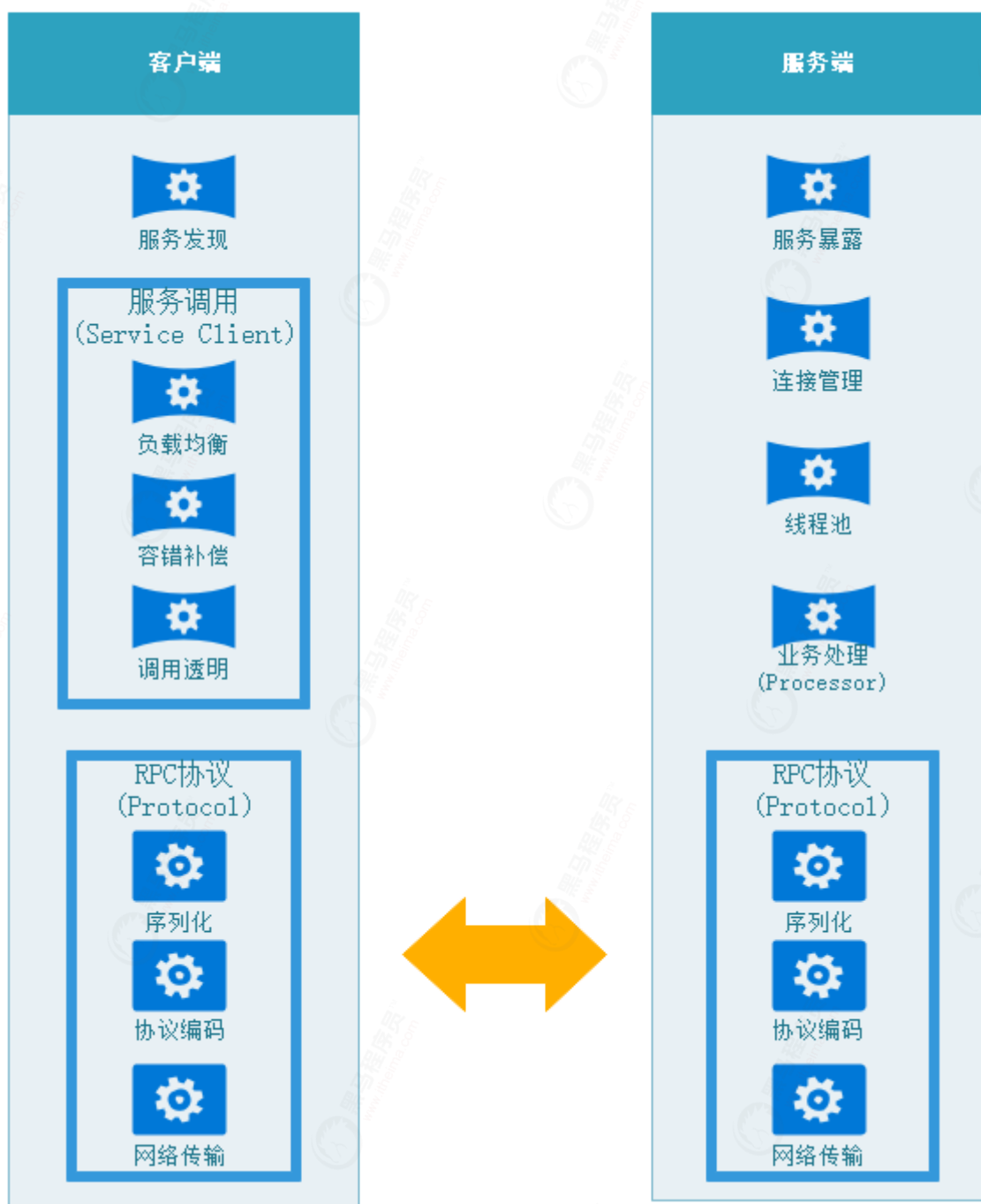
- 注册发布机制。
- 安全性，没有暴露资源操作。
- 微服务的支持。

应用例举:

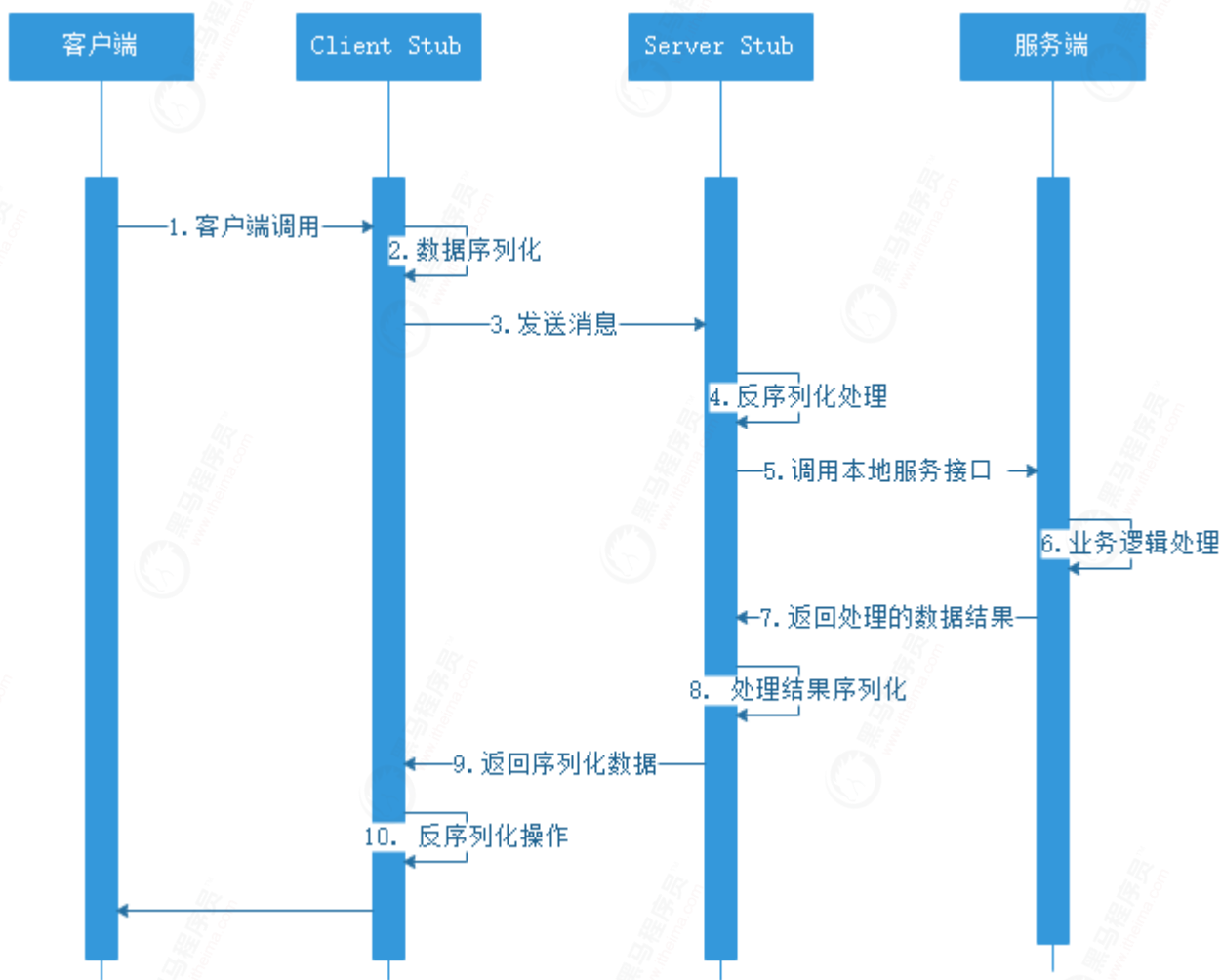
- 分布式操作系统的进程间通讯
- 构造分布式设计的软件环境
- 远程数据库服务
- 分布式应用程序设计
- 分布式程序的调试

2. 深入RPC原理

2.1 设计架构



2.2 调用流程



具体调用过程：

1. 服务消费者（client客户端）通过本地调用的方式调用服务。
2. 客户端存根（client stub）接收到请求后负责将方法、入参等信息序列化（组装）成能够进行网络传输的消息体。
3. 客户端存根（client stub）找到远程的服务地址，并且将消息通过网络发送给服务端。
4. 服务端存根（server stub）收到消息后进行解码（反序列化操作）。
5. 服务端存根（server stub）根据解码结果调用本地的服务进行相关处理。
6. 本地服务执行具体业务逻辑并将处理结果返回给服务端存根（server stub）。
7. 服务端存根（server stub）将返回结果重新打包成消息（序列化）并通过网络发送至消费方。
8. 客户端存根（client stub）接收到消息，并进行解码（反序列化）。
9. 服务消费方得到最终结果。

所涉及的技术：

1. 动态代理

生成Client Stub（客户端存根）和Server Stub（服务端存根）的时候需要用到java动态代理技术。

2. 序列化 在网络中，所有的数据都将会被转化为字节进行传送，需要对这些参数进行序列化和反序列化操作。

目前主流高效的开源序列化框架有Kryo、fastjson、Hessian、Protobuf等。

3. NIO通信

Java 提供了 NIO 的解决方案，Java 7 也提供了更优秀的 NIO.2 支持。可以采用Netty或者mina框架来解决 NIO数据传输的问题。开源的RPC框架Dubbo就是采用NIO通信，集成支持netty、mina、grizzly。

4. 服务注册中心

通过注册中心，让客户端连接调用服务端所发布的服务。主流的注册中心组件：Redis、Zookeeper、Consul、Etcd。Dubbo采用的是ZooKeeper提供服务注册与发现功能。

5. 负载均衡

在高并发的场景下，需要多个节点或集群来提升整体吞吐能力。

6. 健康检查

健康检查包括，客户端心跳和服务端主动探测两种方式。

2.3 Dubbo Spring Cloud调用

1. 集成搭建配置

2. Dubbo Rpc调用演示

3. sentinel启动命令：

```
java -Dserver.port=8090 -Dcsp.sentinel.dashboard.server=localhost:8090 -jar
D:/TestCode/sources/sentinel-dashboard-1.6.3.jar
```

4. Nacos Sentinel配置：

sentinel-degrade为降级配置策略， 内容：

```
[
  {
    "resource": "placeOrder",
    "count": 0.2,
    "grade": 1,
    "timeWindow": 4
  }
]
```

resource: 为资源名称。

count: 为百分比[0-1]， 这里代表20%

grade: 为降级策略， 0: 代表响应时间， 1: 代表异常比例， 2: 代表异常数量， 这里采用的是异常比。

timeWindow: 为时间窗， 单位为秒。

sentinel-user-flow为限流配置策略： 内容：

```
[
  {
    "resource": "placeOrder",
    "controlBehavior": 0,
    "count": 2,
    "grade": 1,
    "limitApp": "default",
    "strategy": 0
  }
]
```

resource: 为资源名称。

controlBehavior: 流量整形的控制效果，目前支持快速失败和匀速排队两种模式，默认是0, 快速失败。

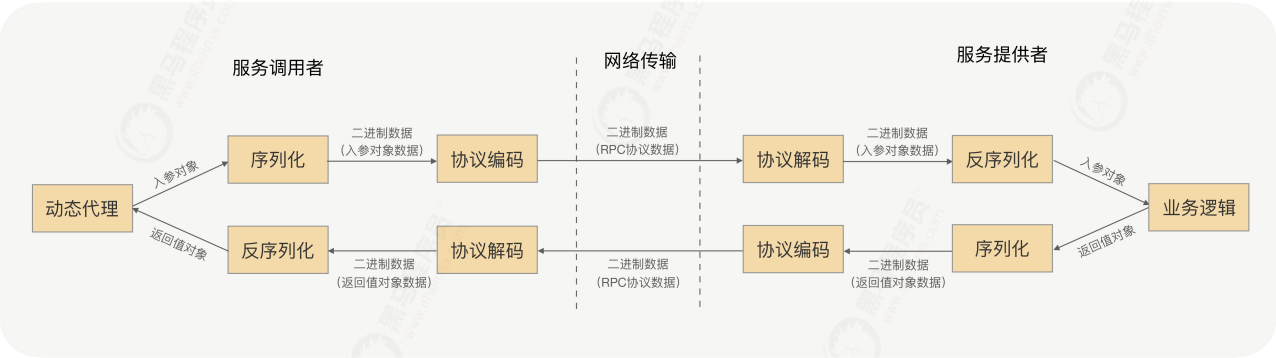
count: 线程数量。

grade: 限流配置策略， 0: 代表线程数量， 1: 代表QPS并发数。

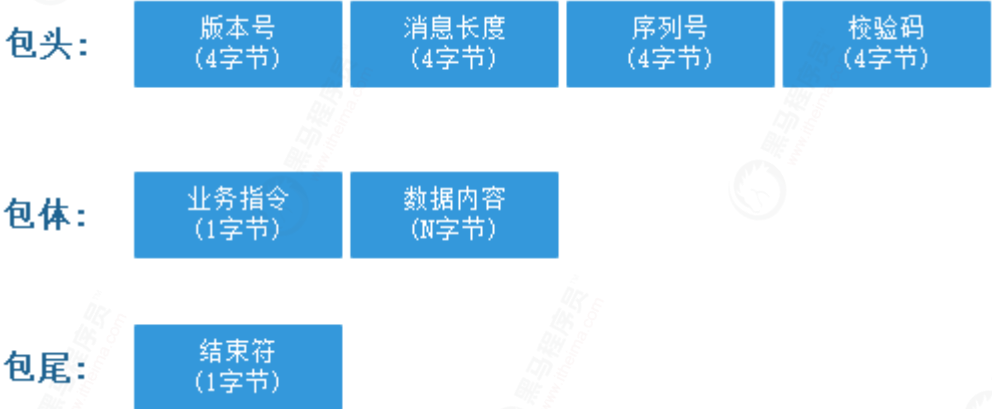
limitApp: 限流针对的来源， 填写default即可。

2.4 RPC深入解析

2.4.1 序列化技术



- 序列化的作用
- 在网络传输中，数据必须采用二进制形式，所以在RPC调用过程中，需要采用序列化技术，对入参对象和返回值对象进行序列化与反序列化。
- 如何进行序列化
- 自定义的二进制协议来实现序列化：



