本文内容出自：https://github.com/gzc426/Java-Interview

以后有更新内容，会在 github 更新

加入**互联网 IT** 求职、技术交流、资料共享 QQ 群 **691206173**，3T 编程资料等你来拿，

群主（群号：**691206173**）：本硕就读于**哈尔滨工业大学**，计算机专业，2019 硕士毕业，已拿**百度** java 后台开发 offer，另外还有去哪儿，华为，茄子快传，vipkid,秒针，创新工厂一点资讯这些公司的 offer。公众号中的文章有分享群主找工作的经验，java 学习/C++学习/机器学习/前端的指导路线，以及未来在百度的学习成长之路，满满都是干货，除了干货分享还有 3T 编程资料**（java/C++/算法/php/机器学习/大数据/人工智能/面试等）**等你来拿，另外还有微信交流群以及群主的**个人微信（抽空提供一对一指导意见）,当然也希望你能帮忙在朋友圈转发推广一下**

公众号

Dubbo

简介

Dubbo是一个分布式服务框架，致力于提供高性能和透明化的RPC远程服务调用方案，以及SOA服务治理方案。简单的说，dubbo就是个服务框架，如果没有分布式的需求，其实是不需要用的，只有在分布式的时候，才有dubbo这样的分布式服务框架的需求，并且本质上是个服务调用的东东，说白了就是个远程服务调用的分布式框架（告别Web Service模式中的WSdl，以服务者与消费者的方式在dubbo上注册）

其核心部分包含:

1. 远程通讯: 提供对多种基于长连接的NIO框架抽象封装，包括多种线程模型，序列化，以及“请求-响应”模式的信息交换方式。
2. 集群容错: 提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持。
3. 自动发现: 基于注册中心目录服务，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

Dubbo能做什么?

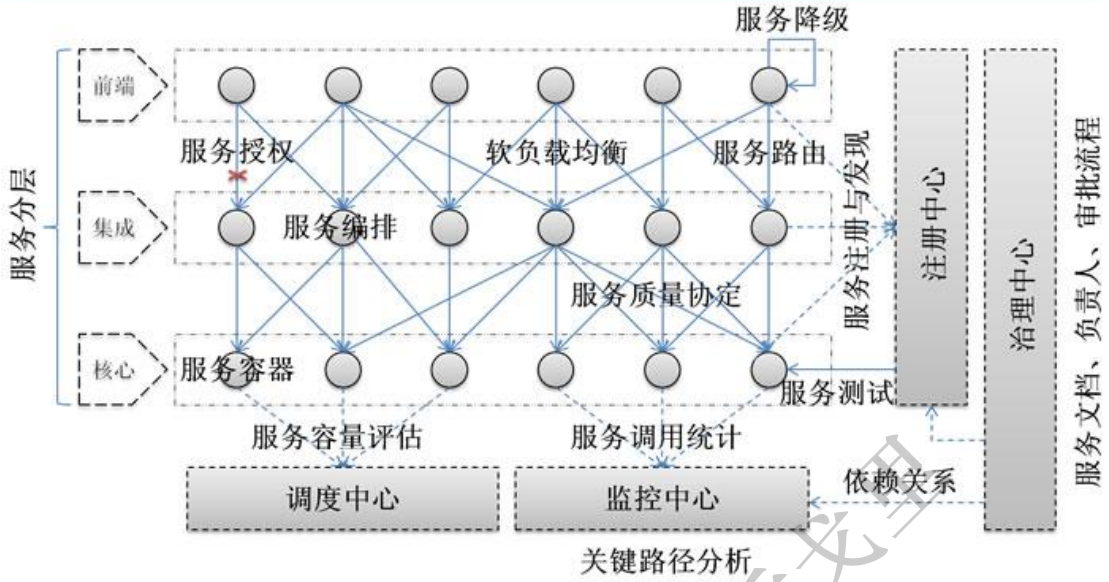
1.透明化的远程方法调用，就像调用本地方法一样调用远程方法，只需简单配置，没有任何API侵入。

2.软负载均衡及容错机制，可在内网替代F5等硬件负载均衡器，降低成本，减少单点。

3. 服务自动注册与发现，不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的IP地址，并且能够平滑添加或删除服务提供者。

4. Dubbo采用全Spring配置方式，透明化接入应用，对应用没有任何API侵入，只需用Spring加载Dubbo的配置即可，Dubbo基于Spring的Schema扩展进行加载。

Dubbo服务治理



在大规模服务化之前，应用可能只是通过 RMI 或 Hessian 等工具，简单的暴露和引用远程服务，通过配置服务的URL地址进行调用，通过 F5 等硬件进行负载均衡。

