Dubbo的出现背景

大规模服务化对于服务治理的要求

我认为到目前为止，还只是满足了通信的基础需求，但是当企业开始大规模的服务化以后，远程通信带来的弊端就越来越明显了。比如说

1. 服务链路变长了，如何实现对服务链路的跟踪和监控呢?
2. 服务的大规模集群使得服务之间需要依赖第三方注册中心来解决服务的发现和服务的感知问题
3. 服务通信之间的异常，需要有一种保护机制防止一个节点故障引发大规模的系统故障，所以要有容错机制
4. 服务大规模集群会是的客户端需要引入负载均衡机制实现请求分发

而这些对于服务治理的要求，传统的RPC技术在这样的场景中显得有点力不从心，因此很多企业开始研发自己的RPC框架，比如阿里的HSF、Dubbo；京东的JSF框架、当当的dubbox、新浪的motan、蚂蚁金服的sofa等等

有技术输出能力的公司，都会研发适合自己场景的rpc框架，要么是从0到1开发，要么是基于现有的思想结合公司业务特色进行改造。而没有技术输出能力的公司，遇到服务治理的需求时，会优先选择那些比较成熟的开源框架。而Dubbo就是其中一个

dubbo主要是一个分布式服务治理解决方案，那么什么是服务治理？服务治理主要是针对大规模服务化以后，服务之间的路由、负载均衡、容错机制、服务降级这些问题的解决方案，而Dubbo实现的不仅仅是远程服务通信，并且还解决了服务路由、负载、降级、容错等功能。

Dubbo的发展历史

Dubbo是阿里巴巴内部使用的一个分布式服务治理框架，2012年开源，因为Dubbo在公司内部经过了很多的验证相对来说比较成熟，所以在很短的的还是件就被很多互联网公司使用，再加上阿里出来的很多技术大牛进入各个创业公司担任技术架构以后，都以Dubbo作为主推的RPC框架使得dubbo很快成为了很多互联网公司的首要选择。并且很多公司在应用dubbo时，会基于自身业务特性进行优化和改进，所以也衍生了很多版本，比如京东的JSF、比如新浪的Motan、比如当当的dubbox.

在2014年10月份，Dubbo停止了维护。后来在2017年的9月份，阿里宣布重启Dubbo，并且对于Dubbo做好了长期投入的准备，并且在这段时间Dubbo进行了非常多的更新，目前的版本已经到了2.7.

2018年1月8日，Dubbo创始人之一梁飞在Dubbo交流群里透露了Dubbo 3.0正在动工的消息。Dubbo 3.0内核与Dubbo2.0完全不同，但兼容Dubbo 2.0。Dubbo 3.0将支持可选Service Mesh

2018年2月份， Dubbo捐给了Apache。另外，阿里巴巴对于Spring Cloud Alibaba生态的完善，以及Spring Cloud团队对于alibaba整个服务治理生态的支持，所以Dubbo未来依然是国内绝大部分公司的首要选择。

What Dubbo3 is

如开篇所述，Dubbo 提供了构建云原生微服务业务的一站式解决方案，可以使用 Dubbo 快速定义并发布微服务组件，同时基于 Dubbo 开箱即用的丰富特性及超强的扩展能力，构建运维整个微服务体系所需的各项服务治理能力，如 Tracing、Transaction 等，Dubbo 提供的基础能力包括：

- 服务发现

- 流式通信
- 负载均衡
- 流量治理
- .....

Dubbo 计划提供丰富的多语言客户端实现，其中 Java、Golang 版本是当前稳定性、活跃度最好的版本，其他多语言客户端[]正在持续建设中。

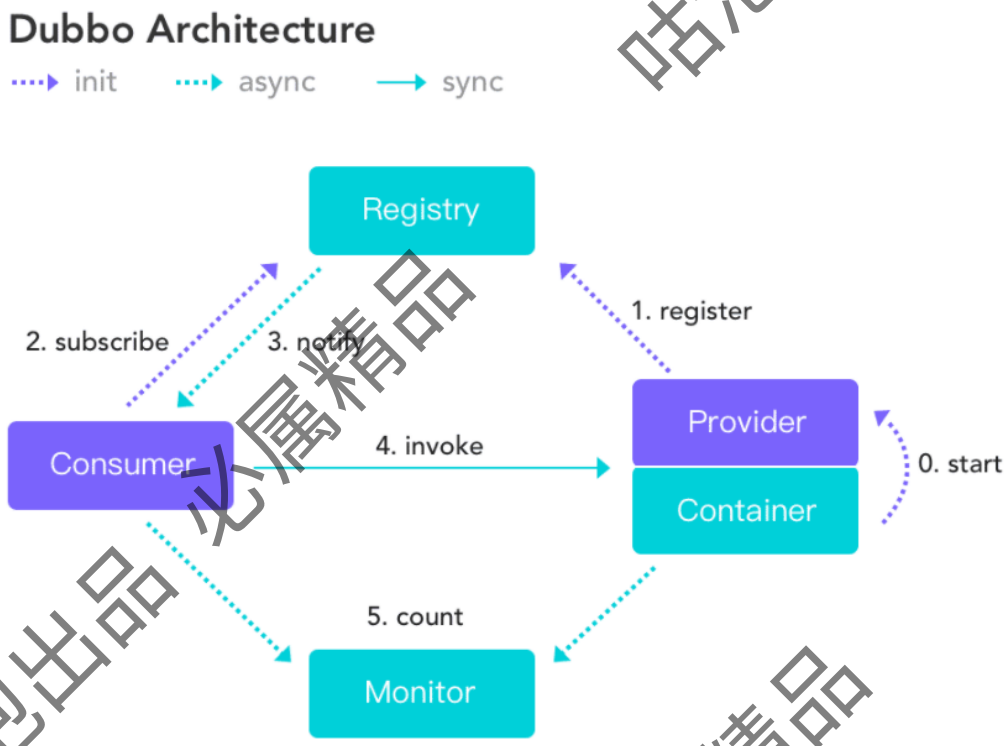
自开源以来，Dubbo 就被一众大规模互联网、IT公司选型，经过多年企业实践积累了大量经验。Dubbo3 是站在巨人肩膀上的下一代产品，它汲取了上一代的优点并针对已知问题做了大量优化，因此，Dubbo 在解决业务落地与规模化实践方面有着无可比拟的优势：