行序列化? 下面以User对象例举讲解:

User对象:

一个对象是如何进

```

package com.itcast;
public class User {

    /**
     * 用户编号
     */
    private String userNo = "0001";

    /**
     * 用户名称
     */
    private String name = "zhangsan";

}

```

包体的数据组成:

业务指令为0x00000001占1个字节, 类的包名com.itcast占10个字节, 类名User占4个字节;

属性UserNo名称占6个字节, 属性类型string占2个字节表示, 属性值为0001占4个字节;

属性name名称占4个字节, 属性类型string占2个字节表示, 属性值为zhangsan占8个字节;

包体共计占有 $1+10+4+6+2+4+4+2+8 = 41$ 字节。

包头的数据组成:

版本号v1.0占4个字节, 消息包体实际长度为41占4个字节表示, 序列号0001占4个字节, 校验码32位表示占4个字节。

包头共计占有 $4+4+4+4 = 16$ 字节。

包尾的数据组成:

通过回车符标记结束\r\n, 占用1个字节。

整个包的序列化二进制字节流共 $41+16+1 = 58$ 字节。这里讲解的是整个序列化的处理思路, 在实际的序列化处理中还要考虑更多细节, 比如说方法和属性的区分, 方法权限的标记, 嵌套类型的处理等等。

● 序列化的处理要素

1. **解析效率**: 序列化协议应该首要考虑的因素, 像xml/json解析起来比较耗时, 需要解析dom树, 二进制自定义协议解析起来效率要快很多。
2. **压缩率**: 同样一个对象, xml/json传输起来有大量的标签冗余信息, 信息有效性低, 二进制自定义协议占用的空间相对来说会小很多。
3. **扩展性与兼容性**: 是否能够利于信息的扩展, 并且增加字段后旧版客户端是否需要强制升级, 这都是需要考虑的问题, 在自定义二进制协议时候, 要做好充分考虑设计。
4. **可读性与可调试性**: xml/json的可读性会比二进制协议好很多, 并且通过网络抓包是可以直接读取, 二进制则需要反序列化才能查看其内容。
5. **跨语言**: 有些序列化协议是与开发语言紧密相关的, 例如dubbo的Hessian序列化协议就只能支持Java的RPC调用。
6. **通用性**: xml/json非常通用, 都有很好的第三方解析库, 各个语言解析起来都十分方便, 二进制数据的处理方面也有Protobuf和Hessian等插件, 在做设计的时候尽量做到较好的通用性。

● 常用的序列化技术

1. JDK原生序列化

代码:

```

...
public static void main(String[] args) throws IOException, ClassNotFoundException {
    String basePath = "D:/TestCode";
    FileOutputStream fos = new FileOutputStream(basePath + "tradeUser.clazz");
    TradeUser tradeUser = new TradeUser();
    tradeUser.setName("Mirson");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(tradeUser);
    oos.flush();
    oos.close();
    FileInputStream fis = new FileInputStream(basePath + "tradeUser.clazz");
    ObjectInputStream ois = new ObjectInputStream(fis);
    TradeUser deStudent = (TradeUser) ois.readObject();
    ois.close();
    System.out.println(deStudent);
}
...

```

- (1) 在Java中，序列化必须要实现java.io.Serializable接口。
- (2) 通过ObjectOutputStream和ObjectInputStream对象进行序列化及反序列化操作。
- (3) 虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，一个非常重要的一点是两个类的序列化 ID 是否一致（也就是在代码中定义的序列ID private static final long serialVersionUID）
- (4) 序列化并不会保存静态变量。
- (5) 要想将父类对象也序列化，就需要让父类也实现Serializable 接口。
- (6) Transient 关键字的作用是控制变量的序列化，在变量声明前加上该关键字，可以阻止该变量被序列化到文件中，在被反序列化后，transient 变量的值被设为初始值，如基本类型 int为 0，封装对象型 Integer则为null。
- (7) 服务器端给客户端发送序列化对象数据并非加密的，如果对象中有一些敏感数据比如密码等，那么在对密码字段序列化之前，最好做加密处理,这样可以一定程度保证序列化对象的数据安全。

2. JSON序列化

一般在HTTP协议的RPC框架通信中，会选择JSON方式。

优势：JSON具有较好的扩展性、可读性和通用性。

缺陷：JSON序列化占用空间开销较大,没有JAVA的强类型区分，需要通过反射解决，解析效率和压缩率都较差。

如果对并发和性能要求较高，或者是传输数据量较大的场景，不建议采用JSON序列化方式。

3. Hessian2序列化

Hessian 是一个动态类型，二进制序列化，并且支持跨语言特性的序列化框架。

Hessian 性能上要比JDK、JSON 序列化高效很多，并且生成的字节数也更小。有非常好的兼容性和稳定性，所以 Hessian 更加适合作为 RPC 框架远程通信的序列化协议。

代码示例：

```

...
TradeUser tradeUser = new TradeUser();
tradeUser.setName("Mirson");

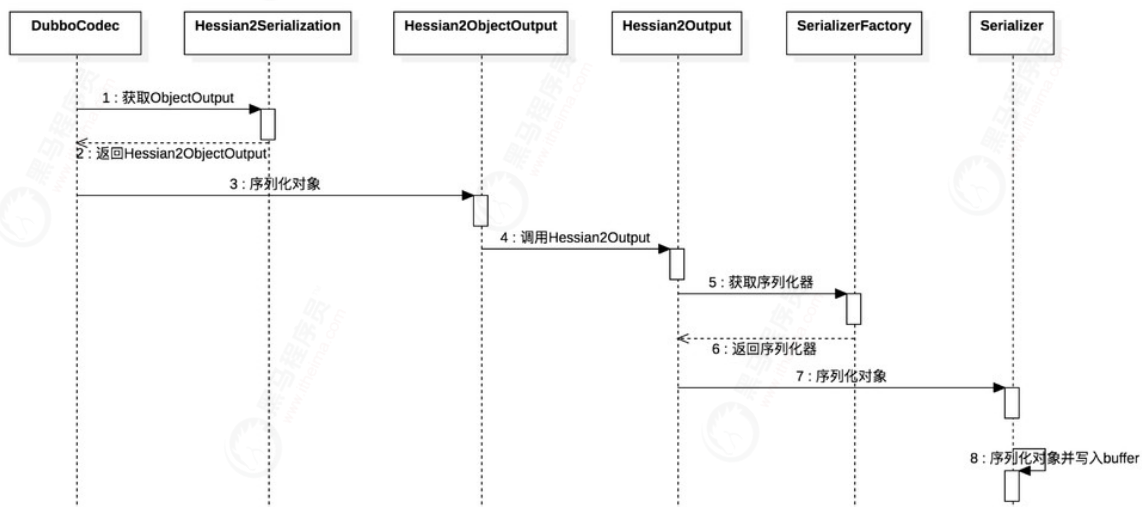
//tradeUser对象序列化处理
ByteArrayOutputStream bos = new ByteArrayOutputStream();
Hessian2Output output = new Hessian2Output(bos);
output.writeObject(tradeUser);
output.flushBuffer();
byte[] data = bos.toByteArray();
bos.close();

//tradeUser对象反序列化处理
ByteArrayInputStream bis = new ByteArrayInputStream(data);
Hessian2Input input = new Hessian2Input(bis);
TradeUser deTradeUser = (TradeUser) input.readObject();
input.close();

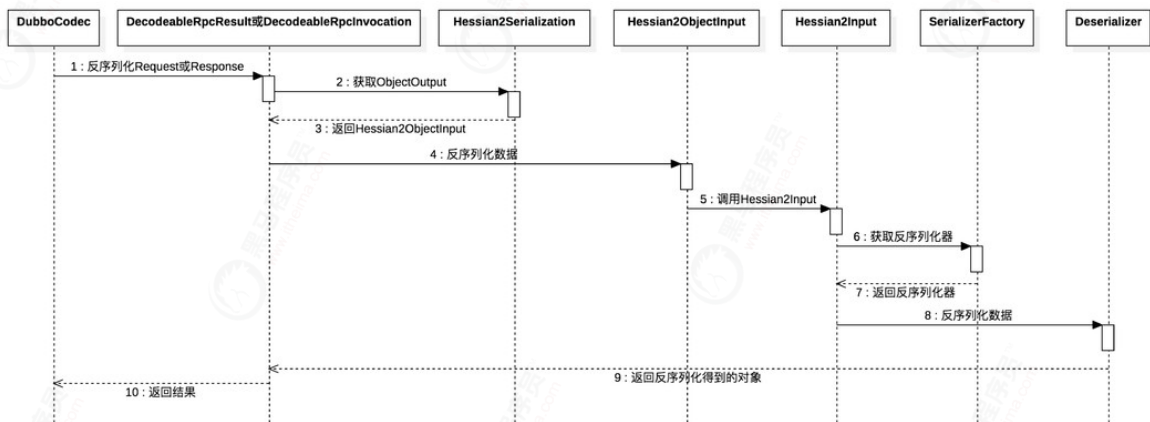
System.out.println(deTradeUser);
...

```

Dubbo Hessian Lite序列化流程:



Dubbo Hessian Lite反序列化流程:



Hessian自身也存在一些缺陷，大家在使用过程中要注意：

- 对Linked系列对象不支持，比如LinkedHashMap、LinkedHashSet 等，但可以通过CollectionSerializer类修复。
- Locale 类不支持，可以通过扩展 ContextSerializerFactory 类修复。
- Byte/Short 在反序列化的时候会转成 Integer。

4. Protobuf序列化

Protobuf 是 Google 推出的开源序列库，它是一种轻便、高效的结构化数据存储格式，可以用于结构化数据序列化，支持 Java、Python、C++、Go 等多种语言。

Protobuf 使用的时候需要定义 IDL（Interface description language），然后使用不同语言的 IDL 编译器，生成序列化工具类，它具备以下优点：

- 压缩比高，体积小，序列化后体积相比 JSON、Hessian 小很多；
- IDL 能清晰地描述语义，可以帮助并保证应用程序之间的类型不会丢失，无需类似 XML 解析器；
- 序列化反序列化速度很快，不需要通过反射获取类型；
- 消息格式的扩展、升级和兼容性都不错，可以做到向后兼容。

代码示例：

Protobuf脚本定义：

```
// 定义Proto版本
syntax = "proto3";
// 是否允许生成多个JAVA文件
option java_multiple_files = false;
// 生成的包路径
option java_package = "com.itcast.bulls.stock.struct.netty.trade";
// 生成的JAVA类名
option java_outer_classname = "TradeUserProto";

// 预警通知消息体
message TradeUser {

    /**
     * 用户ID
     */
    int64 userId = 1 ;

    /**
     * 用户名称
     */
    string userName = 2 ;
}
```

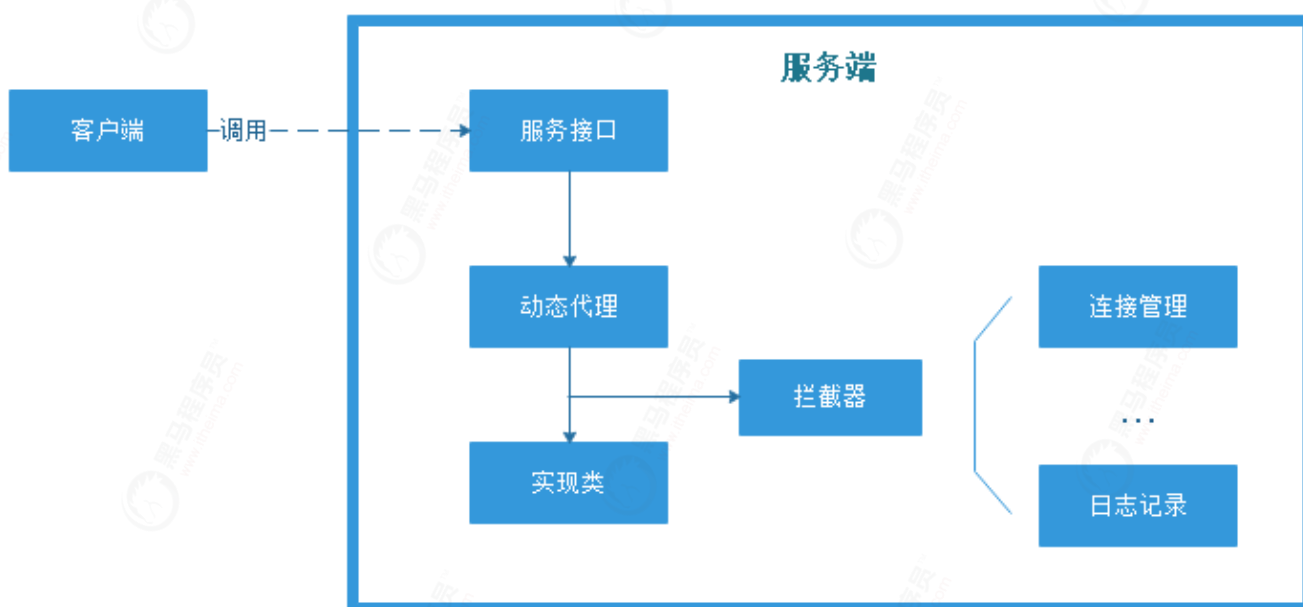
代码操作：

```
// 创建TradeUser的Protobuf对象
TradeUserProto.TradeUser.Builder builder = TradeUserProto.TradeUser.newBuilder();
builder.setUserId(101);
builder.setUserName("Mirson");

//将TradeUser做序列化处理
TradeUserProto.TradeUser msg = builder.build();
byte[] data = msg.toByteArray();

//反序列化处理，将刚才序列化的byte数组转化为TradeUser对象
TradeUserProto.TradeUser deTradeUser = TradeUserProto.TradeUser.parseFrom(data);
System.out.println(deTradeUser);
```

2.4.2 动态代理



- 内部接口如何调用实现?
RPC的调用内部核心技术采用的就是动态代理。
- JDK动态代理的如何实现?
实例代码:

```

public class JdkProxyTest {

    /**
     * 定义用户的接口
     */
    public interface User {
        String job();
    }

    /**
     * 实际的调用对象
     */
    public static class Teacher {

        public String invoke(){
            return "i'm Teacher";
        }

    }

    /**
     * 创建JDK动态代理类
     */
    public static class JDKProxy implements InvocationHandler {
        private Object target;

        JDKProxy(Object target) {
            this.target = target;
        }

        @Override
        public Object invoke(Object proxy, Method method, Object[] paramValues) {
            return ((Teacher)target).invoke();
        }

    }

    public static void main(String[] args){
        // 构建代理器
        JDKProxy proxy = new JDKProxy(new Teacher());
        ClassLoader classLoader = ClassLoaderUtils.getClassLoader();

        // 生成代理类
        User user = (User) Proxy.newProxyInstance(classLoader, new Class[]{User.class},
        proxy);

        // 接口调用
        System.out.println(user.job());
    }
}

```

JDK动态代理的实现原理:

代理的创建:



代理的调用:



JDK内部如何处理?

