

1, 简介

Dubbo是一款RPC服务开发框架，用于解决微服务架构下的服务治理和通信问题。

优点

- **微服务编程范式和工具**

Dubbo 支持基于 IDL 或语言特定方式的服务定义，提供多种形式的服务调用形式（如同步、异步、流式等）

- **高性能的 RPC 通信**

Dubbo 帮助解决微服务组件之间的通信问题，提供了基于 HTTP、HTTP/2、TCP 等的多种高性能通信协议实现，并支持序列化协议扩展，在实现上解决网络连接管理、数据传输等基础问题。

- **微服务监控与治理**

Dubbo 官方提供的服务发现，动态配置，负载均衡，流量路由等基础组件解决微服务基础实践的问题，还可完成限流降级，数据一致性，链路追踪的能力

注意：

- Dubbo不只是一个RPC框架，可使用多种通信协议，Dubbo提供了丰富的服务治理能力和生态（类似Spring cloud）

面试

1、为什么Dubbo不用jdk的SPI，而是要自己实现？

java SPI

概念原理

SPI：是官方提供的一种**服务发现机制**，它允许在**运行时动态加载实现特定接口的类**，而不需要在代码中显式的指定该类，从而实现**解耦和灵活性**。

本质就是：约定一个目录文件，通过接口名去找那个目录文件，文件解析得到实现类的全类名，循环加载实现类和实例化。

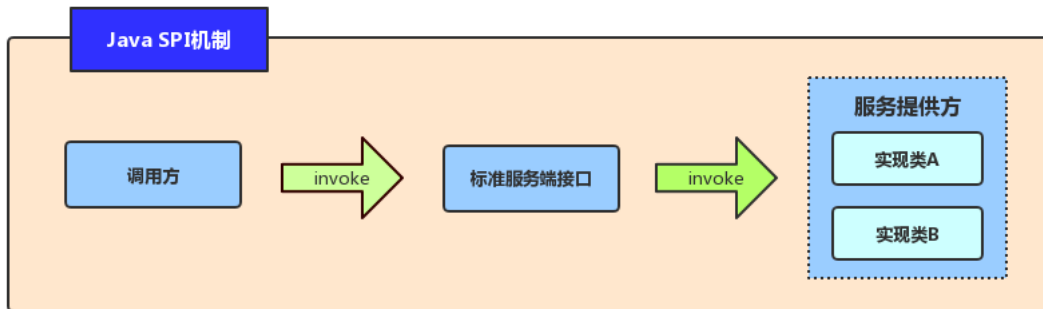
- **使用步骤**

1. **定义接口**：首先需要定义一个接口，所有实现该接口的类都将被注册为服务提供者。
2. **创建实现类**：创建一个或多个实现接口的类，这些类将作为服务提供者。
3. **配置文件**：在 META-INF/services 目录下创建一个以接口全限定名命名的文件，文件内容为实现该接口的类的全限定名，每个类名占一行。
4. **加载使用服务**：使用 java.util.ServiceLoader 类的静态方法 load(Class service) 加载服务，默认情况下会加载 classpath 中所有符合条件的提供者。调用 ServiceLoader 实例的 iterator() 方法获取迭代器，遍历迭代器即可获取所有实现了该接口的类的实例。

注意事项

- **接口必须是公共的，且只能包含抽象方法。**

- 实现类必须有一个无参构造函数。
- 配置文件中指定的类必须是实现了相应接口的非抽象类。
- 配置文件必须放在 META-INF/services 目录下。
- 配置文件的文件名必须为接口的全限定名。



java spi的实现原理是java类加载机制和反射机制。

当使用 `ServiceLoader.load(Class<T> service)` 加载服务时，会检查 `META-INF/services` 目录下是否存在以接口全名命名的文件，如果存在，则读取该文件内容，获取该接口的所有实现类的全名，并通过 `Class.forName()` 方法加载到对应的类。

java 是**懒加载迭代器**的思想，按需加载。【此按需加载是需要SPI的实现类时，读取配置文件，将所有类加载进来，而不是读取配置文件只加载某个需要的类】

在调用 `ServiceLoader.load(Class<T> service)` 加载服务时，不会将所有所有实现类都加载进来，而是返回一个懒加载迭代器。在使用该实现类时，才按需加载，创建该实例。