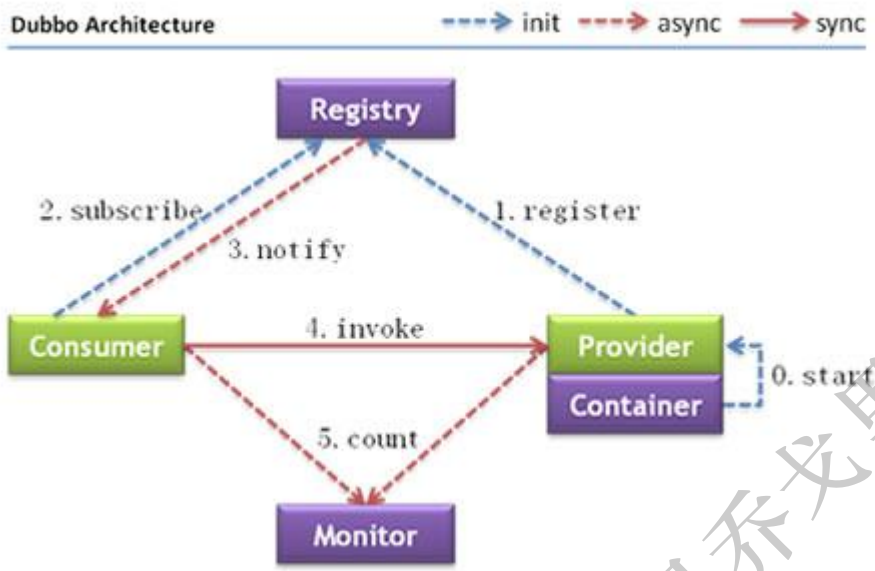
当服务越来越多时，服务 **URL** 配置管理变得非常困难，**F5** 硬件负载均衡器的单点压力也越来越大。 此时需要一个服务注册中心，动态的注册和发现服务，使服务的位置透明。并通过在消费方获取服务提供方地址列表，实现软负载均衡和 Failover，降低对 F5 硬件负载均衡器的依赖，也能减少部分成本。

当进一步发展，服务间依赖关系变得错踪复杂，甚至分不清哪个应用要在哪个应用之前启动，架构师都不能完整的描述应用的架构关系。 这时，需要自动画出应用间的依赖关系图，以帮助架构师理清理关系。

接着，服务的调用量越来越大，服务的容量问题就暴露出来，这个服务需要多少机器支撑？什么时候该加机器？ 为了解决这些问题，第一步，要将服务现在每天的调用量，响应时间，都统计出来，作为容量规划的参考指标。其次，要可以动态调整权重，在线上，将某台机器的权重一直加大，并在加大的过程中记录响应时间的变化，直到响应时间到达阀值，记录此时的访问量，再以此访问量乘以机器数反推总容量。

以上是 Dubbo 最基本的几个需求。

架构



节点角色说明

节点	角色说明
`Provider`	暴露服务的服务提供方
`Consumer`	调用远程服务的服务消费方
`Registry`	服务注册与发现的注册中心
`Monitor`	统计服务的调用次调和调用时间的监控中心
`Container`	服务运行容器

调用关系说明

1. 服务容器负责启动，加载，运行服务提供者。
2. 服务提供者在启动时，向注册中心注册自己提供的服务。
3. 服务消费者在启动时，向注册中心订阅自己所需的服务。
4. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。
5. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，**如果调用失败，再选另一台调用。**
6. **服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。**

连通性

1. 注册中心负责服务地址的注册与查找，相当于目录服务，**服务提供者和消费者只在启动时与注册中心交互**，注册中心不转发请求，压力较小
2. 监控中心负责统计各服务调用次数，调用时间等，统计先在内存汇总后每分钟一次发送到监控中心服务器，并以报表展示
3. 服务提供者向注册中心注册其提供的服务，并汇报调用时间到监控中心，此时间不包含网络开销
4. 服务消费者向注册中心获取服务提供者地址列表，并根据负载算法直接调用提供者，同时汇报调用时间到监控中心，此时间包含网络开销
5. **注册中心，服务提供者，服务消费者三者之间均为长连接，监控中心除外**
6. **注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者**
7. 注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表
8. 注册中心和监控中心都是可选的，服务消费者可以直连服务提供者

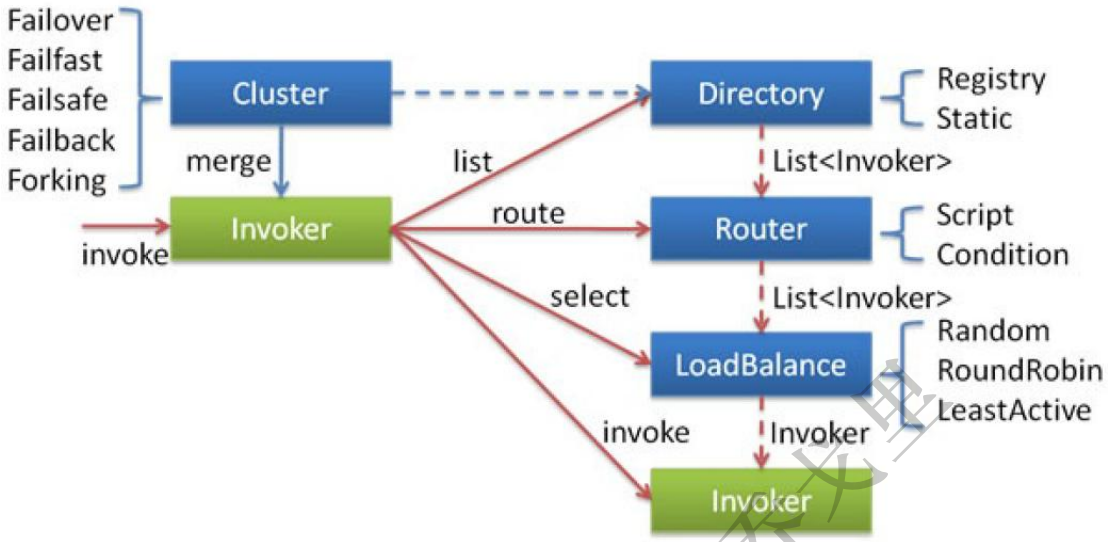
健状性

1. 监控中心宕掉不影响使用，只是丢失部分采样数据
2. 数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务
3. 注册中心对等集群，任意一台宕掉后，将自动切换到另一台
4. 注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯
5. 服务提供者无状态，任意一台宕掉后，不影响使用
6. 服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复