- 开箱即用
  - 易用性高，如 Java 版本的面向接口代理特性能实现本地透明调用
  - 功能丰富，基于原生库或轻量扩展即可实现绝大多数的微服务治理能力
- 超大规模微服务集群实践
  - 高性能的跨进程通信协议
  - 地址发现、流量治理层面，轻松支持百万规模集群实例
- 企业级微服务治理能力
  - 服务测试
  - 服务Mock

Dubbo3 是在云原生背景下诞生的，使用 Dubbo 构建的微服务遵循云原生思想，能更好的复用底层云原生基础设施、贴合云原生微服务架构。这体现在：

- 服务支持部署在容器、Kubernetes平台，服务生命周期可实现与平台调度周期对齐；
- 支持经典 Service Mesh 微服务架构，引入了 Proxyless Mesh 架构，进一步简化 Mesh 的落地与迁移成本，提供更灵活的选择；
- 作为桥接层，支持与 SpringCloud、gRPC 等异构微服务体系的互调互通

Dubbo的整体架构



Dubbo的使用

首先，构建两个maven项目

- dubbo-p
- dubbo-c
- dubbo-api

dubbo-api

dubbo-api提供服务的公共契约，里面提供了对外的服务。

```
public interface UserService {
    String queryUser(String var1);

    void doKill(String var1);
}
```

dubbo-p

在dubbo-p服务中，提供UserService的实现

```
@DubboService(methods = {@Method(name = "doKill",executes = 10,actives = 10)},connections =
10,protocol = "dubbo")
public class UserServiceImpl implements UserService {
    @Override
```

```
    public String queryUser(String s) {
        String username = RpcContext.getContext().getAttachment("username");
        System.out.println(username);
        System.out.println("=======provider======" + s);
        return "OK--" + s;
    }

    @Override
    public void doKill(String s) {
        System.out.println("=======provider======" + s);
    }
}
```

dubbo-c

```
    @Test
    public void test1() throws InterruptedException {
        System.out.println(userService.queryUser("jack"));
    }
```

问题来了，现在dubbo-c作为服务消费者，如何去调用远程服务dubbo-p呢？

按照前面对于服务远程通信的原理来说，服务提供方必然需要将服务发布到网络上，并且提供对应的访问协议。而服务消费端必然需要基于这个协议来进行访问。

> 这个时候，dubbo这个中间件就派上用场了，它的最基本作用就是提供服务的发布和服务的远程访问。

引入Dubbo发布服务

引入dubbo依赖包

```
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo</artifactId>
    <version>3.0.2.1</version>
</dependency>
```

基于注解的方式对服务进行暴露

```
@DubboService(methods = {@Method(name = "doKill",executes = 10,actives = 10)},connections =
10,protocol = "dubbo")
public class UserServiceImpl implements UserService {
    @Override
    public String queryUser(String s) {
        String username = RpcContext.getContext().getAttachment("username");
        System.out.println(username);
        System.out.println("=======provider======" + s);
        return "OK--" + s;
    }

    @Override
```

```
    public void doKill(String s) {
        System.out.println("=======provider======" + s);
    }
}
```

@DubboService注解就是服务暴露的注解，想要暴露什么服务就在类上加上该注解。

@DubboService的扫描

在配置类上加入@EnableDubbo注解即可，如下：

```
@Configuration
@EnableDubbo(scanBasePackages = "cn.enjoy")
@PropertySource("classpath:/dubbo-provider.properties")
public class ProviderConfiguration {
    @Bean
    public ProviderConfig providerConfig() {
        ProviderConfig providerConfig = new ProviderConfig();
        providerConfig.setTimeout(1000);
        return providerConfig;
    }
}
```

启动服务

```
public class AnnotationProviderBootstrap {

    public static void main(String[] args) throws Exception {
        new EmbeddedZooKeeper(2181, false).start();
        AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(ProviderConfiguration.class);
        System.out.println("dubbo service started.");
        new CountDownLatch(1).await();
    }
}
```

采用嵌入式zookeeper的方式启动了服务。

启动成功后，会在控制台看到如下日志

```
[DUBBO] Export dubbo service cn.enjoy.service.UserService to local registry url :
injvm://127.0.0.1/cn.enjoy.service.UserService?
anyhost=true&application=dubbo_provider&bind.ip=192.168.8.44&bind.port=20880&connections=10
&deprecated=false&doKill.actives=10&doKill.executes=10&doKill.return=true&doKill.sent=true&
dubbo=2.0.2&dynamic=true&generic=false&interface=cn.enjoy.service.UserService&metadata-
type=remote&methods=doKill,queryUser&pid=5784&release=3.0.2.1&revision=1.0-
SNAPSHOT&side=provider&timeout=1000&timestamp=1632296280067&token=2f518b59-a264-4b8f-afce-
a29b9af5f5a9, dubbo version: 3.0.2.1, current host: 192.168.116.1
[22/09/21 03:38:00:000 CST] main  INFO config.ServiceConfig:  [DUBBO] Register dubbo
service cn.enjoy.service.UserService url cn.enjoy.service.UserService to registry
127.0.0.1:2181, dubbo version: 3.0.2.1, current host: 192.168.116.1
```

通过上述步骤，就表示UserService已经发布到了网络上，基于NettyServer的形式，默认监听20880端口

服务消费者引入dubbo

- 添加jar包依赖

```
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo</artifactId>
    <version>3.0.2.1</version>
</dependency>
```

- 基于注解的形式引入服务

```
@DubboReference(check = false,protocol = "dubbo",retries = 3,timeout =
1000000000,cluster = "failover",loadbalance = "random",sticky = true,methods =
{@Method(name = "doKill",isReturn = false)}/*,url = "dubbo://localhost:20880"*/)
UserService userService;
```

- 扫描属性上的@DubboReference注解，配置类如下：

```
@Configuration
@EnableDubbo(scanBasePackages = "cn.enjoy")
@PropertySource("classpath:/dubbo-consumer.properties")
@ComponentScan(value = {"cn.enjoy"})
public class ConsumerConfiguration {
}
```