懒加载的优点:

- 节省内存：若一次性把所有实现类都加载进来，导致内存过大，影响性能。
- 增强灵活性：由于 `ServiceLoader` 是动态加载的，因此可以在程序运行时添加或删除实现类，而无需修改代码或重新编译。（一般用不到）

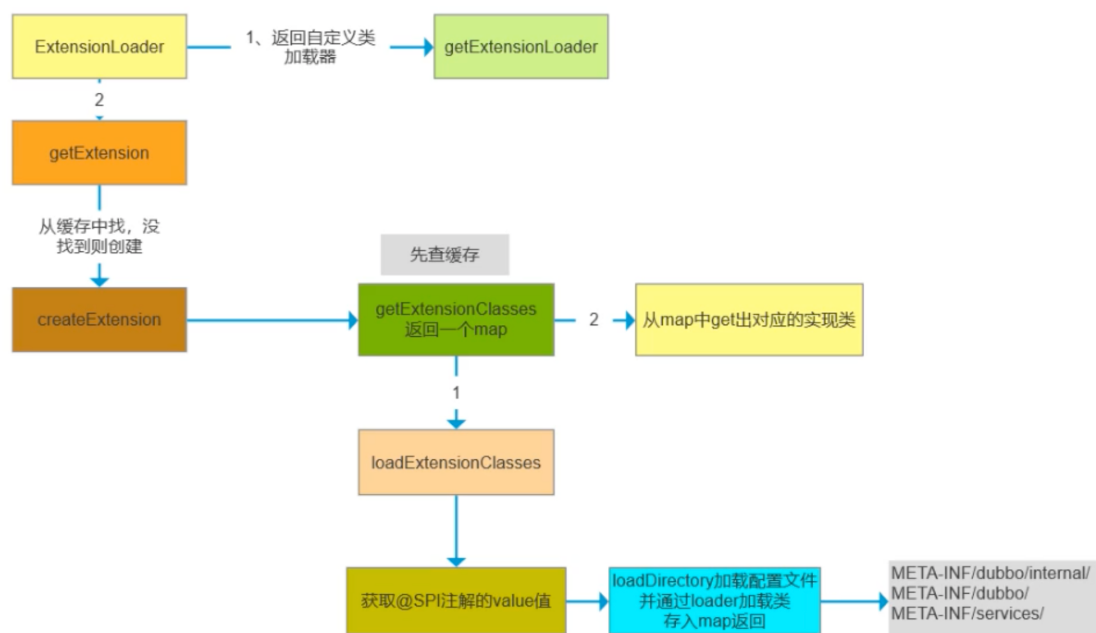
优缺点

- 优点
 - **松耦合性**：程序可以在程序运行时，动态的加载实现类，无需在编译时将实现类硬编码到代码中。
 - **拓展性**：通过SPI，应用程序为同一个接口定义多个实现类，方便拓展
 - **易于使用**：使用SPI，开发者只需要实现接口，并制定类的类名，即可使用
- 缺点
 - **配置麻烦**：需要在在 `META-INF/services` 目录下创建配置文件，并将实现类的类名写入其中。这使得配置相对较为繁琐。
 - **安全性不足**：若配置文件内容不正确，则存在风险
 - **性能损失**：每次找到对应的服务提供者都需要重新读取配置文件，增加了启动时间和性能开销