伸缩性

1. 注册中心为对等集群，可动态增加机器部署实例，所有客户端将自动发现新的注册中心
2. 服务提供者无状态，可动态增加机器部署实例，注册中心将推送新的服务提供者信息给消费者

集群容错



各节点关系：

1. 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
2. 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
3. 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
4. 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
5. 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

Feature	Maturity	Strength	Problem	Advise	User
Failover Cluster	Stable	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（推荐使用）	重试会带来更长延迟	可用于生产环境	Aliba
Failfast Cluster	Stable	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	如果有机器正在重启，可能会出现调用失败	可用于生产环境	Aliba
Failsafe Cluster	Stable	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	调用信息丢失	可用于生产环境	Moni

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		可用于生产环境	
Forking Cluster	Tested	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	需要浪费更多服务资源	可用于生产环境	
Broadcast Cluster	Tested	广播调用所有提供者，逐个调用，任意一台报错则报错，通常用于更新提供方本地状态	速度慢，任意一台报错则报错	可用于生产环境	

负载均衡

Random LoadBalance

随机，按权重设置随机概率。
在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。

RoundRobin LoadBalance

轮循，按公约后的权重设置轮循比率。
存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。

LeastActive LoadBalance

最少活跃调用数，相同活跃数的随机，**活跃数指调用前后计数差**。
使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。

每个服务维护一个活跃数计数器。当 A 机器开始处理请求，该计数器加 1，此时 A 还未处理

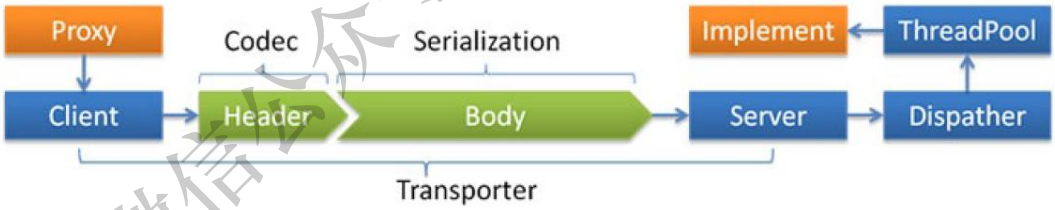
完成。若处理完毕则计数器减 1。而 B 机器接受到请求后很快处理完毕。那么 A,B 的活跃数分别是 1，0。当又产生了一个新的请求，则选择 B 机器去执行(B 活跃数最小)，这样使慢的机器 A 收到少的请求。

ConsistentHash LoadBalance

一致性 **Hash**，相同参数的请求总是发到同一提供者。
当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。
算法参见：http://en.wikipedia.org/wiki/Consistent_hashing
缺省只对第一个参数 Hash，如果要修改，请配置 <dubbo:parameter key="hash.arguments" value="0,1" />
缺省用 160 份虚拟节点，如果要修改，请配置 <dubbo:parameter key="hash.nodes" value="320" />

线程模型

如果事件处理的逻辑能迅速完成，并且不会发起新的 IO 请求，比如只是在内存中记个标识，则直接在 IO 线程上处理更快，因为减少了线程池调度。
但如果事件处理逻辑较慢，或者需要发起新的 IO 请求，比如需要查询数据库，则必须派发到线程池，否则 IO 线程阻塞，将导致不能接收其它请求。
如果用 IO 线程处理事件，又在事件处理过程中发起新的 IO 请求，比如在连接事件中发起登录请求，会报“可能引发死锁”异常，但不会真死锁。



需要通过不同的派发策略和不同的线程池配置的组合来应对不同的场景:
<dubbo:protocol name="dubbo" dispatcher="all" threadpool="fixed" threads="100" />

Dispatcher

all 所有消息都派发到线程池，包括请求，响应，连接事件，断开事件，心跳等。
direct 所有消息都不派发到线程池，全部在 IO 线程上直接执行。
message 只有请求响应消息派发到线程池，其它连接断开事件，心跳等消息，直接在 IO 线程上执行。
execution 只请求消息派发到线程池，不含响应，响应和其它连接断开事件，心跳等消息，直接在 IO 线程上执行。
connection 在 IO 线程上，将连接断开事件放入队列，有序逐个执行，其它消息派发到线程池。