代理类 \$Proxy 里面会定义相同签名的接口, 然后内部会定义一个变量绑定JDKProxy代理对象, 当调用 User.job 接口方法, 实质上调用的是JDKProxy.invoke()方法。

- 为什么要加入动态代理?

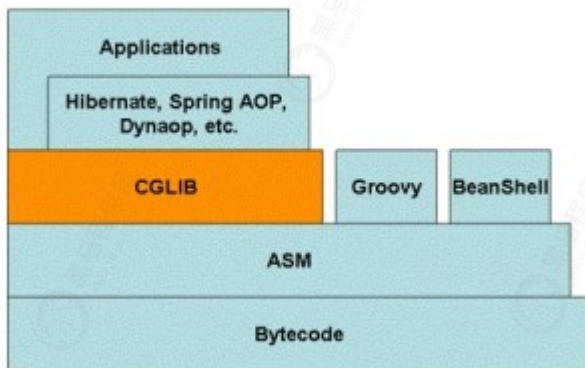
第一, 缺点: 不便于管理, 不利于扩展维护。

第二, 优点: 可以做到拦截, 添加其他额外功能。

- 动态代理开源技术

(1) Cglib 动态代理

Cglib是一个强大的、高性能的代码生成包。



(2) Javassist 动态代理

一个开源的分析、编辑和创建Java字节码的类库。

(3) Byte Buddy 字节码增强库

Byte Buddy是致力于解决字节码操作和 简化操作复杂性的开源框架。

几种动态代理性能比较:

综合结果:

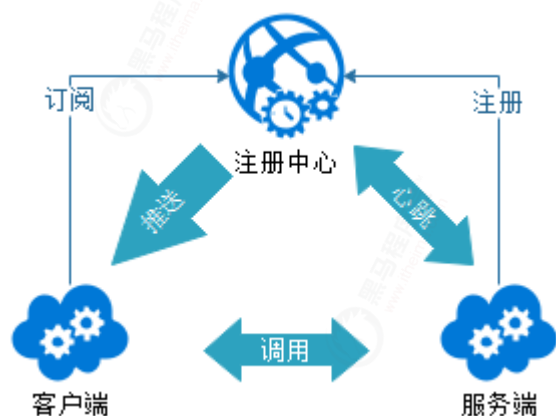
Byte Buddy > Javassist > CGLIB > JDK。

2.4.3 服务注册发现

- 服务注册发现的作用

感知服务端的变化, 获取最新服务节点的连接信息。

- 服务注册发现的处理流程



服务注册：服务提供方将对外暴露的接口发布到注册中心内，注册中心为了检测服务的有效状态，一般会建立双向心跳机制。

服务订阅：服务调用方去注册中心查找并订阅服务提供方的 IP，并缓存到本地用于后续调用。

- 如何实现服务的注册发现

基于 ZooKeeper 的服务发现方式：



A. 在 ZooKeeper 中创建一个服务根路径，可以根据接口命名命名（例如：/micro/service/com.itcast.orderService），在这个路径再创建服务提供方与调用方目录（server、client），分别用来存储服务提供方和调用方的节点信息。

B. 服务端发起注册时，会在服务提供方目录中创建一个临时节点，节点中存储注册信息。

C. 客户端发起订阅时，会在服务调用方目录中创建一个临时节点，节点中存储调用方的信息，同时watch 服务提供方的目录（/micro/service/com.itcast.orderService/server）中所有的服务节点数据。当服务端产生变化时ZK就会通知给订阅的客户端。

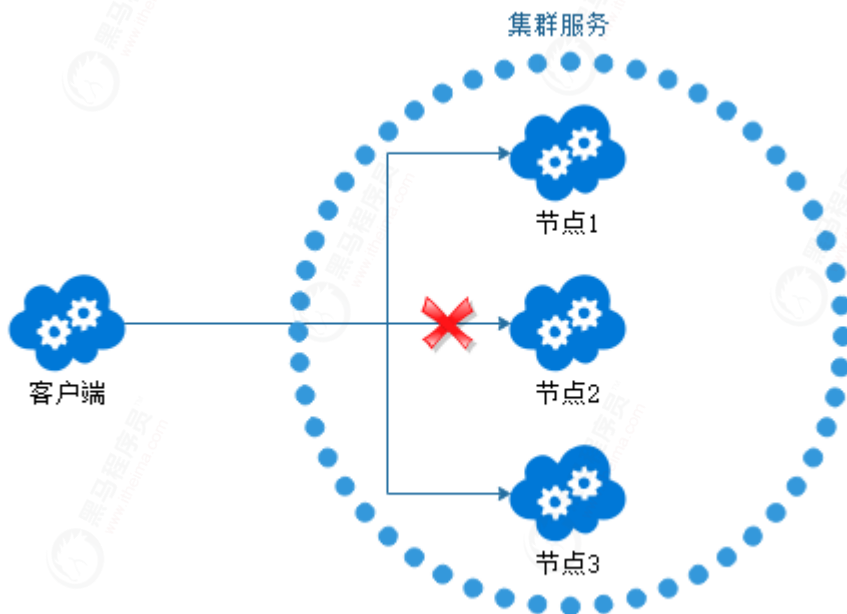
ZooKeeper方案的特点：

强一致性，ZooKeeper 集群的每个节点的数据每次发生更新操作，都会通知其它 ZooKeeper 节点同时执行更新。

2.4.4 健康监测

- 为什么需要健康监测？

比如网络中的波动，硬件设施的老化等等。可能造成集群当中的某个节点存在问题，无法正常调用。



- 健康监测实现分析

心跳检测的过程总共包含以下状态:

- 健康状态
- 波动状态
- 失败状态

- 完善的解决方案

- (1) 阈值: 健康监测增加失败阈值记录。
- (2) 成功率: 可以再追加调用成功率的记录 (成功次数/总次数)。
- (3) 探针: 对服务节点有一个主动的存活检测机制。

2.4.5 网络IO模型

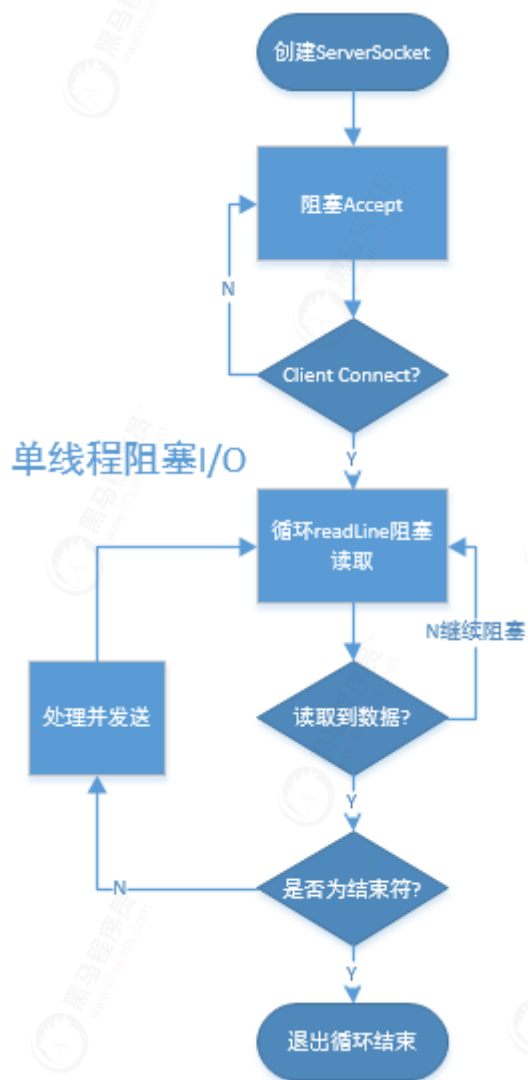
- 有哪些网络IO模型

分为五种:

- 同步阻塞 IO (BIO)
- 同步非阻塞 IO (NIO)
- IO 多路复用
- 信号驱动IO
- 异步非阻塞 IO (AIO)