应用场景

- 数据库驱动程序加载
 - JDBC为了实现可插拔的数据库驱动，定义了一套API规范，而具体的数据库厂商实现Driver接口，提供自己数据库驱动程序，在java中，驱动程序的加载是通过SPI机制实现的
- 日志框架的实现
 - log4j, slf4j都采用了SPI机制，需要修改代码
- Spring代码
 - Spring中的Bean加载机制就是实现了SPI思想，通过读取classpath下的META-INF/spring.factories文件来加载各种自定义的Bean。
- Dubbo框架
 - 通过注解@SPI 声明拓展点接口，并在classpath下的META-INF/dubbo目录中提供实现类的配置文件，来实现扩展点的动态加载。

Dubbo SPI



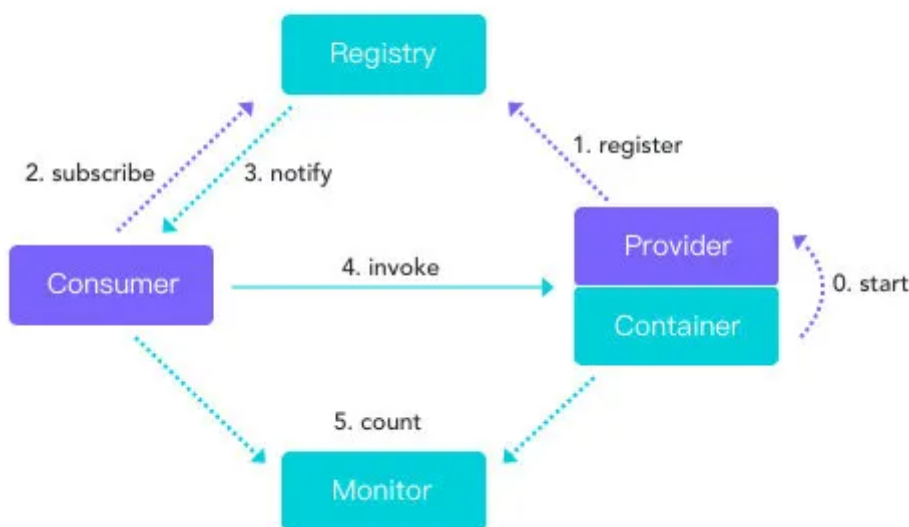
Dubbo通过ExtensionLoader获取自定义的加载器。然后获取对应的实现「按需加载」。对应的实现，可以通过 @SPI注解指定 value值，即SPI的路径。「方便了SPI指定路径，即拓展更新方便」

- java spi 缺点
 - 会遍历所有实现，并反射创建实例化。若有的实现类初始化过程缓慢，耗时。而使用者代码没有用到，则造成了资源的浪费
 - 没有使用缓存，每次load，都需要重新加载对象，也增加了启动时间和性能的开销
- Dubbo spi
 - 每个实现类都配置了名字，通过名字去文件里找对应的实现类，并实例化，按需加载。文件配置是键值对形式，而java SPI是每一行一个全类名
 - 增加了缓存存储实例，提高了读取性能
 - 提供了对Spring中IOC和AOP的支持，方便了拓展

2、Dubbo工作流程

Dubbo Architecture

.....▶ init ▶ async —▶ sync



1. Start: 启动Spring时，自动启动Dubbo的provider
2. Register: Dubbo的provider在启动后，会去注册中心注册内容，包括：IP、端口号、接口列表（接口类、方法名）、版本、provider的协议
3. Subscribe: 订阅，Consumer启动时，自动去Registry注册中心获取到已经注册的provider的信息，并缓存下来
4. Notify: 当Provider的信息发生变更时，自动由Registry注册中心向Consumer推变更信息，Consumer会更新自己的缓存信息
5. Invoke: Consumer调用Provider中的方法
6. Count: 次数，每2分钟，Provider和Consumer自动向Monitor 监控中心发送访问次数，Monitor 进行统计。

注意：

- 注册中心和监控中心是可选的，可以没有注册中心和监控中心，Provider的相关信息，可以在Consumer机器的配置文件中写。
- 注册中心，消费者和生产者都是长连接，和监控方不是长连接。**在调用时，消费者直接调用生产者，不经过注册中心。**
- 注册中心全部宕机也不会影响已经正常运行的消费者和生产者，因为消费者已经将生产者的相关信息都缓存到本地。