ThreadPool

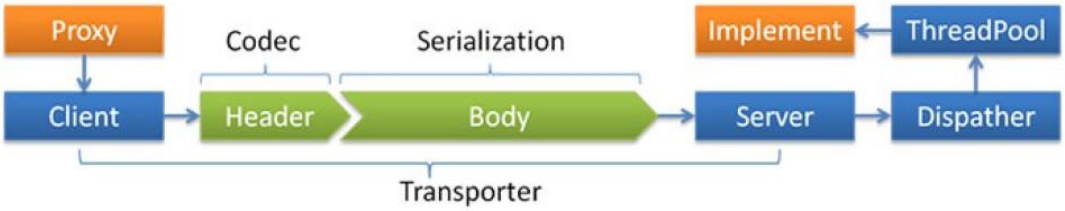
fixed 固定大小线程池，启动时建立线程，不关闭，一直持有。(缺省)
cached 缓存线程池，空闲一分钟自动删除，需要时重建。
limited 可伸缩线程池，但池中的线程数只会增长不会收缩。只增长不收缩的目的是为了避免收缩时突然来了大流量引起的性能问题。

协议

Feature	Maturity	Strength	Problem	Advise	User
Dubbo协议	Stable	采用NIO复用单一长连接，并使用线程池并发处理请求，减少握手和加大并发效率，性能较好（推荐使用）	在大文件传输时，单一连接会成为瓶颈	可用于生产环境	Aliba
Rmi协议	Stable	可与原生RMI互操作，基于TCP协议	偶尔会连接失败，需重建Stub	可用于生产环境	Aliba
Hessian协议	Stable	可与原生Hessian互操作，基于HTTP协议	需hessian.jar支持，http短连接的开销大	可用于生产环境	

dubbo

Dubbo 缺省协议采用单一长连接和 NIO 异步通讯，适合于小数据量大并发的服务调用，以及服务消费者机器数远大于服务提供者机器数的情况。
反之，Dubbo 缺省协议不适合传送大数据量的服务，比如传文件，传视频等，除非请求量很低。



- Transporter: mina, netty, grizzy
- Serialization: dubbo, hessian2, java, json
- Dispatcher: all, direct, message, execution, connection
- ThreadPool: fixed, cached

特性

缺省协议，使用基于 mina 1.1.7 和 hessian 3.2.1 的 tbremoting 交互。
连接个数：单连接
连接方式：长连接
传输协议：TCP
传输方式：NIO 异步传输
序列化：**Hessian** 二进制序列化
适用范围：**传入传出参数数据包较小（建议小于 100K），消费者比提供者个数多，单一消费者无法压满提供者，尽量不要用 dubbo 协议传输大文件或超大字符串。**
适用场景：常规远程服务方法调用

约束

参数及返回值需实现 Serializable 接口
参数及返回值不能自定义实现 List , Map , Number , Date , Calendar 等接口，只能用 JDK 自带的实现，因为 hessian 会做特殊处理，自定义实现类中的属性值都会丢失。
Hessian 序列化，只传成员属性值和值的类型，不传方法或静态变量，兼容情况

数据通讯	情况	结果
A->B	类A多一种 属性（或者说类B少一种 属性）	不抛异常，A多的那 个属性的值，B没有， 其他正常
A->B	枚举A多一种 枚举（或者说B少一种 枚举），A使用多 出来的枚举进行传输	抛异常
A->B	枚举A多一种 枚举（或者说B少一种 枚举），A不使用 多出来的枚举进行传输	不抛异常，B正常接 收数据
A->B	A和B的属性 名相同，但类型不相同	抛异常
A->B	serialId 不相同	正常传输

接口增加方法，对客户端无影响，如果该方法不是客户端需要的，客户端不需要重新部署。
输入参数和结果集中增加属性，对客户端无影响，如果客户端并不需要新属性，不用重新部署。
输入参数和结果集属性名变化，对客户端序列化无影响，但是如果客户端不重新部署，不管输入还是输出，属性名变化的属性值是获取不到的。
总结：服务器端和客户端对领域对象并不需要完全一致，而是按照最大匹配原则。

常见问题

为什么要消费者比提供者个数多?

因 dubbo 协议采用单一长连接，假设网络为千兆网卡 ，根据测试经验数据每条连接最多只能压满 7MByte(不同的环境可能不一样，供参考)，理论上 1 个服务提供者需要 20 个服务消费者才能压满网卡。

为什么不能传大包?

因 dubbo 协议采用单一长连接，如果每次请求的数据包大小为 500KByte，假设网络为千兆网卡 ，每条连接最大 7MByte(不同的环境可能不一样，供参考)，单个服务提供者的 TPS(每秒处理事务数)最大为：128MByte / 500KByte = 262。单个消费者调用单个服务提供者的 TPS(每秒处理事务数)最大为：7MByte / 500KByte = 14。如果能接受，可以考虑使用，否则网络将成为瓶颈。

为什么采用异步单一长连接?