@EnableDubbo(scanBasePackages = "cn.enjoy")就会扫描@DubboReference注解的属性并且进行属性的依赖注入。

无注册中心下指定服务提供端的url

配置如下

```
 @DubboReference(check = false,protocol = "dubbo",retries = 3,timeout = 1000000000,cluster =
"failover",loadbalance = "random",sticky = true,methods = {@Method(name = "doKill",isReturn
= false)},url = "dubbo://localhost:20880")
UserService userService;
```

总结

简单总结一下上面的整个过程，其实不难发现，Dubbo这个中间件为我们提供了服务远程通信的解决方案。通过dubbo这个框架，可以开发者快速高效的构建微服务架构下的远程通信实现。

不知道大家是否发现，我们在使用dubbo发布服务，或者消费服务的时候，全程都是采用spring的配置来完成的，这样的好处是我们在学习或者使用dubbo时，如果你用过spring这个框架，那么对于它的学习难度会大大的降低。而且我们也可以看到，dubbo是完全集成Spring 的，因此后续我们去分析dubbo的源码时，还是会有一些和spring有关的内容。而且如果大家之前学习过我手写RPC的那节课，也基本能猜测到它的整个实现结构，大家不妨大胆的去猜测dubbo的一些实现细节，以助于后续在深度学习dubbo时更好的理解。

引入注册中心

Dubbo并不仅仅只是一个RPC框架，他还是一个服务治理框架，它提供了对服务的统一管理、以及服务的路由等功能。

在上面的案例中，我们只是掩饰了Dubbo作为RPC通信的点对点服务，但是就像咱们前面在学习spring cloud的内容一样，服务多了以后，如何管理和维护，以及动态发现呢？

而且，从Dubbo的架构图中可以看到，Dubbo天然就支持服务注册与发现，官方最早推荐的服务注册中心是zookeeper，当然，目前dubbo能够支持的注册中心已经非常多了，比如

consul、etcd、nacos、sofa、zookeeper、eureka、redis等等，很显然，Dubbo已经在往一个独立微服务解决方案的生态在发展。

集成Zookeeper作为服务注册中心

- 添加zookeeper的jar包依赖

```
<dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-dependencies-zookeeper</artifactId>
        <version>${dubbo.version}</version>
        <type>pom</type>
</dependency>
```

- 添加注册中心配置

  创建dubbo-provider.properties文件用于取代xml中的配置属性，然后用@PropertySource加载该properties配置文件即可 。 dubbo-provider.properties文件内容如下：

```
dubbo.application.name=dubbo_provider
dubbo.registry.address=zookeeper://${zookeeper.address:127.0.0.1}:2181
dubbo.protocol.name=dubbo
dubbo.protocol.port=20880
#dubbo.provider.token=true
```

配置管理

- 配置组件

  Dubbo框架的配置项比较繁多，为了更好地管理各种配置，将其按照用途划分为不同的组件，最终所有配置项都会汇聚到URL中，传递给后续处理模块。

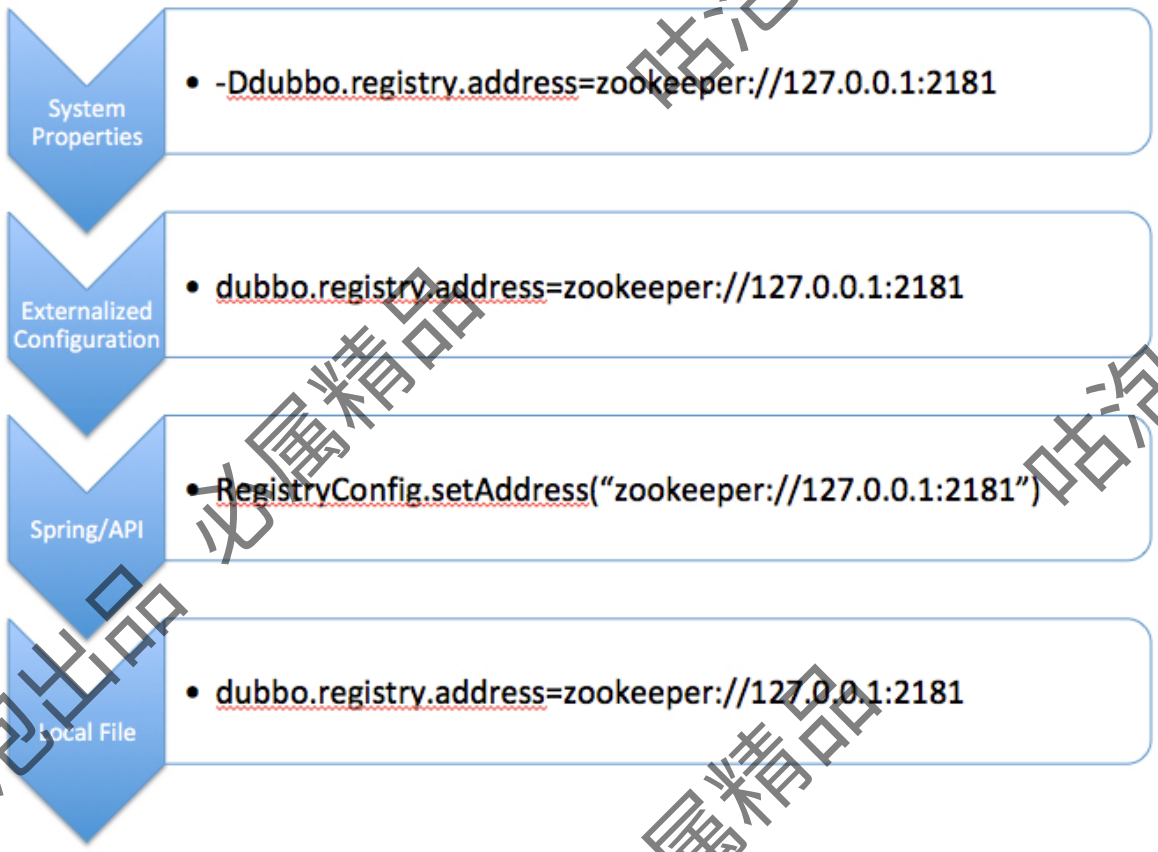
  常用配置组件如下：

  - application: Dubbo应用配置
  - registry: 注册中心
  - protocol: 服务提供者RPC协议
  - config-center: 配置中心
  - metadata-report: 元数据中心
  - service: 服务提供者配置
  - reference: 远程服务引用配置
  - provider: service的默认配置或分组配置
  - consumer: reference的默认配置或分组配置
  - module: 模块配置
  - monitor: 监控配置
  - metrics: 指标配置
  - ssl: SSL/TLS配置
- 配置来源