常用的是同步阻塞 IO 和 IO 多路复用模型。

- 什么是阻塞IO模型



- **IO多路复用**

IO多路复用的实现主要有select，poll和epoll模式。

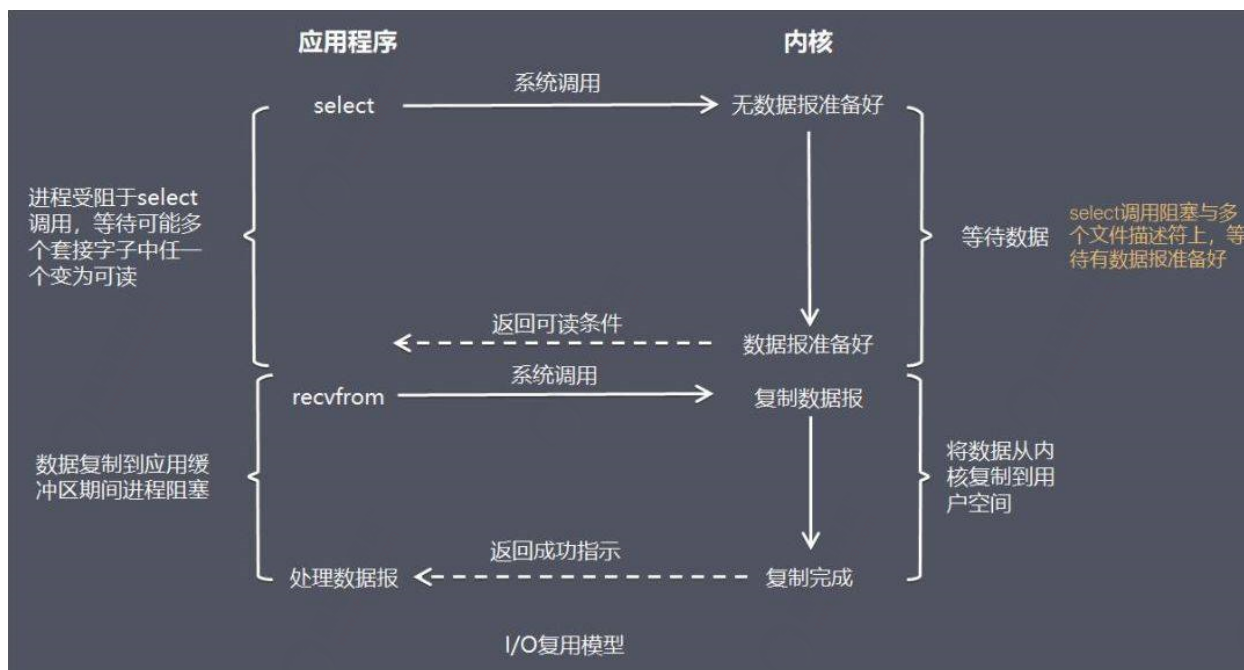
文件描述符：

在Linux系统中一切皆可以看成是文件，文件又可分为：普通文件、目录文件、链接文件和设备文件。

三者的区别：

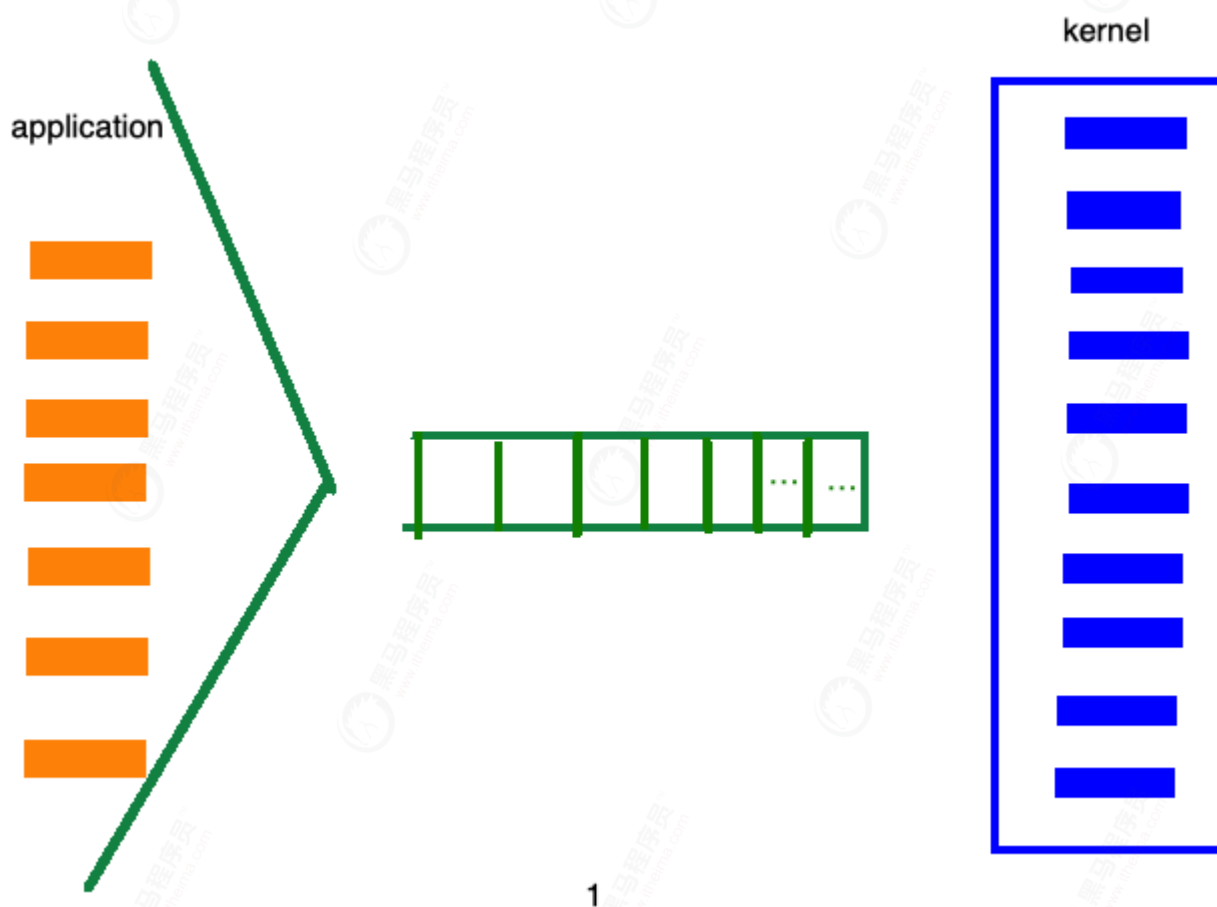
	select	poll	epoll
操作方式	遍历	遍历	回调
底层实现	bitmap	数组	红黑树
IO效率	每次调用都进行线性遍历，时间复杂度为 $O(n)$	每次调用都进行线性遍历，时间复杂度为 $O(n)$	事件通知方式，每当fd就绪，系统注册的回调函数就会被调用，将就绪fd放到readyList里面，时间复杂度 $O(1)$
最大连接数	1024 (x86) 或 2048 (x64)	无上限	无上限
fd拷贝	每次调用select，都需要把fd集合从用户态拷贝到内核态	每次调用poll，都需要把fd集合从用户态拷贝到内核态	调用epoll_ctl时拷贝进内核并保存，之后每次epoll_wait不拷贝

基于select的 I/O 复用模型：

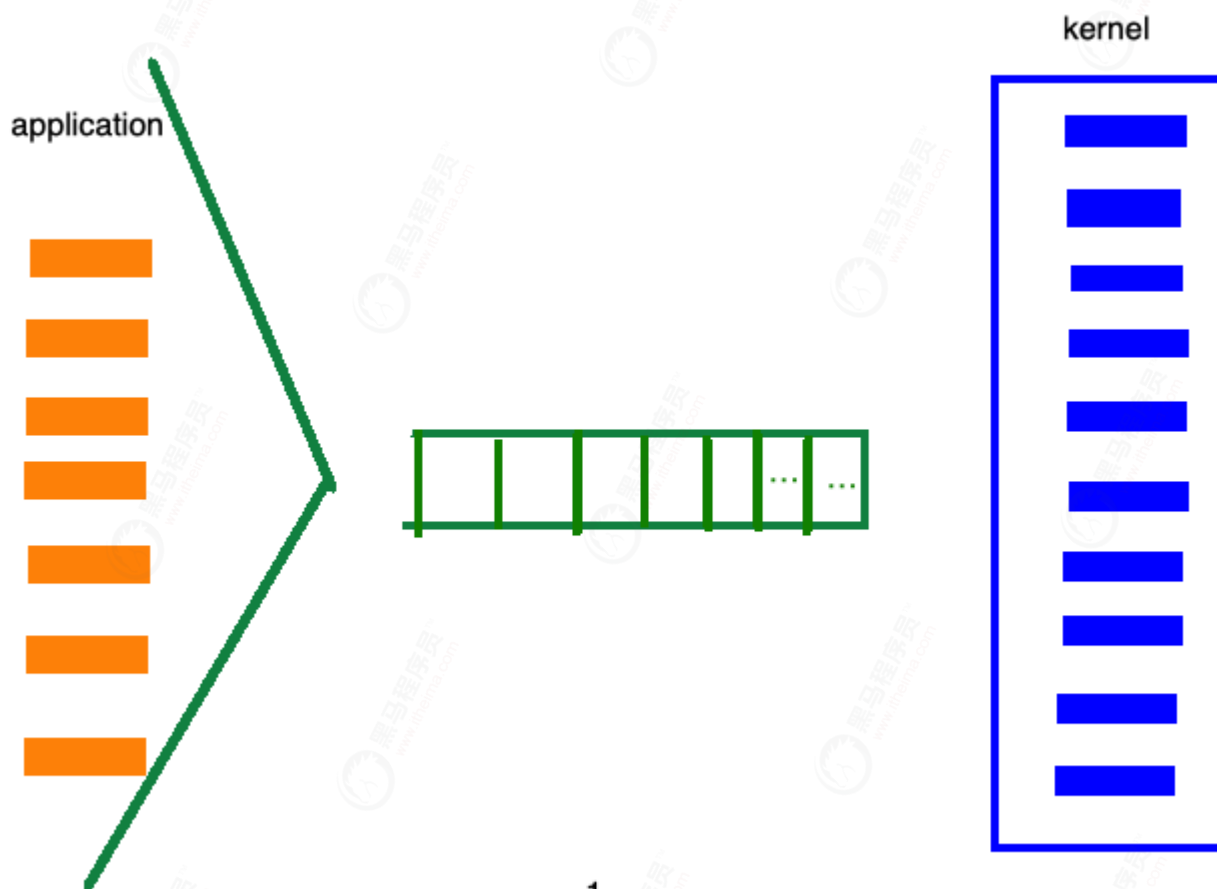


在基于select的 I/O 复用模型中，会用到 Select，这个函数也会使进程阻塞，但是和阻塞 I/O 所不同的是这两个函数可以同时多个 I/O 操作。而且可以同时多个读操作，多个写操作的 I/O 函数进行检测，直到有数据可读或可写时，才真正调用 I/O 操作函数。

select/poll处理流程:



epoll的处理流程:



1

epoll 可以说是I/O 多路复用最新的一个实现，epoll 修复了poll 和select绝大部分问题, 比如

epoll 是线程安全的。epoll 不仅告诉你sock组里面的数据，还会告诉你具体哪个sock连接有数据，不用进程独自轮询查找。

- 为什么阻塞 IO 和 IO 多路复用最为常用？

在实际的网络 IO 的应用中，需要的是系统内核的支持以及编程语言的支持。现在大多数系统内核都会支持阻塞 IO、非阻塞 IO 和 IO 多路复用，但像信号驱动 IO、异步 IO，只有高版本的 **Linux** 系统内核才会支持。

- **RPC** 框架应该采用哪种网络 IO 模型？

1) IO 多路复用应用特点：

IO 多路复用更适合高并发的场景，可以用较少的进程（线程）处理较多的 socket 的 IO 请求，但使用难度比较高。

2) 阻塞 IO应用特点：

与 IO 多路复用相比，阻塞 IO 每处理一个 socket 的 IO 请求都会阻塞进程（线程），但使用难度较低。

3) RPC框架应用：

RPC 调用在大多数的情况下，是一个高并发调用的场景，在 RPC 框架的实现中，一般会选择 **IO** 多路复用的方式。

2.4.6 零拷贝

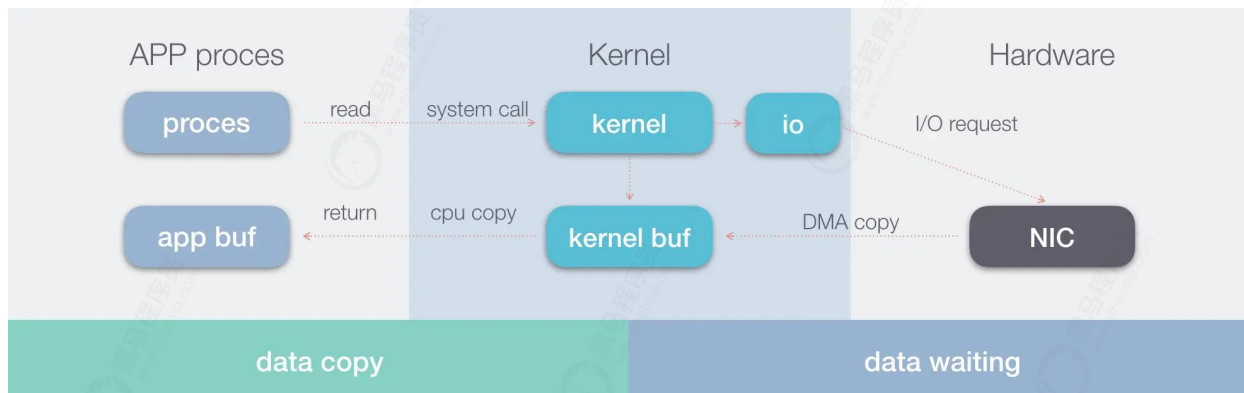
- 什么是零拷贝

系统内核处理 IO 操作分为两个阶段：等待数据和拷贝数据。

等待数据，就是系统内核在等待网卡接收到数据后，把数据写到内核中。

拷贝数据，就是系统内核在获取到数据后，将数据拷贝到用户进程的空间中。

具体流程：



所谓的零拷贝，就是取消用户空间与内核空间之间的数据拷贝操作，应用进程每一次的读写操作，都可以通过一种方式，让应用进程向用户空间写入或者读取数据，就如同直接向内核空间写入或者读取数据一样，再通过 DMA 将内核中的数据拷贝到网卡，或将网卡中的数据 copy 到内核。

• RPC框架中的零拷贝应用

Netty 框架是否也有零拷贝机制？

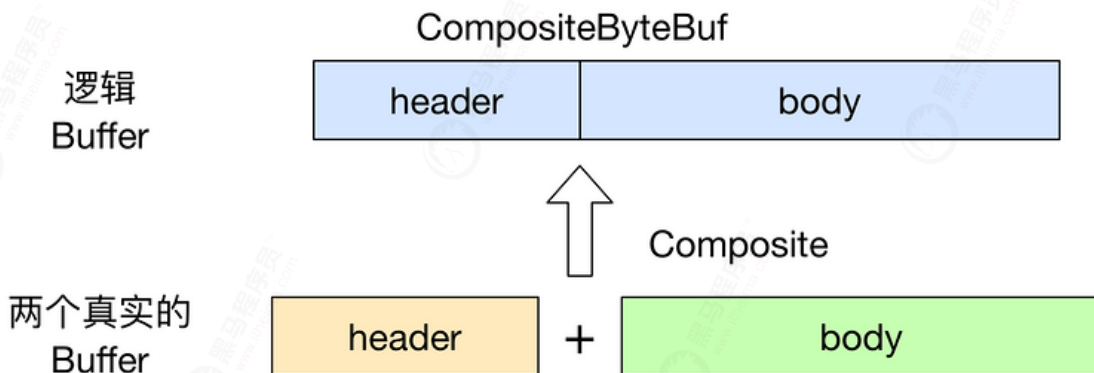
Netty 的零拷贝则有些不一样，他完全站在了用户空间上，也就是基于 JVM 之上。

Netty 当中的零拷贝是如何实现的？

RPC 并不会把请求参数作为一个整体数据包发送到对端机器上，中间可能会拆分，也可能会合并其他请求，所以消息都需要有边界。接收到消息之后，需要对数据包进行处理，根据边界对数据包进行分割和合并，最终获得完整的消息。

Netty 零拷贝主要体现在三个方面：

- Netty 的接收和发送 ByteBuffer 是采用 DIRECT BUFFERS，使用堆外的直接内存（内存对象分配在 JVM 中堆以外的内存）进行 Socket 读写，不需要进行字节缓冲区的二次拷贝。如果采用传统堆内存（HEAP BUFFERS）进行 Socket 读写，JVM 会将堆内存 Buffer 拷贝一份到直接内存中，然后写入 Socket 中。
- Netty 提供了组合 Buffer 对象，也就是 CompositeByteBuffer 类，可以将 ByteBuffer 分解为多个共享同一个存储区域的 ByteBuffer，避免了内存的拷贝。



- Netty 的文件传输采用了 FileRegion 中包装 NIO 的 FileChannel.transferTo() 方法，它可以直接将文件缓冲区的数据发送到目标 Channel，避免了传统通过循环 write 方式导致的内存拷贝问题。