3、Dubbo 服务暴露过程

Dubbo 定义URL 作为参数类型

```
protocol://username:password@host:port/path?key=value&key=value
protocol: dubbo中的各种协议，如dubbo、http等
username/password: 用户名、密码
host/port: 主机端口号
path: 接口名称
parameters: 参数键值对
```

ServiceBean实现了ApplicationListener，监听ContextRefreshedEvent事件，在Spring IOC容器刷新完成后调用onApplicationEvent方法，服务暴露的启动方法。获取配置的URL信息，再利用Dubbo SPI机制根据URL的参数，选择合适的实现类，实现拓展

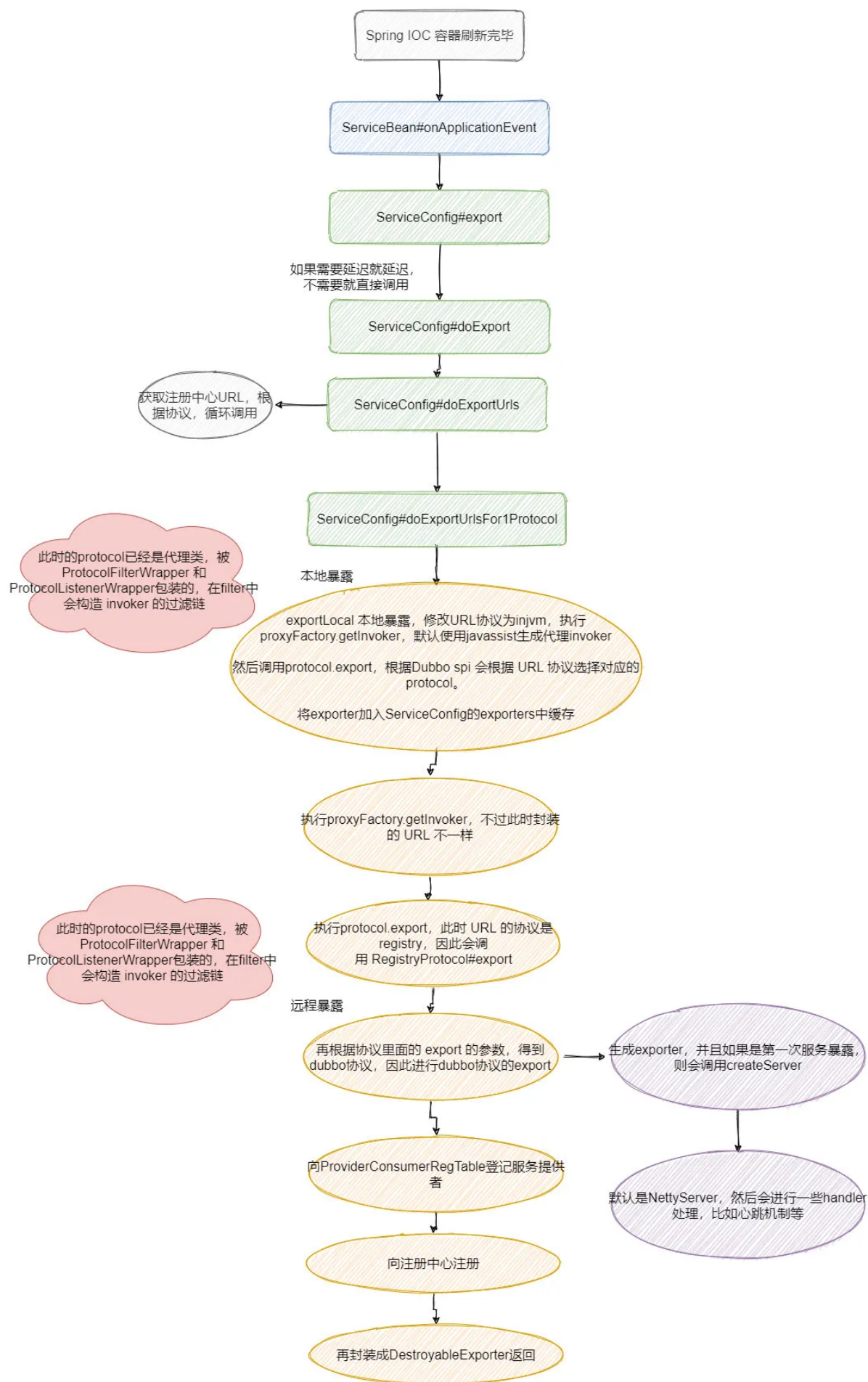
通过javassist动态封装服务实现类，统一暴露出Invoker使得调用更方便，屏蔽了底层实现细节，然后封装成Exporter存储起来，等待消费者的调用，并且会将URL注册到注册中心，使得消费者可以获取服务提供者的信息。