  从Dubbo支持的配置来源说起，默认有6种配置来源：

  - JVM System Properties，JVM -D 参数
  - System environment，JVM进程的环境变量
  - Externalized Configuration，外部化配置，从配置中心读取
  - Application Configuration，应用的属性配置，从Spring应用的Environment中提取"dubbo"打头的属性集
  - API / XML /注解等编程接口采集的配置可以被理解成配置来源的一种，是直接面向用户编程的配置采集方式
  - 从classpath读取配置文件 dubbo.properties
- 覆盖关系

  下图展示了配置覆盖关系的优先级，从上到下优先级依次降低：



API配置

以API 配置的方式来配置你的 Dubbo 应用

通过API编码方式组装配置，启动Dubbo，发布及订阅服务。此方式可以支持动态创建ReferenceConfig/ServiceConfig，结合泛化调用可以满足API Gateway或测试平台的需要。

- 服务提供者

  通过ServiceConfig暴露服务接口，发布服务接口到注册中心。

```
public class ProviderApi {
    public static void main(String[] args) throws IOException {
        new EmbeddedZooKeeper(2181, false).start();
        // 服务实现
        UserService demoService = new UserServiceImpl();

        // 当前应用配置
        ApplicationConfig application = new ApplicationConfig();
        application.setName("demo-provider");

        // 连接注册中心配置
        RegistryConfig registry = new RegistryConfig();
        registry.setAddress("zookeeper://127.0.0.1:2181");

        // 服务提供者协议配置
```

```
        ProtocolConfig protocol = new ProtocolConfig();
        protocol.setName("dubbo");
        protocol.setPort(12345);
        protocol.setThreads(200);

        // 注意：ServiceConfig为重对象，内部封装了与注册中心的连接，以及开启服务端口
        // 服务提供者暴露服务配置
        ServiceConfig<UserService> service = new ServiceConfig<UserService>(); // 此实例很
重，封装了与注册中心的连接，请自行缓存，否则可能造成内存和连接泄漏
        service.setApplication(application);
        service.setRegistry(registry); // 多个注册中心可以用setRegistries()
        service.setProtocol(protocol); // 多个协议可以用setProtocols()
        service.setInterface(UserService.class);
        service.setRef(demoService);
        service.setVersion("1.0.0");

        // 暴露及注册服务
        service.export();

        // 挂起等待(防止进程退出)
        System.in.read();
    }
}
```

- 服务消费者

  通过ReferenceConfig引用远程服务，从注册中心订阅服务接口。

```
@Test
    public void api() {
        ApplicationConfig applicationConfig = new ApplicationConfig();
        applicationConfig.setName("dubbo_consumer");

        RegistryConfig registryConfig = new RegistryConfig();
        registryConfig.setAddress("zookeeper://127.0.0.1:2181");

        ReferenceConfig<UserService> referenceConfig = new ReferenceConfig<>();
        referenceConfig.setApplication(applicationConfig);
        referenceConfig.setRegistry(registryConfig);
        referenceConfig.setInterface(UserService.class);
        UserService userService = referenceConfig.get();
        System.out.println(userService.queryUser("jack"));
    }
```

- bootstrap 服务发布

```
public class BootstrapApi {
    public static void main(String[] args) {
        new EmbeddedZooKeeper(2181, false).start();
        ConfigCenterConfig configCenter = new ConfigCenterConfig();
        configCenter.setAddress("zookeeper://127.0.0.1:2181");

        // 服务提供者协议配置
```

```
        ProtocolConfig protocol = new ProtocolConfig();
        protocol.setName("dubbo");
        protocol.setPort(12345);
        protocol.setThreads(200);

        // 注意：ServiceConfig为重对象，内部封装了与注册中心的连接，以及开启服务端口
        // 服务提供者暴露服务配置
        ServiceConfig<UserService> demoServiceConfig = new ServiceConfig<>();
        demoServiceConfig.setInterface(UserService.class);
        demoServiceConfig.setRef(new UserServiceImpl());
        demoServiceConfig.setVersion("1.0.0");

        // 第二个服务配置
        ServiceConfig<MockService> fooServiceConfig = new ServiceConfig<>();
        fooServiceConfig.setInterface(MockService.class);
        fooServiceConfig.setRef(new MockServiceImpl());
        fooServiceConfig.setVersion("1.0.0");

        // 通过DubboBootstrap简化配置组装，控制启动过程
        DubboBootstrap.getInstance()
                .application("demo-provider") // 应用配置
                .registry(new RegistryConfig("zookeeper://127.0.0.1:2181")) // 注册中心配
置
                .protocol(protocol) // 全局默认协议配置
                .service(demoServiceConfig) // 添加ServiceConfig
                .service(fooServiceConfig)
                .start()    // 启动Dubbo
                .await();   // 挂起等待(防止进程退出)
    }
}
```

- bootstrap 服务发现

```
public class BootstrapApi {
    public static void main(String[] args) {
        // 引用远程服务
        ReferenceConfig<UserService> demoServiceReference = new
ReferenceConfig<UserService>();
        demoServiceReference.setInterface(UserService.class);
        demoServiceReference.setVersion("1.0.0");

        ReferenceConfig<MockService> fooServiceReference = new
ReferenceConfig<MockService>();
        fooServiceReference.setInterface(MockService.class);
        fooServiceReference.setVersion("1.0.0");

        // 通过DubboBootstrap简化配置组装，控制启动过程
        DubboBootstrap bootstrap = DubboBootstrap.getInstance();
        bootstrap.application("demo-consumer") // 应用配置
                .registry(new RegistryConfig("zookeeper://127.0.0.1:2181")) // 注册中心配
置
                .reference(demoServiceReference) // 添加ReferenceConfig
```