零拷贝带来的作用就是避免没必要的 CPU 拷贝，减少了 CPU 在用户空间与内核空间之间的上下文切换，从而提升了网络通信效率与应用程序的整体性能。

2.4.7 时间轮

- 为什么需要时间轮？

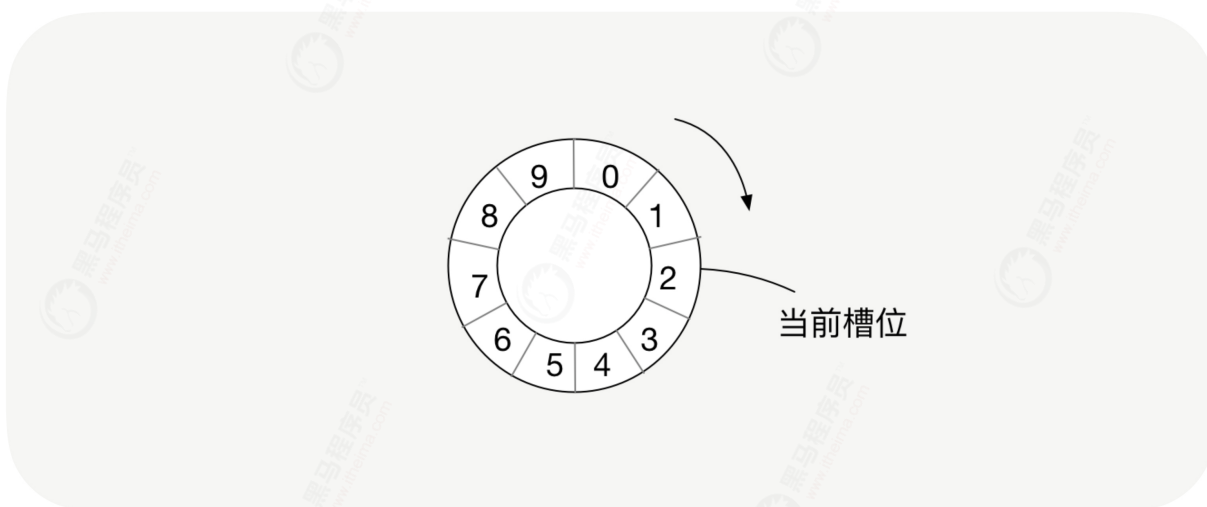
在Dubbo中，为增强系统的容错能力，会有相应的监听判断处理机制。比如RPC调用的超时机制的实现，消费者判断RPC调用是否超时，如果超时会将超时结果返回给应用层。在Dubbo最开始的实现中，是将所有的返回结果（DefaultFuture）都放入一个集合中，并且通过一个定时任务，每隔一定时间间隔就扫描所有的future，逐个判断是否超时。

这样的实现方式虽然比较简单，但是存在一个问题就是会有很多无意义的遍历操作开销。比如一个RPC调用的超时时间是10秒，而设置的超时判定的定时任务是2秒执行一次，那么可能会有4次左右无意义的循环检测判断操作。

为了解决上述场景中的类似问题，Dubbo借鉴Netty，引入了时间轮算法，减少无意义的轮询判断操作。

- 时间轮原理

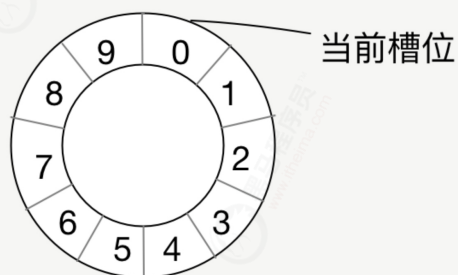
对于以上问题，目的是要减少额外的扫描操作就可以了。比如说一个定时任务是在5秒之后执行，那么在4.9秒之后才扫描这个定时任务，这样就可以极大减少CPU开销。这时我们就可以利用时钟轮的机制了。



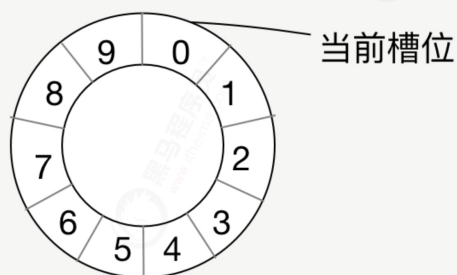
时钟轮的实质上是参考了生活中的时钟跳动的原理，那么具体是如何实现呢？

在时钟轮机制中，有时间槽和时钟轮的概念，时间槽就相当于时钟的刻度；而时钟轮就相当于指针跳动的一个周期，我们可以将每个任务放到对应的时间槽位上。

如果时钟轮有 10 个槽位，而时钟轮一轮的周期是 10 秒，那么我们每个槽位的单位时间就是 1 秒，而下一层时间轮的周期就是 100 秒，每个槽位的单位时间也就是 10 秒，这就好比秒针与分针，在秒针周期下，刻度单位为秒，在分针周期下，刻度为分。

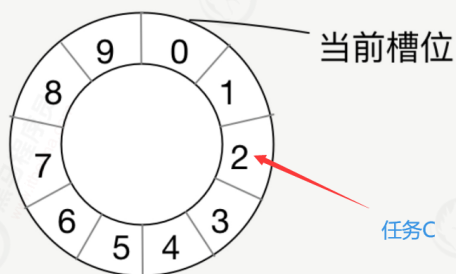


下一层时钟轮：周期100秒

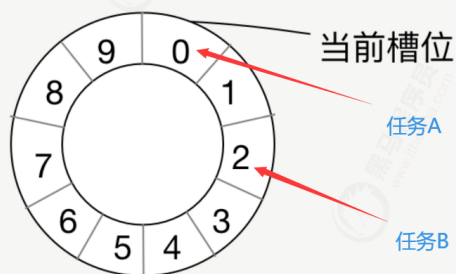


当前时钟轮：周期10秒

假设现在我们有 3 个任务，分别是任务 A（0.9秒之后执行）、任务 B（2.1秒后执行）与任务 C（12.1秒之后执行），我们将这 3 个任务添加到时钟轮中，任务 A 被放到第 0 槽位，任务 B 被放到第 2槽位，任务 C 被放到下一层时间轮的第2个槽位，如下图所示：



下一层时钟轮：周期100秒



当前时钟轮：周期10秒

通过这个场景我们可以了解到，时钟轮的扫描周期仍是最小单位1秒，但是放置其中的任务并没有反复扫描，每个任务会按要求只扫描执行一次，这样就能够很好的解决CPU 浪费的问题。

Dubbo中的时间轮原理是如何实现？

主要是通过Timer，Timeout，TimerTask几个接口定义了一个定时器的模型，再通过HashedWheelTimer这个类实现了一个时间轮定时器（默认的时间槽的数量是512，可以自定义这个值）。它对外提供了简单易用的接口，只需要调用newTimeout接口，就可以实现对只需执行一次任务的调度。通过该定时器，Dubbo在响应的场景中实现了高效的任务调度。

• 时间轮在RPC的应用

- 调用超时：上面所讲的客户端调用超时的处理，就可以应用到时钟轮，我们每发一次请求，都创建一个处理请求超时的定时任务放到时钟轮里，在高并发、高访问量的情况下，时钟轮每次只轮询一个时间槽位中的任务，这样会节省大量的 CPU。
- 启动加载：调用端与服务端启动也可以应用到时钟轮，比如说在服务启动完成之后要去加载缓存，执行定时任务等，都可以放在时钟轮里。

- 定时心跳检测：RPC 框架调用端定时向服务端发送的心跳检测，来维护连接状态，我们可以将心跳的逻辑封装为一个心跳任务，放到时钟轮里。心跳是要定时重复执行的，而时钟轮中的任务执行一遍就被移除了，对于这种需要重复执行的定时任务我们该如何处理呢？我们在定时任务逻辑结束的最后，再加上一段逻辑，重设这个任务的执行时间，把它重新丢回到时钟轮里。这样就可以实现循环执行。

3. RPC的高级运用

3.1 基于Dubbo分布式高并发场景下的运用要点

3.1.1 异步处理机制

- 为什么要采用异步？

如果采用同步调用，CPU 大部分的时间都在等待而没有去计算，从而导致 CPU 的利用率不够。

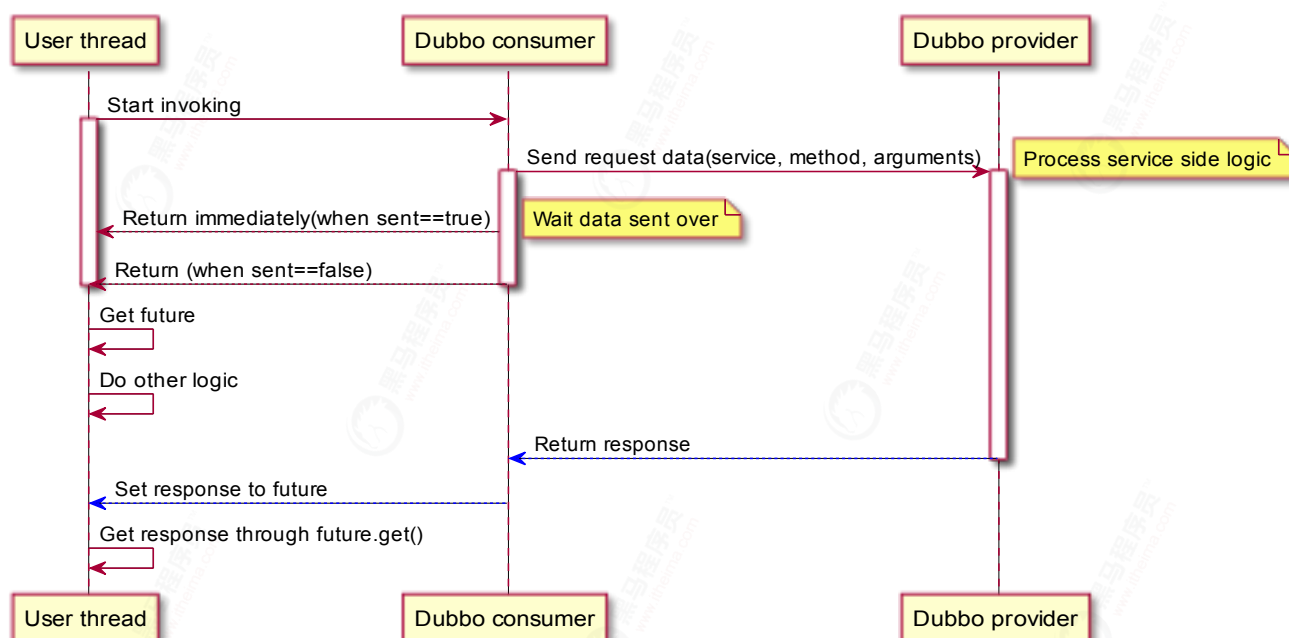
RPC 请求比较耗时的原因主要是在哪里？

在大多数情况下，RPC 本身处理请求的效率是在毫秒级的。RPC 请求的耗时大部分都是业务耗时。

- 调用端如何实现异步？

常用的方式就是Future 方式，它是返回 Future 对象，通过GET方式获取结果；或者采用入参为 Callback 对象的回调方式，处理结果。

基于RPC的DUBBO框架是如何实现异步调用呢？



- 服务端如何实现异步？

为了提升性能，连接请求与业务处理不会放在一个线程处理，这个就是服务端的异步化。服务端业务处理逻辑加入异步处理机制。

在RPC 框架提供一种回调方式，让业务逻辑可以异步处理，处理完之后调用 RPC 框架的回调接口。

RPC 框架的异步策略主要是调用端异步与服务端异步。调用端的异步就是通过 Future 方式。

服务端异步则需要一种回调方式，让业务逻辑可以异步处理。这样就实现了RPC调用的全异步化。

3.1.2 路由与负载均衡

- 为什么要采用路由？

真实的环境中一般是以集群的方式提供服务，对于服务调用方来说，一个接口会有多个服务提供方同时提供服务，所以 RPC 在每次发起请求的时候，都需要从多个服务节点里面选取一个用于处理请求的服务节点。

这就需要在RPC应用中增加路由功能。

- 如何实现路由？

服务注册发现方式：

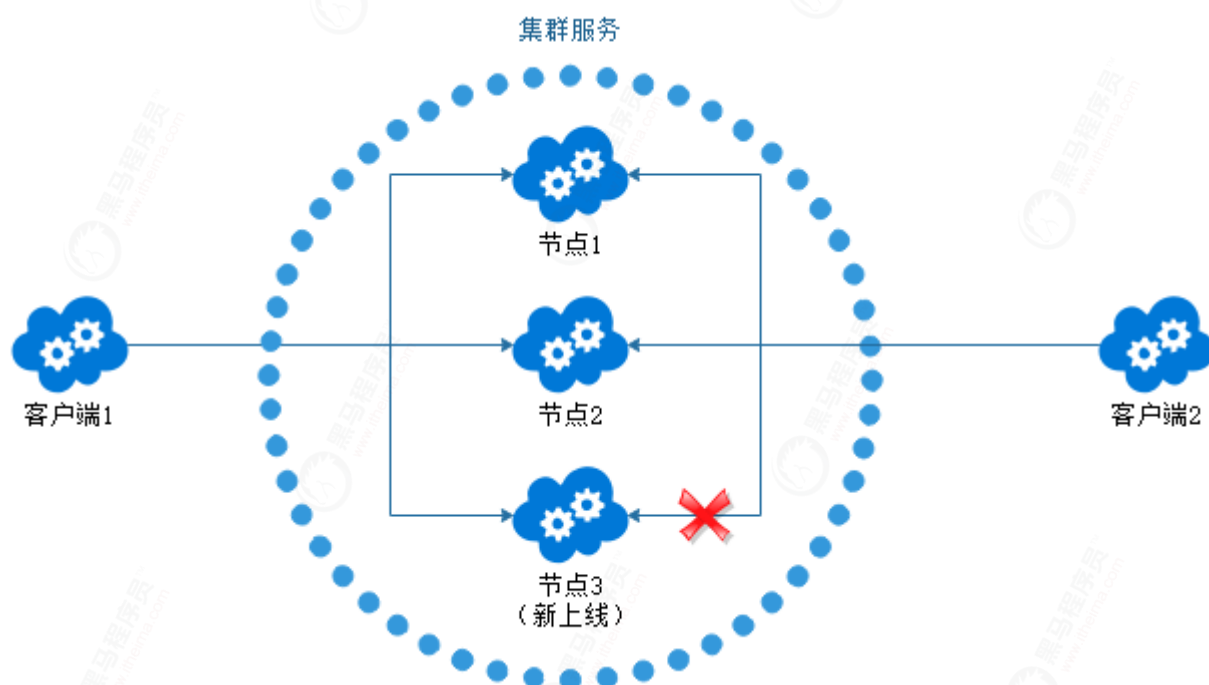
通过服务发现的方式从逻辑上看是可行，但注册中心是用来保证数据的一致性。通过服务发现方式来实现请求隔离并不理想。