为什么采用异步单一长连接?
因为服务的现状大都是服务提供者少，通常只有几台机器，而服务的消费者多，可能整个网站都在访问该服务，比如 Morgan 的提供者只有 6 台提供者，却有上百台消费者，每天有 1.5 亿次调用，如果采用常规的 hessian 服务，服务提供者很容易就被压跨，**通过单一连接，保证单一消费者不会压死提供者，长连接，减少连接握手验证等，并使用异步 IO，复用线程池，防止 C10K 问题**
网络服务在处理数以万计的客户端连接时，往往出现效率底下甚至完全瘫痪，这被成为 C10K 问题。(C10K = connection 10 kilo 问题)。k 表示 kilo，即 1000 比如：kilometer(千米)，kilogram(千克)。

附加功能

服务分组

当一个接口有多种实现时，可以用 group 区分。

服务

```
<dubbo:service group="feedback" interface="com.xxx.IndexService" />
<dubbo:service group="member" interface="com.xxx.IndexService" />
```

引用

```
<dubbo:reference id="feedbackIndexService" group="feedback" interface="com.xxx.IndexSe
rvice" />
<dubbo:
```

多版本

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

老版本服务提供者配置：

```
<dubbo:service interface="com.foo.BarService" version="1.0.0" />
```

新版本服务提供者配置：

```
<dubbo:service interface="com.foo.BarService" version="2.0.0" />
```

老版本服务消费者配置：

```
<dubbo:reference id="barService" interface="com.foo.BarService" version="1.0.0" />
```

新版本服务消费者配置：

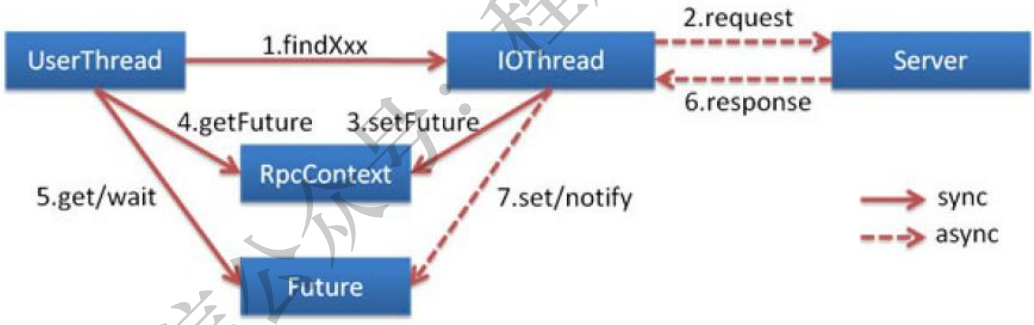
```
<dubbo:reference id="barService" interface="com.foo.BarService" version="2.0.0" />
```

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 。

异步调用

基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小。



事件通知

在调用之前、调用之后、出现异常时，会触发 oninvoke 、 onreturn 、 onthrow 三个事件，
可以配置当事件发生时，通知哪个类的哪个方法 。
服务消费者 Callback 配置

```
<bean id ="demoCallback" class = "com.alibaba.dubbo.callback.implicit.NofifyImpl" />
<dubbo:reference id="demoService" interface="com.alibaba.dubbo.callback.implicit.IDemo
Service" version="1.0.0" group="cn" >
<dubbo:method name="get" async="true" onreturn = "demoCallback.onreturn" onthrow=
"demoCallback.onthrow" />
</dubbo:reference>
```

callback 与 async 功能正交分解， async=true 表示结果是否马上返回， onreturn 表

示是
否需要回调。
两者叠加存在以下几种组合情况 ：
异步回调模式： async=true onreturn="xxx"
同步回调模式： async=false onreturn="xxx"
异步无回调 ： async=true
同步无回调 ： async=false'

本地伪装

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。
在 spring 配置文件中按以下方式配置：

```
<dubbo:service interface="com.foo.BarService" mock="true" />
```

或

```
<dubbo:service interface="com.foo.BarService" mock="com.foo.BarServiceMock" />
```

如果服务的消费方经常需要 try-catch 捕获异常，如：

```
Offer offer = null;
try {
offer = offerService.findOffer(offerId);
} catch (RpcException e) {
logger.error(e);
}
```

请考虑改为 Mock 实现，并在 Mock 实现中 return null。如果只是想简单的忽略异常，在
2.0.11 以上版本可用：

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

令牌验证

通过令牌验证在注册中心控制权限，以决定要不要下发令牌给消费者，可以防止消费者绕过注册中心访问提供者，另外通过注册中心可灵活改变授权方式，而不需修改或升级提供者

可以全局设置开启令牌验证：

```
<!--随机token令牌，使用UUID生成-->
<dubbo:provider interface="com.foo.BarService" token="true" />
```

或

```
<!--固定token令牌，相当于密码-->
<dubbo:provider interface="com.foo.BarService" token="123456" />
```

也可在服务级别设置：

```
<!--随机token令牌，使用UUID生成-->
<dubbo:service interface="com.foo.BarService" token="true" />
```

或

```
<!--固定token令牌，相当于密码-->
<dubbo:service interface="com.foo.BarService" token="123456" />
```

还可在协议级别设置：

```
<!--随机token令牌，使用UUID生成-->
<dubbo:protocol name="dubbo" token="true" />
```