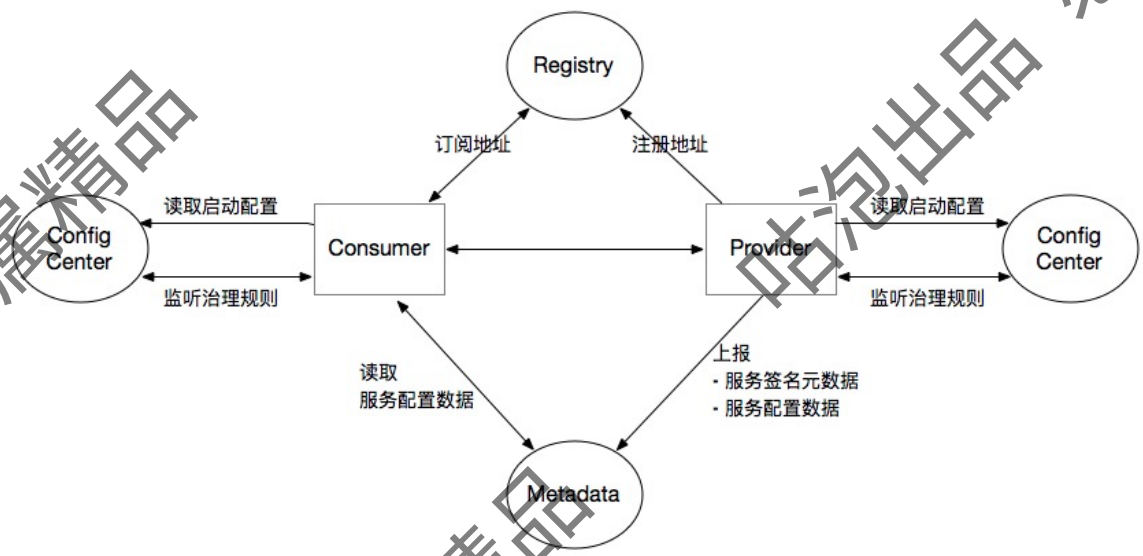
```
                .reference(fooServiceReference)
                .start();    // 启动Dubbo

        // 和本地bean一样使用demoService
        // 通过Interface获取远程服务接口代理，不需要依赖ReferenceConfig对象
        UserService demoService =
DubboBootstrap.getInstance().getCache().get(UserService.class);
        System.out.println(demoService.queryUser("jack"));

        MockService fooService =
DubboBootstrap.getInstance().getCache().get(MockService.class);
        System.out.println(fooService.queryArea("1"));
    }
}
```

部署架构（注册中心 配置中心 元数据中心）

了解 Dubbo 的三大中心化组件，它们各自的职责、工作方式。

作为一个微服务框架，Dubbo sdk 跟随着微服务组件被部署在分布式集群各个位置，为了在分布式环境下实现各个微服务组件间的协作， Dubbo 定义了一些中心化组件，这包括：

- 注册中心。协调 Consumer 与 Provider 之间的地址注册与发现
- 配置中心。
  - 存储 Dubbo 启动阶段的全局配置，保证配置的跨环境共享与全局一致性
  - 负责服务治理规则（路由规则、动态配置等）的存储与推送。
  - 简单来说，就是把 dubbo.properties 中的属性进行集中式存储，存储在其他的服务器上
  - 目前 Dubbo 能支持的配置中心有：apollo、nacos、zookeeper
- 元数据中心。
  - 接收 Provider 上报的服务接口元数据，为 Admin 等控制台提供运维能力（如服务测试、接口文档等）
  - 作为服务发现机制的补充，提供额外的接口/方法级别配置信息的同步能力，相当于注册中心的额外扩展



dubbo高级用法

启动是检查

通过spring配置文件

关闭某个服务的启动时检查 (没有提供者时报错):

```
<dubbo:reference interface="com.foo.BarService" check="false" />
```

关闭所有服务的启动时检查 (没有提供者时报错):

```
<dubbo:consumer check="false" />
```

关闭注册中心启动时检查 (注册订阅失败时报错):

```
<dubbo:registry check="false" />
```

通过 dubbo.properties

```
dubbo.reference.com.foo.BarService.check=false
dubbo.consumer.check=false
dubbo.registry.check=false
```

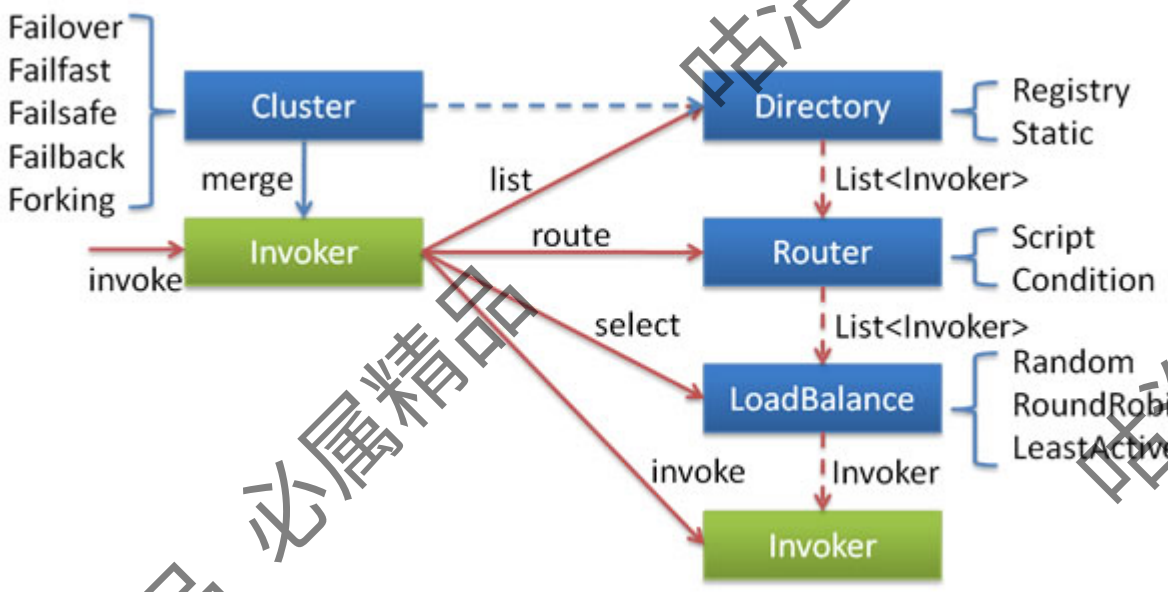
通过 -D 参数

```
java -Ddubbo.reference.com.foo.BarService.check=false
java -Ddubbo.consumer.check=false
java -Ddubbo.registry.check=false
```