RPC路由策略：

从服务提供方节点集合里面选择一个合适的节点（负载均衡），把符合我们要求的节点筛选出来。这个就是路由策略：

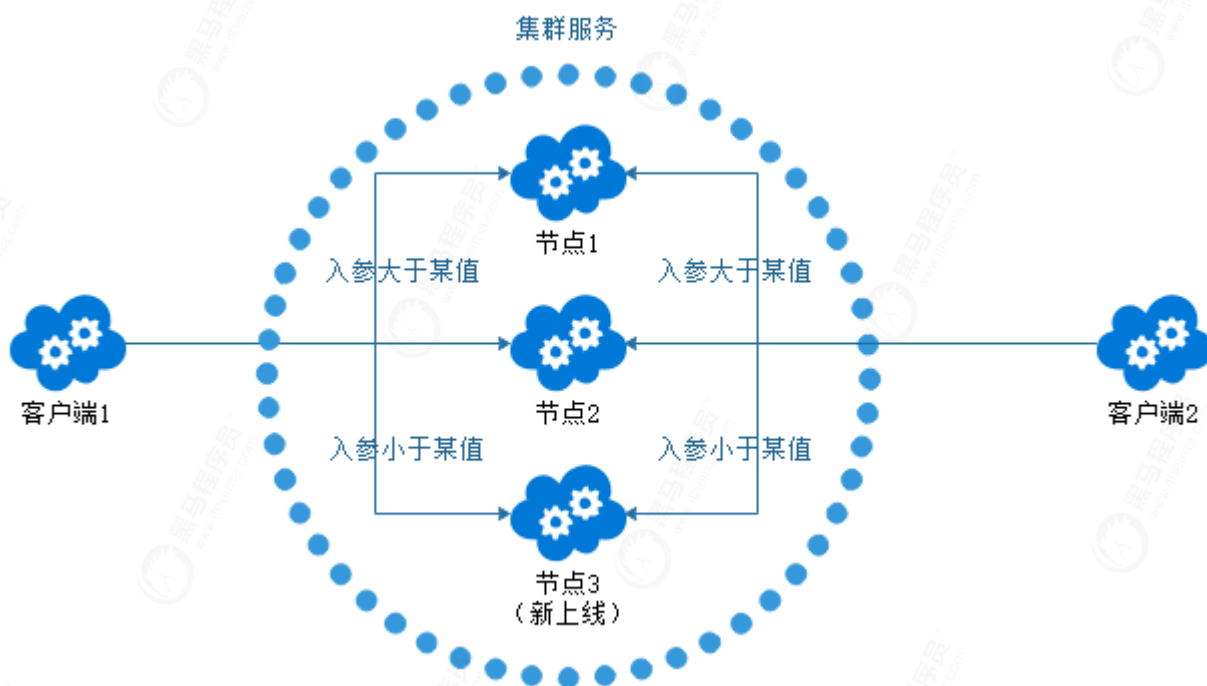
接收请求-->请求校验-->路由策略-->负载均衡-->

使用了 IP 路由策略后，整个集群的调用拓扑如下图所示：



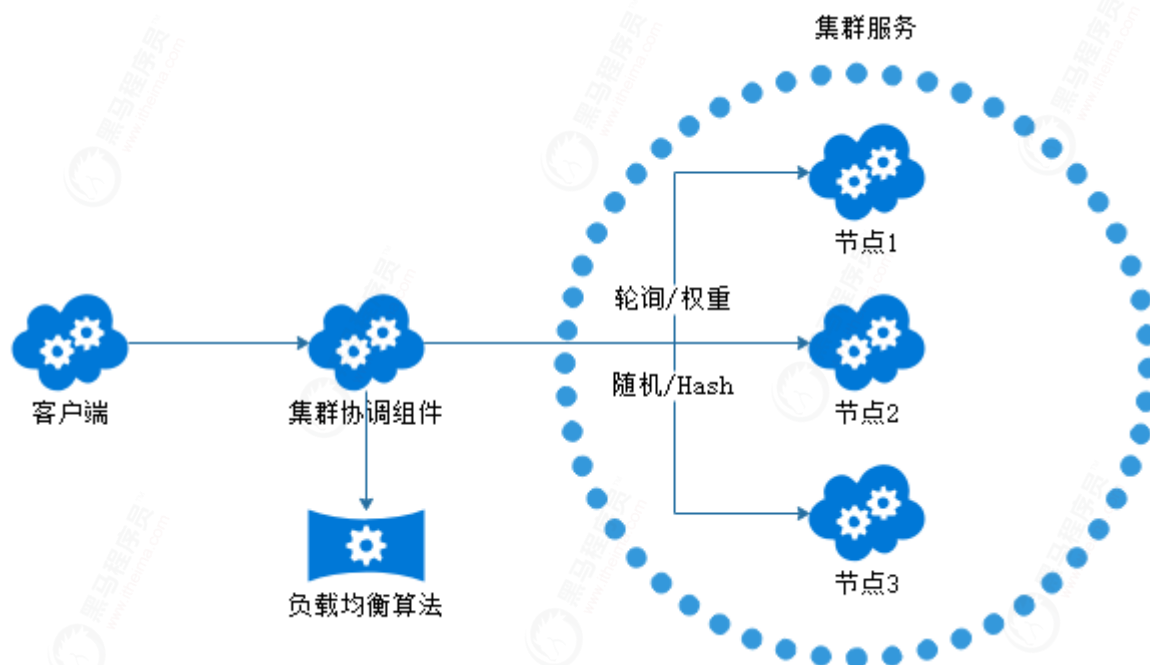
有些场景下，可能还需要更细粒度的路由方式，比如说根据SESSIONID要落到相同的服务节点上以保持会话的有效性；

可以考虑采用参数化路由：



- **RPC框架中的负载均衡**

RPC 的负载均衡是由 RPC 框架自身提供实现，自主选择一个最佳的服务节点，发起 RPC 调用请求。



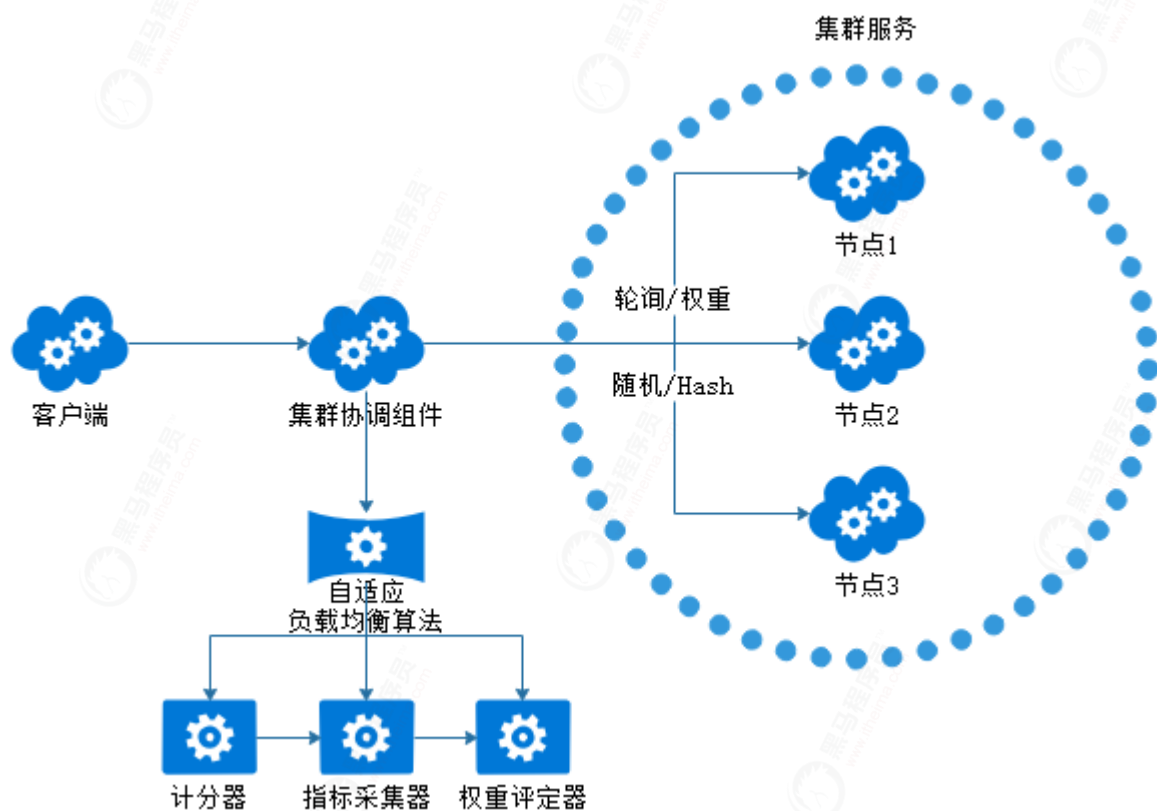
RPC 负载均衡策略一般包括轮询、随机、权重、最少连接等。Dubbo默认就是使用随机负载均衡策略。

- **自适应的负载均衡策略**

RPC 的负载均衡完全由 RPC 框架自身实现，通过所配置的负载均衡组件，自主选择合适服务节点。这个就是自适应的负载均衡策略。

具体如何实现？

这就需要判定服务节点的处理能力。



主要步骤：

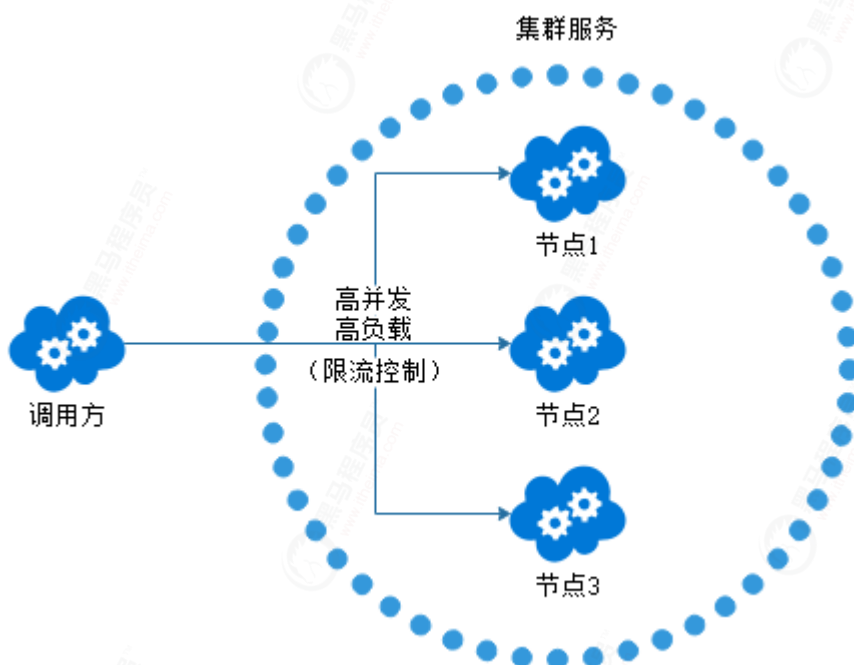
- (1) 添加计分器和指标采集器。
- (2) 指标采集器收集服务节点 CPU 核数、CPU 负载以及内存占用率等指标。
- (3) 可以配置开启哪些指标采集器，并设置这些参考指标的具体权重。
- (4) 通过对服务节点的综合打分，最终计算出服务节点的实际权重，选择合适的服务节点。