或

```
<!--固定token令牌，相当于密码-->
<dubbo:protocol name="dubbo" token="123456" />
```

写入路由规则

向注册中心写入路由规则的操作通常由监控中心或治理中心的页面完成

condition:// 表示路由规则的类型，支持条件路由规则和脚本路由规则，可扩展，必填。
0.0.0.0 表示对所有 IP 地址生效，如果只想对某个 IP 的生效，请填入具体 IP，必填。
com.foo.BarService 表示只对指定服务生效，必填。

category=routers 表示该数据为动态配置类型，必填。
dynamic=false 表示该数据为持久数据，当注册方退出时，数据依然保存在注册中心，必填。
enabled=true 覆盖规则是否生效，可不填，缺省生效。
force=false 当路由结果为空时，是否强制执行，如果不强制执行，路由结果为空的路由规则将自动失效，可不填，缺省为 flase 。
runtime=false 是否在每次调用时执行路由规则，否则只在提供者地址列表变更时预先执行并缓存结果，调用时直接从缓存中获取路由结果。如果用了参数路由，必须设为 true ，需要注意设置会影响调用的性能，可不填，缺省为 flase 。
priority=1 路由规则的优先级，用于排序，优先级越大越靠前执行，可不填，缺省为 0 。
rule=URL.encode("host = 10.20.153.10 => host = 10.20.153.11") 表示路由规则的内容，必填。

优雅停机

‘Dubbo 是通过 JDK 的 ShutdownHook 来完成优雅停机的，所以如果用户使用 kill -9 PID 等强制关闭指令，是不会执行优雅停机的，只有通过 kill PID 时，才会执行。

原理

服务提供方

停止时，先标记为不接收新请求，新请求过来时直接报错，让客户端重试其它机器。
然后，检测线程池中的线程是否正在运行，如果有，等待所有线程执行完成，除非超时，则强制关闭。

服务消费方

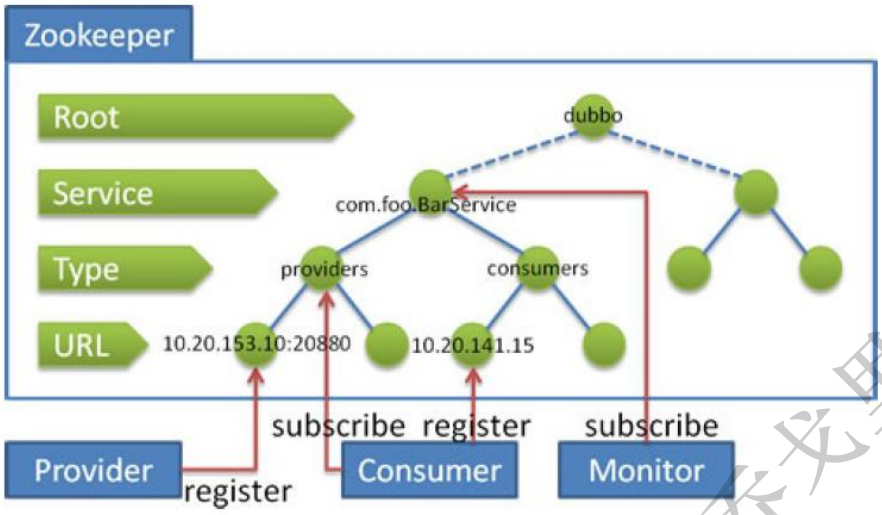
停止时，不再发起新的调用请求，所有新的调用在客户端即报错。
然后，检测有没有请求的响应还没有返回，等待响应返回，除非超时，则强制关闭。

设置方式

设置优雅停机超时时间，缺省超时时间是 10 秒，如果超时则强制关闭。

```
<dubbo:application ...>
<dubbo:parameter key="shutdown.timeout" value="60000" /> <!-- 单位毫秒 -->
</dubbo:application>
```

注册中心



流程说明：

服务提供者启动时：向 /dubbo/com.foo.BarService/providers 目录下写入自己的 URL 地址服务消费者启动时：订阅 /dubbo/com.foo.BarService/providers 目录下的提供者 URL 地址。并向 /dubbo/com.foo.BarService/consumers 目录下写入自己的 URL 地址监控中心启动时：订阅 /dubbo/com.foo.BarService 目录下的所有提供者和消费者 URL 地址。

支持以下功能：

当提供者出现断电等异常停机时，注册中心能自动删除提供者信息
当注册中心重启时，能自动恢复注册数据，以及订阅请求
当会话过期时，能自动恢复注册数据，以及订阅请求
当设置 <dubbo:registry check="false" /> 时，记录失败注册和订阅请求，后台定时重试
可通过 <dubbo:registry username="admin" password="1234" /> 设置 zookeeper 登录信息
可通过 <dubbo:registry group="dubbo" /> 设置 zookeeper 的根节点，不设置将使用无根树支持 * 号通配符 <dubbo:reference group="*" version="*" /> ，可订阅服务的所有分组和所有版本的提供者使用

最佳实践

分包

建议将服务接口，服务模型，服务异常等均放在 API 包中，因为服务模型及异常也是 API 的一部分，同时，这样做也符合分包原则：重用发布等价原则(REP)，共同重用原则(CRP)。
如果需要，也可以考虑在 API 包中放置一份 spring 的引用配置，这样使用方，只需在 spring 加载过程中引用此配置即可，配置建议放在模块的包目录下，以免冲突，

如： com/alibaba/china/xxx/dubbo-reference.xml 。

粒度

服务接口尽可能大粒度，每个服务方法应代表一个功能，而不是某功能的一个步骤，否则将面临分布式事务问题，Dubbo 暂未提供分布式事务支持。
服务接口建议以业务场景为单位划分，并对相近业务做抽象，防止接口数量爆炸。
不建议使用过于抽象的通用接口，如： Map query(Map) ，这样的接口没有明确语义，会给后期维护带来不便。