集群容错

集群调用失败时，Dubbo 提供的容错方案

在集群调用失败时，Dubbo 提供了多种容错方案，缺省为 failover 重试。



各节点关系：

- 这里的 `Invoker` 是 `Provider` 的一个可调用 `Service` 的抽象，`Invoker` 封装了 `Provider` 地址及 `Service` 接口信息
- `Directory` 代表多个 `Invoker`，可以把它看成 `List<Invoker>`，但与 `List` 不同的是，它的值可能是动态变化的，比如注册中心推送变更
- `Cluster` 将 `Directory` 中的多个 `Invoker` 伪装成一个 `Invoker`，对上层透明，伪装过程包含了容错逻辑，调用失败后，重试另一个
- `Router` 负责从多个 `Invoker` 中按路由规则选出子集，比如读写分离，应用隔离等
- `LoadBalance` 负责从多个 `Invoker` 中选出具体的一个用于本次调用，选的过程包含了负载均衡算法，调用失败后，需要重选

集群容错模式

Failover Cluster

失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长延迟。可通过 `retries="2"` 来设置重试次数(不含第一次)。该配置为缺省配置

Failfast Cluster

快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

Failsafe Cluster

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 `forks="2"` 来设置最大并行数。

Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错。通常用于通知所有提供者更新缓存或日志等本地资源信息。

Available Cluster

调用目前可用的实例（只调用一个），如果当前没有可用的实例，则抛出异常。通常用于不需要负载均衡的场景。

配置：

```
@reference(cluster = "broadcast", parameters = {"broadcast.fail.percent", "20"})
```

负载均衡

Dubbo 提供的集群负载均衡策略

在集群负载均衡时，Dubbo 提供了多种均衡策略，缺省为 `random` 随机调用。

具体实现上，Dubbo 提供的是客户端负载均衡，即由 Consumer 通过负载均衡算法得出需要将请求提交到哪个 Provider 实例。

负载均衡策略

目前 Dubbo 内置了如下负载均衡算法，用户可直接配置使用：

算法	特性	备注
RandomLoadBalance	加权随机	默认算法，默认权重相同
RoundRobinLoadBalance	加权轮询	借鉴于 Nginx 的平滑加权轮询算法，默认权重相同，
LeastActiveLoadBalance	最少活跃优先 + 加权随机	背后是能者多劳的思想
ShortestResponseLoadBalance	最短响应优先 + 加权随机	更加关注响应速度
ConsistentHashLoadBalance	一致性 Hash	确定的入参，确定的提供者，适用于有状态请求

配置：

```
<dubbo:service interface="..." loadbalance="roundrobin" />
```

直连提供者

Dubbo 中点对点的直连方式

在开发及测试环境下，经常需要绕过注册中心，只测试指定服务提供者，这时候可能需要点对点直连，点对点直连方式，将以服务接口为单位，忽略注册中心的提供者列表，A 接口配置点对点，不影响 B 接口从注册中心获取列表。

配置：

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

服务分组

使用服务分组区分服务接口的不同实现

当一个接口有多种实现时，可以用 group 区分。

配置：

服务端

```
@DubboService(group = "groupImpl1")
public class GroupImpl1 implements Group {
    @Override
    public String doSomething(String s) {
        System.out.println("===========GroupImpl1.doSomething");
        return "GroupImpl1.doSomething";
    }
}
```

```
@DubboService(group = "groupImpl2")
public class GroupImpl2 implements Group {
    @Override
    public String doSomething(String s) {
        System.out.println("===========GroupImpl2.doSomething");
        return "GroupImpl2.doSomething";
    }
}
```

消费端

```
@DubboReference(check = false,group = "groupImpl1"/*,parameters = {"merger","true"}*/)
Group group;
```

多版本

在 Dubbo 中为同一个服务配置多个版本

当一个接口实现，出现不兼容升级时，可以用版本号过渡，版本号不同的服务相互间不引用。

可以按照以下的步骤进行版本迁移：

1. 在低压力时间段，先升级一半提供者为新版本
2. 再将所有消费者升级为新版本
3. 然后将剩下的一半提供者升级为新版本

生产者配置：

```
@DubboService(version = "1.0.0")
public class VersionServiceImpl implements VersionService {
```

```
    @Override
    public String version(String s) {
        System.out.println("========VersionServiceImpl.1.0.0");
        return "========VersionServiceImpl.1.0.0";
    }
}

@DubboService(version = "1.0.1")
public class VersionServiceImpl1 implements VersionService {
    @Override
    public String version(String s) {
        System.out.println("========VersionServiceImpl1.1.0.1");
        return "========VersionServiceImpl1.1.0.1";
    }
}
```

消费者配置：

```
@DubboReference(check = false,version = "1.0.0")
VersionService versionService;
```

分组聚合

通过分组对结果进行聚合并返回聚合后的结果

通过分组对结果进行聚合并返回聚合后的结果，比如菜单服务，用group区分同一接口的多种实现，现在消费方需从每种group中调用一次并返回结果，对结果进行合并之后返回，这样就可以实现聚合菜单项。

生产者配置：

```
@DubboService(group = "groupImpl1")
public class GroupImpl1 implements Group {
    @Override
    public String doSomething(String s) {
        System.out.println("==========GroupImpl1.doSomething");
        return "GroupImpl1.doSomething";
    }
}

@DubboService(group = "groupImpl2")
public class GroupImpl2 implements Group {
    @Override
    public String doSomething(String s) {
        System.out.println("==========GroupImpl2.doSomething");
        return "GroupImpl2.doSomething";
    }
}
```