3.1.3 熔断限流

- 为什么要进行限流？

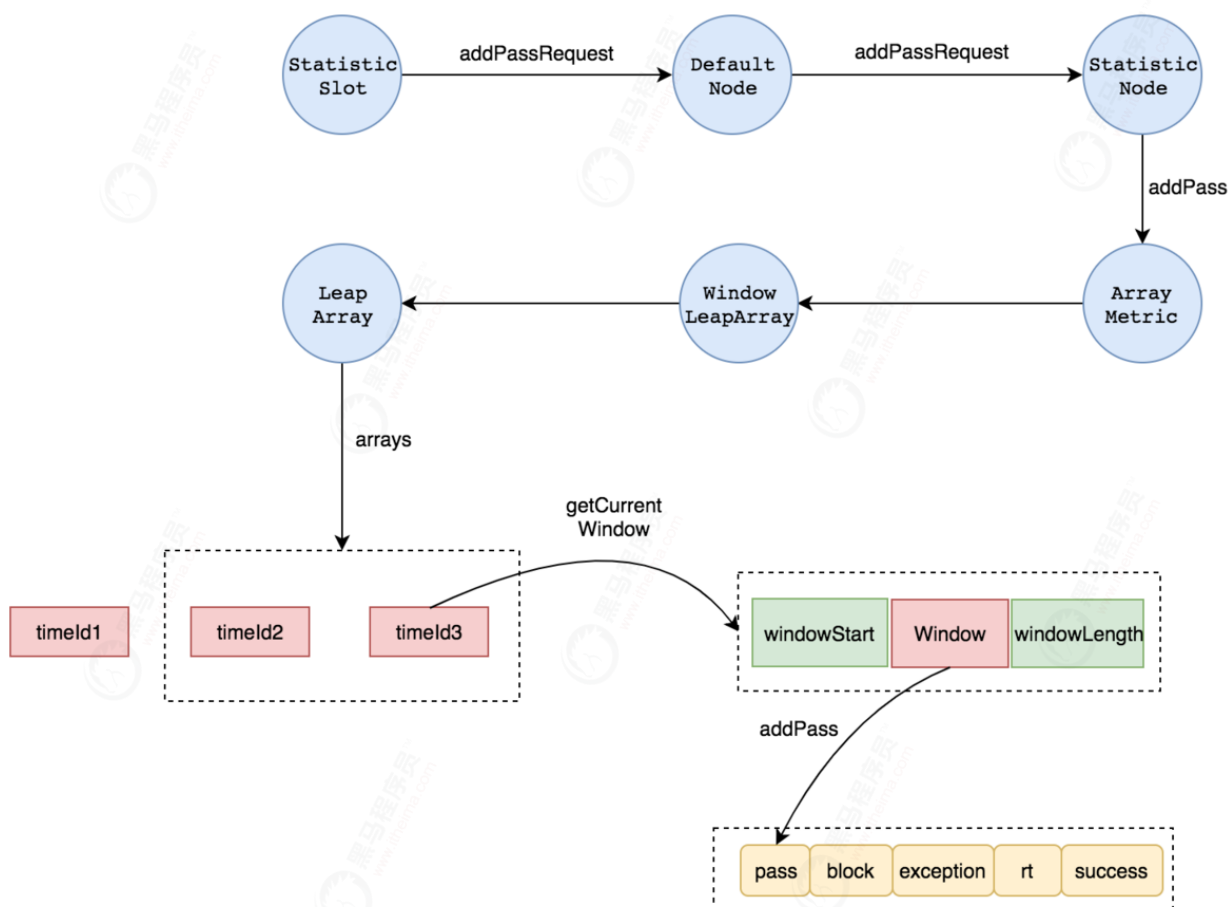
在实际生产环境中，每个服务节点都可能由于访问量过大而引起一系列问题，就需要业务提供方能够进行自我保护，从而保证在高访问量、高并发的场景下，系统依然能够稳定，高效运行。

- 服务端的自我保护实现



在Dubbo框架中，可以通过Sentinel来实现更为完善的熔断限流功能，服务端是具体如何实现限流逻辑的？

方法有很多种，最简单的是计数器，还有平滑限流的滑动窗口、漏斗算法以及令牌桶算法等等。Sentinel采用的是滑动窗口来实现的限流。

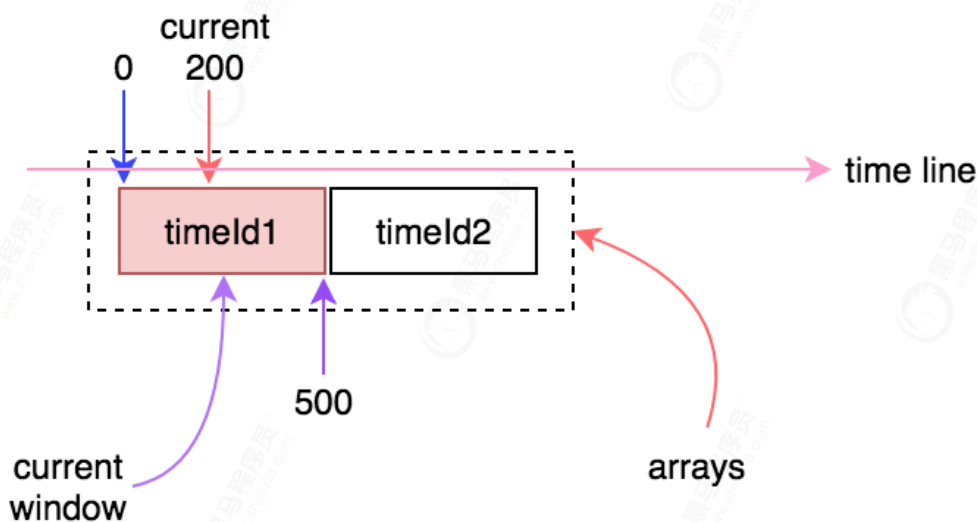


windowStart: 时间窗口的开始时间，单位是毫秒

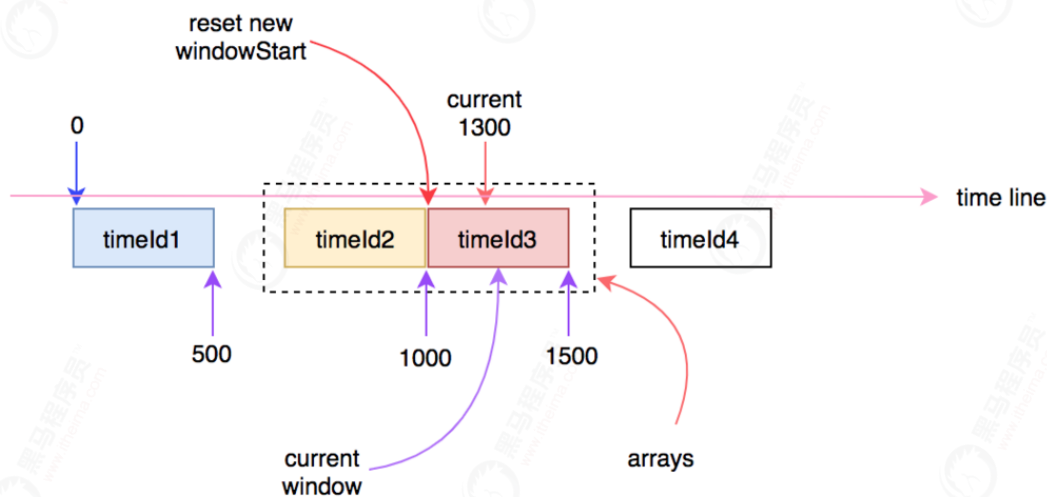
windowLength: 时间窗口的长度，单位是毫秒

value: 时间窗口的内容

初始的时候arrays数组中只有一个窗口，每个时间窗口的长度是**500ms**，这就意味着只要当前时间与时间窗口的差值在500ms之内，时间窗口就不会向前滑动。



时间继续往前走，当超过500ms时，时间窗口就会向前滑动到下一个，这时就会更新当前窗口的开始时间：

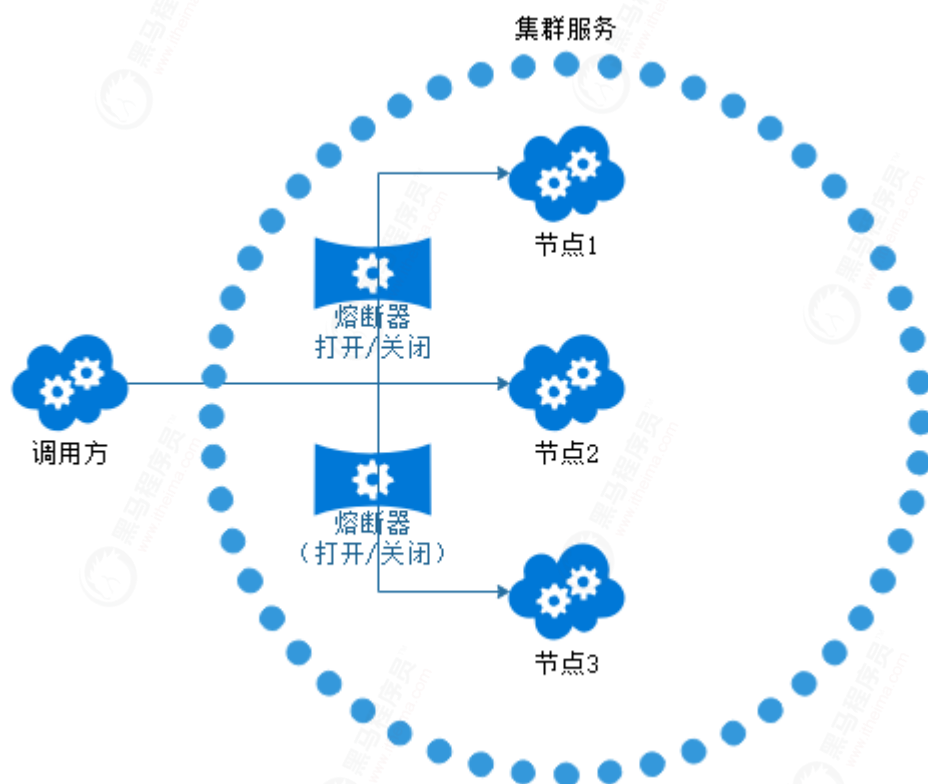


在当前时间点中进入的请求，会被统计到当前时间所对应的时间窗口中。

- 调用方的自我保护

一个服务 A 调用服务 B 时，服务 B 的业务逻辑又调用了服务 C，这时服务 C 响应超时，服务 B 就可能会因为堆积大量请求而导致服务宕机，由此产生服务雪崩的问题。

熔断处理流程：



熔断机制：

熔断器的工作机制主要是关闭、打开和半打开这三个状态之间的切换。

Sentinel 熔断降级组件它可以支持以下降级策略：

- 平均响应时间 (`DEGRADE_GRADE_RT`)：当 1s 内持续进入 N 个请求，对应时刻的平均响应时间（秒级）均超过阈值（ `count` ，以 ms 为单位），那么在接下的时间窗口（ `DegradeRule` 中的 `timeWindow` ，以 s 为单位）之内，对这个方法的调用都会自动地熔断（抛出 `DegradeException` ）。注意 Sentinel 默认统计的 RT 上限是 4900 ms，超出此阈值的都会算作 4900 ms，若需要变更此上限可以通过启动配置项 `-Dcsp.sentinel.statistic.max.rt=xxx` 来配置。
- 异常比例 (`DEGRADE_GRADE_EXCEPTION_RATIO`)：当资源的每秒请求量 $\geq N$ （可配置），并且每秒异常总数占通过量的比值超过阈值（ `DegradeRule` 中的 `count` ）之后，资源进入降级状态，即在接下的时间窗口（ `DegradeRule` 中的 `timeWindow` ，以 s 为单位）之内，对这个方法的调用都会自动地返回。异常比率的阈值范围是 `[0.0, 1.0]`，代表 0% - 100%。
- 异常数 (`DEGRADE_GRADE_EXCEPTION_COUNT`)：当资源近 1 分钟的异常数目超过阈值之后会进行熔断。注意由于统计时间窗口是分钟级别的，若 `timeWindow` 小于 60s，则结束熔断状态后仍可能再进入熔断状态。

更多资料，参考[Sentinel官方文档](#)。

3.1.4 优雅启动

• 什么是启动预热

启动预热就是让刚启动的服务，不直接承担全部的流量，而是让它随着时间的移动慢慢增加调用次数，最终让流量缓和运行一段时间后达到正常水平。

• 如何实现

首先要知道服务提供方的启动时间，有两种获取方法：

- 一种是服务提供方在启动的时候，主动将启动的时间发送给注册中心；
- 另一种就是注册中心来检测，将服务提供方的请求注册时间作为启动时间。

调用方通过服务发现获取服务提供方的启动时间， 然后进行降权， 减少被负载均衡选择的概率， 从而实现预热的过程。

在Dubbo框架中也引入了"warmup"特性， 核心源码是

在"**com.alibaba.dubbo.rpc.cluster.loadbalance.AbstractLoadBalance.java**"中：

```
protected int getWeight(Invoker<?> invoker, Invocation invocation) {
    // 先得到Provider的权重
    int weight = invoker.getUrl().getMethodParameter(invocation.getMethodName(),
        Constants.WEIGHT_KEY, Constants.DEFAULT_WEIGHT);
    if (weight > 0) {
        // 得到provider的启动时间戳
        long timestamp = invoker.getUrl().getParameter(Constants.REMOTE_TIMESTAMP_KEY, 0L);
        if (timestamp > 0L) {
            // provider已经运行时间
            int uptime = (int) (System.currentTimeMillis() - timestamp);
            // 得到warmup的值，默认为10分钟
            int warmup = invoker.getUrl().getParameter(Constants.WARMUP_KEY,
                Constants.DEFAULT_WARMUP);
            // provider运行时间少于预热时间，那么需要重新计算权重weight（即需要降权）
            if (uptime > 0 && uptime < warmup) {
                weight = calculateWarmupWeight(uptime, warmup, weight);
            }
        }
    }
    return weight;
}

static int calculateWarmupWeight(int uptime, int warmup, int weight) {
    // 随着provider的启动时间越来越长，慢慢提升权重weight
    int ww = (int) ( (float) uptime / ( (float) warmup / (float) weight ) );
    return ww < 1 ? 1 : (ww > weight ? weight : ww);
}
```

Dubbo2.7.3版本， 参考源码“org.apache.dubbo.rpc.cluster.loadbalance.AbstractLoadBalance”

执行策略： 如果provider运行了1分钟，那么weight为10，承担10%流量； 如果provider运行了2分钟，那么weight为20，承担20%流量； 如果provider运行了5分钟，那么weight为50，承担50%流量。

3.1.5 优雅关闭

- 为什么需要优雅关闭

调用方会存在以下情况：

- 目标服务已经下线；
- 目标服务正在关闭中。

- 如何实现优雅关闭

关闭的流程：



解决方案:

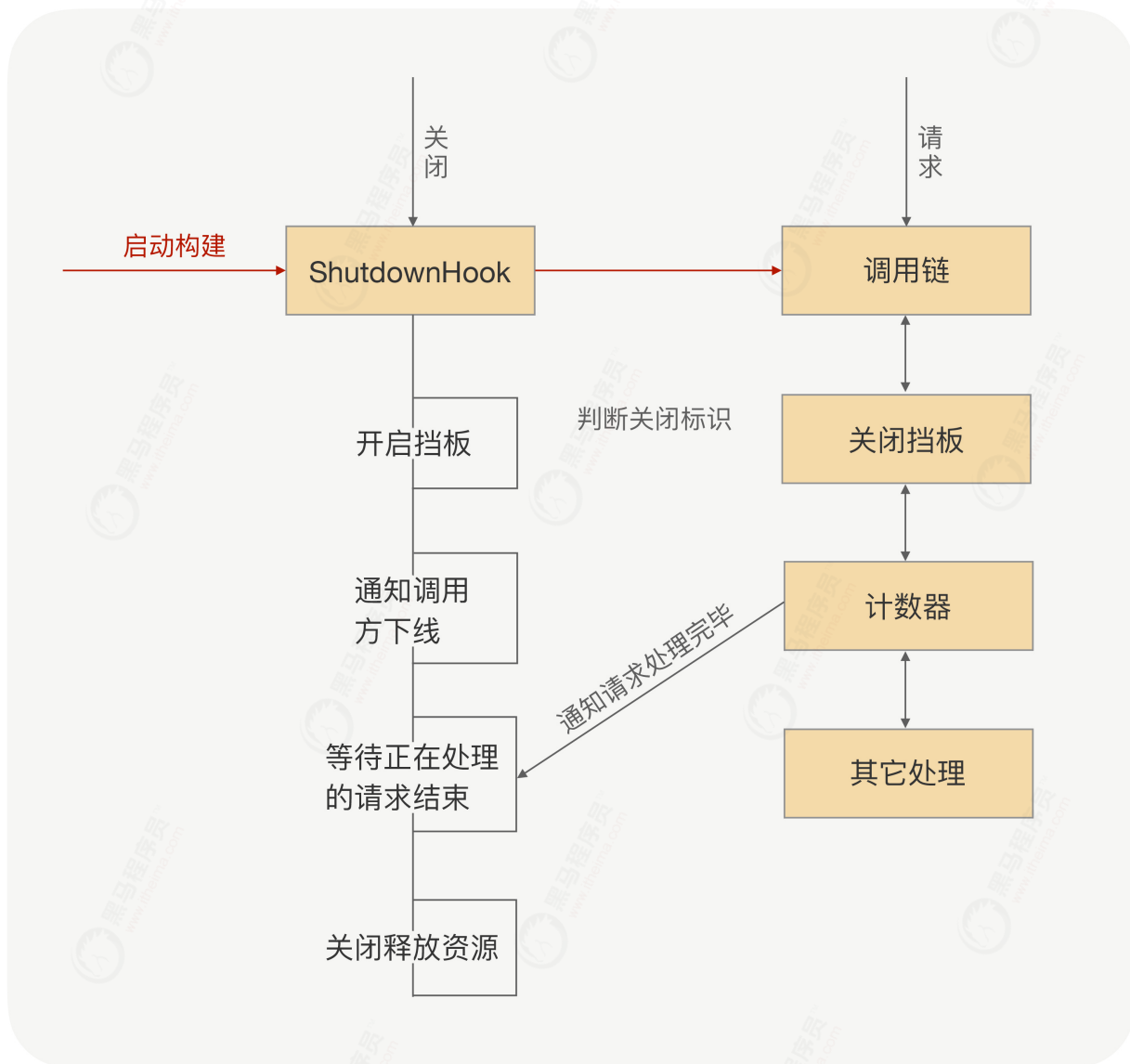
设置请求“挡板”，挡板的作用就是告诉调用方，服务提供方已经开始进入关闭流程了，不能再处理其他请求了。

具体处理流程:

当服务提供方正在关闭，可以直接返回一个特定的异常给调用方。然后调用方把这个节点从健康列表挪出，并把其他请求自动重试到其他节点。如需更为完善，可以再加上主动通知机制。

如何捕获关闭事件呢？

Java应用程序，在接收到结束信号时，会调用`Runtime.addShutdownHook`方法触发关闭钩子。



在Dubbo框架中，在以下场景中会触发优雅关闭：

JVM主动关闭(`System.exit(int)`)；JVM由于资源问题退出(`OOM`)；应用程序接受到进程正常结束信号：`SIGTERM` 或 `SIGINT` 信号。

优雅停机是默认开启的，停机等待时间为10秒。可以通过配置 `dubbo.service.shutdown.wait` 来修改等待时间。

Dubbo 推出了多段关闭的方式来保证服务完全无损。

3.2 RPC线上实战经验(拓展)

3.2.1 如何去实现一个Rpc框架

1. 服务设计：客户端、服务端、ZK注册中心，获取订单接口。

怎么知道服务端的信息？如何去调用的？

2. 先启动服务端：将接口信息注册至ZK。（`ServicePushManager.registerIntoZK`方法）

3. 启动客户端：从ZK拉取服务端接口信息。（`ServicePullManager.pullServiceFromZK`方法）

4. Rpc调用处理流程:

客户端->通过动态代理调用服务端接口 (ProxyHelper.doIntercept) -> 选取不同的调用策略

-> 异步方式调用 (通过MAP存储记录channel, rpcRequestPool.fetchResponse获取结果)

-> 服务端 (根据请求信息调用对应的接口, RpcRequestHandler.channelRead0)

-> 客户端监听接收结果(RpcResponseHandler.channelRead0)

-> 关闭连接 (RpcRequestManager.destroyChannelHolder关闭连接)

拓展课题: 改造netty为长连接通讯方式; 关闭连接的时候最好在finally里面执行

3.2.2 流量复制回放

- 流量复制的作用

最好的测试方式就是采用线上流量来验证, 通过流量复制, 能够有效进行测试验证, 最大限度的保障了服务的稳定性。

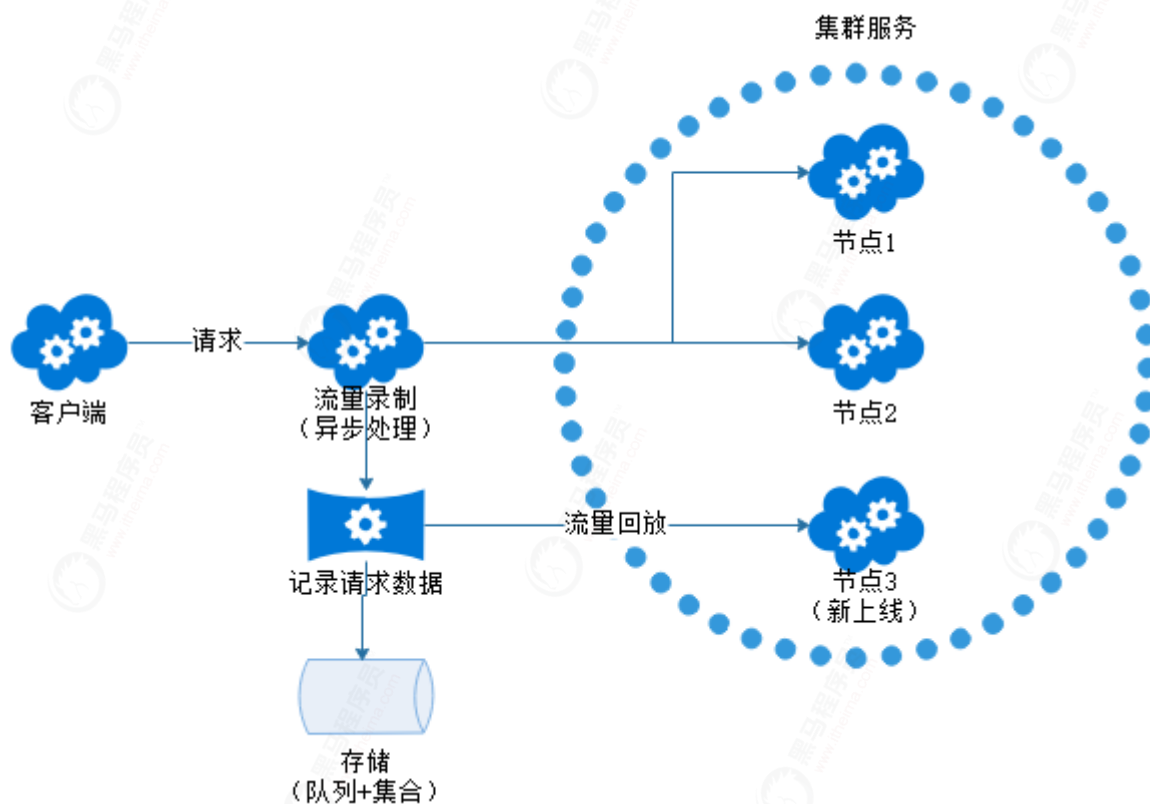
- RPC应用中如何支持流量复制

常见的方案有很多种, 比如基于TcpCopy的Dubbo服务引流、或者通过Nginx的Mirror模块来实现流量复制。

RPC内部如何去实现流量复制呢?

我们在 RPC 通讯当中, 将每次请求的出入参数, 以异步线程方式旁录下来, 做持久化处理, 实现流量的录制功能。

流量是先录制再回放, 如何完全模拟线上实际的高并发场景呢? 重点在于流量录制的数据存储结构。



3.2.3 动态分组

- 为什么需要动态分组？

因为非核心业务的调用量突然增长，导致整个集群变得不可用。把整个大集群根据不同的调用方来划分出不同的小集群，从而实现调用方流量隔离的效果，保障业务之间不会互相影响。

- 如何分组？

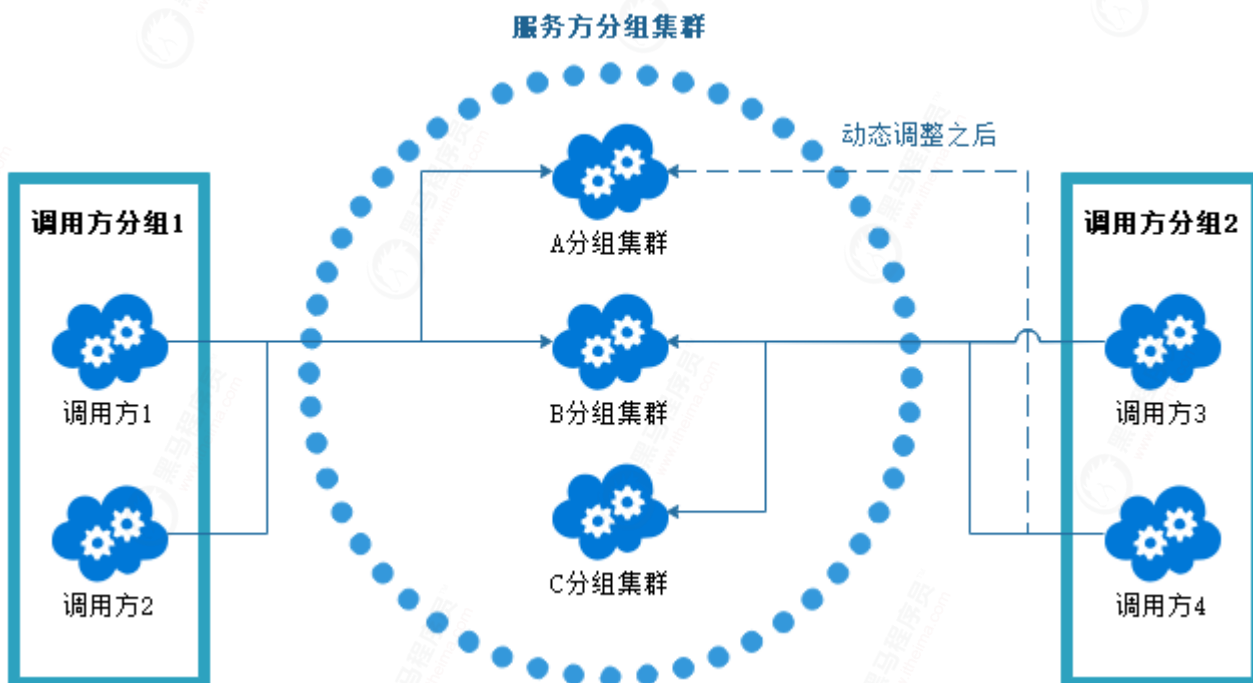
可以按照以下原则进行分组：

按照应用的重要级别来划分，让非核心业务应用跟核心业务应用划分在不同分组内。按照解耦，分离式的原则去做。

- 如何动态分组？

通过修改注册中心的数据来解决这个问题：

把注册中心里面的部分实例的别名做修改，然后通过服务发现功能去调用不同服务提供方的实例集合。



3.2.4 保障调用安全

- 为什么需要保障安全？

RPC的调用会先由服务提供方定义好一个接口，通过 RPC 提供的 **API** 进行调用，这里面其实存在一个安全隐患问题，只要拿到了 Jar 包，就可以通过私服的 Jar 引入到项目中完成 RPC的调用，所以需要有一个安全机制，能够确保调用方的合法身份。

- 如何解决调用安全问题？

解决办法只需要给每个调用方设定一个唯一的身份，只有登记过的调用方才能继续放行，没有登记过的调用方一律拒绝。

- 实现方案

采用类似JWT的签名方案或者采用加密算法约定私钥，对请求数据进行加密处理，由调用双方完自身完成鉴权处理。

- 解决伪造服务方问题

建立一个白名单机制，将开放合法的服务方IP和端口纳入白名单中，有两种处理方案：

1. 白名单可以存放在注册中心中。
2. 由服务端开辟一个白名单专用检测线程，如有不合法节点，发出预警并将其剔除。