版本

每个接口都应定义版本号，为后续不兼容升级提供可能，如： <dubbo:service interface="com.xxx.XxxService" version="1.0" /> 。
建议使用两位版本号，因为第三位版本号通常表示兼容升级，只有不兼容时才需要变更服务版本。
当不兼容时，先升级一半提供者为新版本，再将消费者全部升为新版本，然后将剩下的一半提供者升为新版本。

兼容性

服务接口增加方法，或服务模型增加字段，可向后兼容，删除方法或删除字段，将不兼容，枚举类型新增字段也不兼容，需通过变更版本号升级。

枚举值

如果是完备集，可以用 Enum ，比如： ENABLE , DISABLE 。
如果是业务种类，以后明显会有类型增加，不建议用 Enum ，可以用 String 代替。
如果是在返回值中用了 Enum ，并新增了 Enum 值，建议先升级服务消费方，这样服务提供方不会返回新值。
如果是在传入参数中用了 Enum ，并新增了 Enum 值，建议先升级服务提供方，这样服务消费方不会传入新值。

序列化

服务参数及返回值建议使用 POJO 对象，即通过 setter , getter 方法表示属性的对象。
服务参数及返回值不建议使用接口，因为数据模型抽象的意义不大，并且序列化需要接口实现类的元信息，并不能起到隐藏实现的意图。

服务参数及返回值都必需是 byValue 的，而不能是 byReference 的，消费方和提供方的参数或返回值引用并不是同一个，只是值相同，Dubbo 不支持引用远程对象。

异常

建议使用异常汇报错误，而不是返回错误码，异常信息能携带更多信息，以及语义更友好。
如果担心性能问题，在必要时，可以通过 override 掉异常类的 fillInStackTrace() 方法为空方法，使其不拷贝栈信息。
查询方法不建议抛出 checked 异常，否则调用方在查询时将过多的 try...catch ，并且不能进行有效处理。
服务提供方不应将 DAO 或 SQL 等异常抛给消费方，应在服务实现中对消费方不关心的异常进行包装，否则可能出现消费方无法反序列化相应异常。

调用

不要只是因为是 Dubbo 调用，而把调用 try...catch 起来。 try...catch 应该加上合适的回滚边界上。
对于输入参数的校验逻辑在 Provider 端要有。如有性能上的考虑，服务实现者可以考虑在 API 包上加上服务 Stub 类来完成检验。

Provider 上尽量多配置 Consumer 端属性

原因如下：
作服务的提供者，比服务使用方更清楚服务性能参数，如调用的超时时间，合理的重试次数，等等
在 Provider 配置后，Consumer 不配置则会使用 Provider 的配置值，即 Provider 配置可以作为 Consumer 的缺省值 。否则，Consumer 会使用 Consumer 端的全局设置，这对于 Provider 不可控的，并且往往是不合理的
Provider 上尽量多配置 Consumer 端的属性，让 Provider 实现者一开始就思考 Provider 服务特点、服务质量的问题。

Provider 上配置合理的 Provider 端属性

Provider 上可以配置的 Provider 端属性有：
1. threads 服务线程池大小
2. executes 一个服务提供者并行执行请求上限，即当 Provider 对一个服务的并发调用到上限后，新调用会 Wait，这个时候 Consumer 可能会超时。在方法上配置 dubbo:method

则并发限制针对方法，在接口上配置 dubbo:service ，则并发限制针对服务

使用建议

Dubbo 服务划分

1、服务划分目标

抽取系统中独立的业务模块服务化，按业务独立性进行垂直划分，抽象出基础服务层

2、子系统划分把控：合理划分，过细过粗都不行

3、注意事项

1)表：**避免出现 A 服务关联 B 服务的表的数据操作；服务一旦划分了，那么数据库即便没分开，也要当成 db 表分开了来进行编码；否则 AB 服务难以进行垂直拆库**

2）**避免服务耦合度高，依赖调用**；如果出现，考虑服务调优。

3）**避免分布式事务，不要拆分过细。**

Dubbo 接口划分

1、**接口尽可能大粒度，接口中的方法不要以业务流程来，这个流程尽量在方法逻辑中调用，接口应代表一个完整的功能对外提供；**

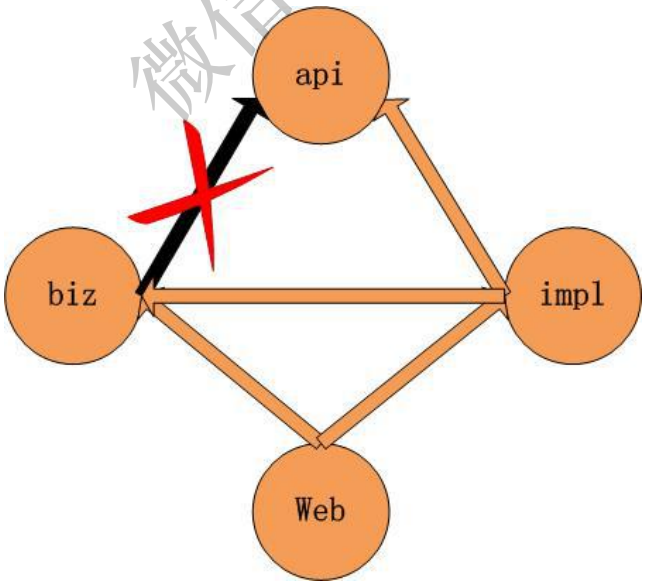
2、接口应以业务为单位，业务相近的进行抽象，避免接口数量爆炸

3、参数先做校验，在传入接口。

4、要做到在设计接口时，已经确定这个接口职责、预测调用频率

依赖

web 应用大致分为两层：biz 和 web，实际上 biz 可能由内部多个工程组成，这里 biz 只是一个抽象概念。impl 依赖 api 和 biz，web 依赖 impl 和 biz，没有其他依赖关系，严禁 biz 依赖 api。关系如下：