消费者配置：

```
@DubboReference(check = false,group = "*",parameters = {"merger","true"})
Group group;
```

参数验证

在 Dubbo 中进行参数验证

参数验证功能是基于 JSR303 实现的，用户只需标识 JSR303 标准的验证 annotation，并通过声明 filter 来实现验证。

Maven 依赖

```
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>1.0.0.GA</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.2.0.Final</version>
</dependency>
```

参数验证类:

```
public class ValidationParamter implements Serializable {
    private static final long serialVersionUID = 32544321432L;
    @NotNull
    @Size(
        min = 2,
        max = 20
    )
    private String name;
    @Min(18L)
    @Max(100L)
    private int age;
    @Past
    private Date loginDate;
    @Future
    private Date expiryDate;
}
```

生产者:

```
@DubboService
public class ValidationServiceImpl implements ValidationService {
    @Override
    public void save(ValidationParamter validationParamter) {
        System.out.println("========ValidationServiceImpl.save");
    }

    @Override
```

```
    public void update(ValidationParamter validationParamter) {
        System.out.println("========ValidationServiceImpl.update");
    }

    @Override
    public void delete(long l, String s) {
        System.out.println("========ValidationServiceImpl.delete");
    }
}
```

消费者：

```
@DubboReference(check = false,validation = "true")
ValidationService validationService;

@Test
public void validation() {
    ValidationParamter paramter = new ValidationParamter();
    paramter.setName("Jack");
    paramter.setAge(98);
    paramter.setLoginDate(new Date(System.currentTimeMillis() - 10000000));
    paramter.setExpiryDate(new Date(System.currentTimeMillis() + 10000000));
    validationService.save(paramter);
}
```

使用泛化调用

实现一个通用的服务测试框架，可通过 `GenericService` 调用所有服务实现

泛化接口调用方式主要用于客户端没有 API 接口及模型类元的情况，参数及返回值中的所有 POJO 均用 `Map` 表示，通常用于框架集成，比如：实现一个通用的服务测试框架，可通过 `GenericService` 调用所有服务实现。

消费者：

```
@Test
public void usegeneric() {
    ApplicationConfig applicationConfig = new ApplicationConfig();
    applicationConfig.setName("dubbo_consumer");

    RegistryConfig registryConfig = new RegistryConfig();
    registryConfig.setAddress("zookeeper://192.168.67.139:2184");

    ReferenceConfig<GenericService> referenceConfig = new ReferenceConfig<>();
    referenceConfig.setApplication(applicationConfig);
    referenceConfig.setInterface("com.xiangxue.jack.service.UserService");
    //这个是使用泛化调用
    referenceConfig.setGeneric(true);
    GenericService genericService = referenceConfig.get();

    Object result = genericService.$invoke("queryUser", new String[]{"java.lang.String"},
new Object[]{"Jack"});
    System.out.println(result);
}
```

上下文信息

通过上下文存放当前调用过程中所需的环境信息

上下文中存放的是当前调用过程中所需的环境信息。所有配置信息都将转换为 URL 的参数，参见 schema 配置参考手册 中的**对应URL参数**一列。

RpcContext 是一个 ThreadLocal 的临时状态记录器，当接收到 RPC 请求，或发起 RPC 请求时，RpcContext 的状态都会变化。比如：A 调 B，B 再调 C，则 B 机器上，在 B 调 C 之前，RpcContext 记录的是 A 调 B 的信息，在 B 调 C 之后，RpcContext 记录的是 B 调 C 的信息。

消费者：

```
// 远程调用
xxxService.xxx();
// 本端是否为消费端，这里会返回true
boolean isConsumerSide = RpcContext.getContext().isConsumerSide();
// 获取最后一次调用的提供方IP地址
String serverIP = RpcContext.getContext().getRemoteHost();
// 获取当前服务配置信息，所有配置信息都将转换为URL的参数
String application = RpcContext.getContext().getUrl().getParameter("application");
// 注意：每发起RPC调用，上下文状态会变化
yyyService.yyy();
```

生产者：

```
public class XxxServiceImpl implements XxxService {

    public void xxx() {
        // 本端是否为提供端，这里会返回true
        boolean isProviderSide = RpcContext.getContext().isProviderSide();
        // 获取调用方IP地址
        String clientIP = RpcContext.getContext().getRemoteHost();
        // 获取当前服务配置信息，所有配置信息都将转换为URL的参数
        String application = RpcContext.getContext().getUrl().getParameter("application");
        // 注意：每发起RPC调用，上下文状态会变化
        yyyService.yyy();
        // 此时本端变成消费端，这里会返回false
        boolean isProviderSide = RpcContext.getContext().isProviderSide();
    }
}
```

隐式参数

通过 Dubbo 中的 Attachment 在服务消费方和提供方之间隐式传递参数

可以通过 `RpcContext` 上的 `setAttachment` 和 `getAttachment` 在服务消费方和提供方之间进行参数的隐式传递。

注意

path, group, version, dubbo, token, timeout 几个 key 是保留字段，请使用其它值。

示例：

```
RpcContext.getContext().setAttachment("index", "1");
String index = RpcContext.getContext().getAttachment("index");
```

本地调用

在 Dubbo 中进行本地调用

本地调用使用了 injvm 协议，是一个伪协议，它不开启端口，不发起远程调用，只在 JVM 内直接关联，但执行 Dubbo 的 Filter 链。

本地调用，调用的就是本地工程的接口实例

示例：

```
@DubboReference(check = false,injvm = true)
StudentService studentService;

@Test
public void inJvm() throws InterruptedException {
    System.out.println(studentService.find("xx"));
}
```

参数回调

通过参数回调从服务器端调用客户端逻辑

参数回调方式与调用本地 callback 或 listener 相同，只需要在 Spring 的配置文件中声明哪个参数是 callback 类型即可。Dubbo 将基于长连接生成反向代理，这样就可以从服务器端调用客户端逻辑。