注意事项

服务化接口涉及的入参类型和返回类型都必须实现序列化接口，并且必须放到 api 包。

- 在提供者的 dubbo 配置文件中，一般都配置了`<dubbo:protocol name="dubbo" port="20880"/>`，表明用 dubbo 协议在 20880 端口暴露服务，当然如果你不配置，dubbo 默认使用 20880 端口暴露服务，所有消费者都是通过 20880 端口进行，对于消费者而言，提供者服务器 8080 端口是透明的，也就是说提供者服务器端口号可以任意改变，服务也不会有任何影响，消费者无需关心。

- zookeeper 的 2181 开放给 provider、consumer、dubbo-admin

- provider 的 20880 开放给所有 consumer，但 8080 服务器端口可以完全屏蔽

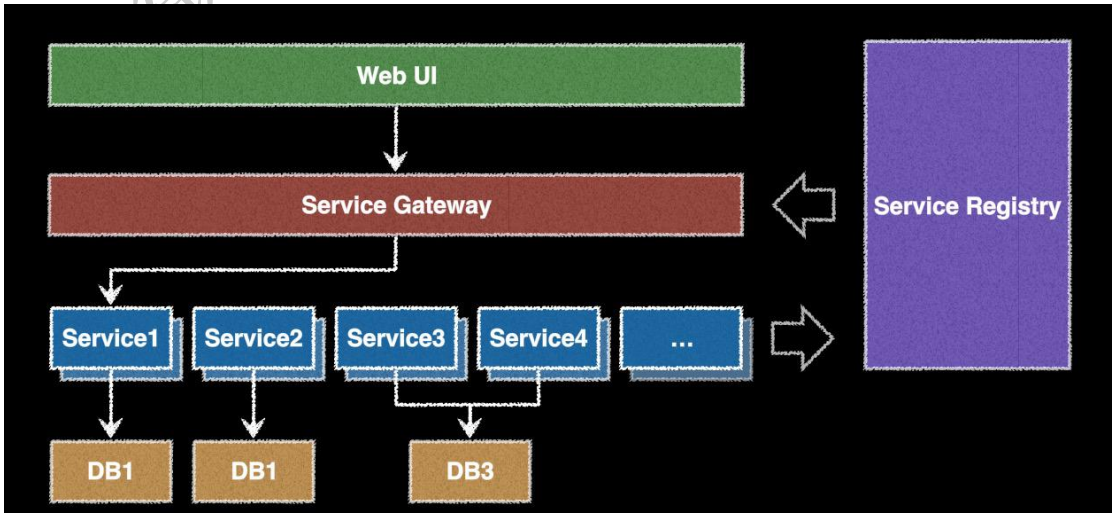
- consumer 的 8080 开放给所有 provider

- dubbo-admin 的 8080 开放给管理员用户，便于通过浏览器监控注册中心服务的情况

分包

biz-service
biz-api
biz-web

web 依赖于 api
service 依赖于 api
service 和 web 没有依赖
controller 是一个工程，service+dao 是一个工程



biz-service 放 converter,dao,service.impl,application,config, properties

biz-api 放 domain,enumeration, exception
biz-web 放 controller,config,properties
common 放公共代码

序列化

默认的 hession2 方式不支持 LocalDateTime 等时间类的序列化，会出现 StackOverflowError

```
dubbo=com.alibaba.dubbo.common.serialize.support.dubbo.DubboSerialization

hessian2=com.alibaba.dubbo.common.serialize.support.hessian.Hessian2Serialization

java=com.alibaba.dubbo.common.serialize.support.java.JavaSerialization

compactedjava=com.alibaba.dubbo.common.serialize.support.java.CompactedJavaSerialization

json=com.alibaba.dubbo.common.serialize.support.json.JsonSerialization

fastjson=com.alibaba.dubbo.common.serialize.support.json.FastJsonSerialization

nativejava=com.alibaba.dubbo.common.serialize.support.nativejava.NativeJavaSerialization

kryo=com.alibaba.dubbo.common.serialize.support.kryo.KryoSerialization

fst=com.alibaba.dubbo.common.serialize.support.fst.FstSerialization

jackson=com.alibaba.dubbo.common.serialize.support.json.JacksonSerialization
```

事务

如果仅仅是应用拆分，而没有数据库的拆分，那么仍可视为单机事务。
如果数据库也进行了拆分，每个应用使用的表被放在相应机器的数据库中，那么需要考虑分布式事务问题。
分布式事务的前提是多个数据库之间难以保证一致性，单一数据库本身的事务机制可以保证数据一致性。