服务接口示例：

```
public interface CallbackService {
    void addListener(String var1, CallbackListener var2);
}
```

CallbackListener.java

```
public interface CallbackListener {
    void changed(String msg);
}
```

生产者：

```
@DubboService(methods = {@Method(name = "addListener", arguments = {@Argument(index = 1,
callback = true)})})
public class CallbackServiceImpl implements CallbackService {
    @Override
    public void addListener(String s, CallbackListener callbackListener) {
        //这里就是回调客户端的方法
        callbackListener.changed(getChanged(s));
    }

    private String getChanged(String key) {
        return "Changed: " + new SimpleDateFormat("yyyy-MM-dd:mm:ss").format(new Date());
    }
}
```

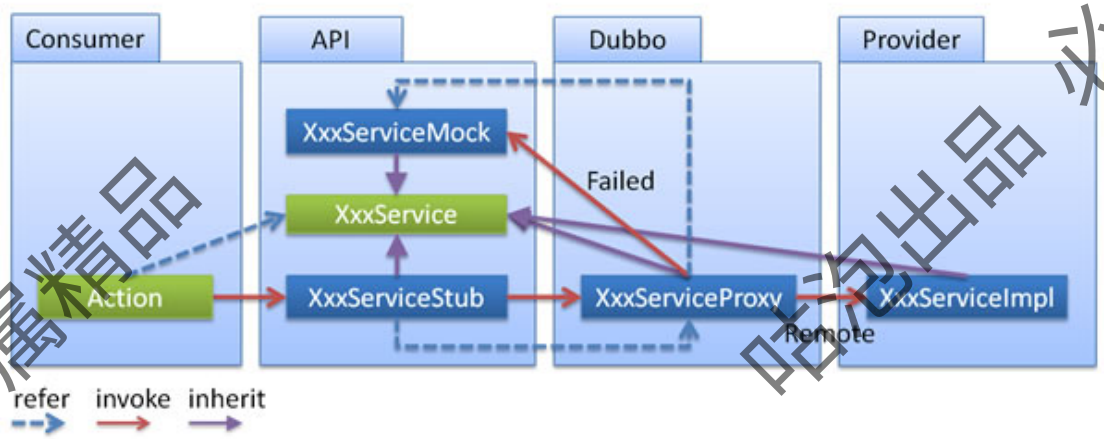
消费者：

```
@DubboReference(check = false)
CallbackService callbackService;

callbackService.addListener("jack", new CallbackListener() {
            public void changed(String arg0) {
                System.out.println("==================callback result: "
                        + arg0);
            }
});
```

本地存根

在 Dubbo 中利用本地存根在客户端执行部分逻辑

远程服务后，客户端通常只剩下接口，而实现全在服务器端，但提供方有些时候想在客户端也执行部分逻辑，比如：做 ThreadLocal 缓存，提前验证参数，调用失败后伪造容错数据等等，此时就需要在 API 中带上 Stub，客户端生成 Proxy 实例，会把 Proxy 通过构造函数传给 Stub 1，然后把 Stub 暴露给用户，Stub 可以决定要不要去调 Proxy。



消费者：

```
@DubboReference(check = false,stub = "cn.enjoy.stub.LocalStubProxy")
StubService stubService;
```

```
 * 接管了studService的调用
 *
 * 只有这个类才会决定要不要远程调用
 */
public class LocalStubProxy implements StubService {

    private StubService stubService;

    //这个必须要，传stubService实例本身
    public LocalStubProxy(StubService stubService) {
        this.stubService = stubService;
    }

    @Override
    public String stub(String s) {
        //代码在客户端执行，你可以在客户端做ThreadLocal本地缓存，或者校验参数之类工作的
        try {
            //用目标对象掉对应的方法  远程调用
            return stubService.stub(s);
        }catch (Exception e) {
            //用来降级
            System.out.println("降级数据");
        }
        //掉完后又执行代码
        return null;
    }
}
```

本地伪装

如何在 Dubbo 中利用本地伪装实现服务降级

本地伪装通常用于服务降级，比如某验权服务，当服务提供方全部挂掉后，客户端不抛出异常，而是通过 Mock 数据返回授权失败。

消费者：

```
    //这两种方式会走rpc远程调用  fail -- 会走远程服务
    @DubboReference(check = false,mock = "true")
//    @DubboReference(check = false,mock = "cn.enjoy.mock.LocalMockService")

    //不走服务直接降级  force -- 是不会走远程服务的，强制降级..这种方式是用dubbo-admin去配置它，服务治理的方式
//    @DubboReference(check = false,mock = "force:return jack")
    MockService mockService;
```

```
 * MockServiceMock
 *
 * 接口名+"Mock"
 */
public class MockServiceMock implements MockService {
    @Override
```

```
    public String mock(String s) {
        System.out.println(this.getClass().getName() + "--mock");
        return s;
    }

    @Override
    public String queryArea(String s) {
        System.out.println(this.getClass().getName() + "--queryArea");
        return s;
    }

    @Override
    public String queryUser(String s) {
        System.out.println(this.getClass().getName() + "--queryUser");
        return s;
    }
}
```

粘滞连接

为有状态服务配置粘滞连接

粘滞连接用于有状态服务，尽可能让客户端总是向同一提供者发起调用，除非该提供者挂了，再连另一台。

粘滞连接将自动开启延迟连接，以减少长连接数。

sticky=true

```
@DubboReference(check = false,protocol = "dubbo",retries = 3,timeout = 1000000000,cluster =
"failover",loadbalance = "random",sticky = true,methods = {@Method(name = "doKill",isReturn
= false)}/*,url = "dubbo://localhost:20880"*/)
UserService userService;
```

Dubbo监控平台安装

Dubbo的监控平台也做了更新，不过目前的功能还没有完善，

在这个网站上下载Dubbo-Admin的包：https://github.com/apache/dubbo-admin

1. 修改dubbo-admin-server/src/main/resources/application.properties中的配置信息
2. mvn clean package 进行构建
3. mvn –projects dubbo-admin-server spring-boot:run
4. 访问localhost:8080

https://hub.docker.com/r/apache/dubbo-admin