一个服务若是有多多个协议，则需要暴露多次，即需要向多个注册中心暴露注册。

流程：

1. ServiceBean实现了ApplicationListener，监听ContextRefreshedEvent事件，在Spring IOC容器刷新完成后调用onApplicationEvent方法，服务暴露的启动方法。检查配置项，如果有些配置空的话会默认创建，组装成URL。
2. 根据URL进行服务暴露，创建代理类Invoker，根据URL得知具体的协议信息，根据Dubbo SPI获取实现类实现Exporter（Exporter 是对外交互的对象，与注册中心交互、Consumer交互）
3. 暴露接口：export
 1. 如果是本地暴露（injvm协议），则将Exporter存入ServiceConfig的缓存
 2. 远程暴露：先通过Registry协议找到RegistryProtocol 进行Export，将URL中export=dubbo://... 先转换成exporter，然后获取注册中心的相关配置，如果需要注册，则向注册中心配置，并且在ProviderConsumerRegtable这个表格中记录服务提供者，本质是在一个concurrentHashMap中put Invoker，key就是服务接口全类名+端口号，value是一个Set，Set里面是包装过的Invoker，根据URL上的Dubbo协议暴露出Exporter，**打开Server调用NettyServer监听服务**（本地暴露不需要打开）

- 为什么搞本地暴露
 - 暴露的本地服务在内部调用的时候可以直接消费同一个JVM服务，避免了网络通信



4、Dubbo服务引用过程

引入过程分为饿汉式和懒汉式。

1. 饿汉式：通过调用ReferenceBean的afterPropertiesSet方法，引入服务。（即通过配置文件配置启动，所有的配置类都需要注入到IOC）
2. 懒汉式：当这个服务被注入到其他类时，才启动引用流程。【用到该服务才引入】默认使用懒汉式。

ReferencgetBean 实现了FactoryBean接口，当任何服务Interface进行注入或者getBean获取时，最后都会调用getObject的服务调用过程，分为三种方式。

- 引入方式
 - 本地引入，则走injvm协议，直接从服务暴露的缓存中获取exporter
 - 直连远程服务（未使用注册中心，消费者配置文件中指定各个Provider信息），直连即可
 - 注册中心引入远程服务，Consumer通过注册中心得知Provider的相关信息，然后进行服务的引入
 - 获取注册中心实例，向注册中心注册自身应用，并且订阅Providers、configurations、routers节点，触发DubboInvoker的生成，cluster（集群）将多个服务调用者进行封装，返回一个invoker。cluster负责决定负载均衡，Consumer调用时，具体访问哪台机器
- **RPC框架不侵入业务代码，因此会创建代理类，包装下，创建invoker，屏蔽创建的细节。**

引入过程：通过配置创建一个map，然后通过map，构建URL信息「IP、端口号、接口列表（接口类、方法名）、版本、provider的协议」，再通过URL上的协议，利用自适应拓展机器调用对应的protocol.refer得到对应的invoker。然后再构建代理，封装invoker返回服务引用。Consumer调用时，都是调用这个invoker。

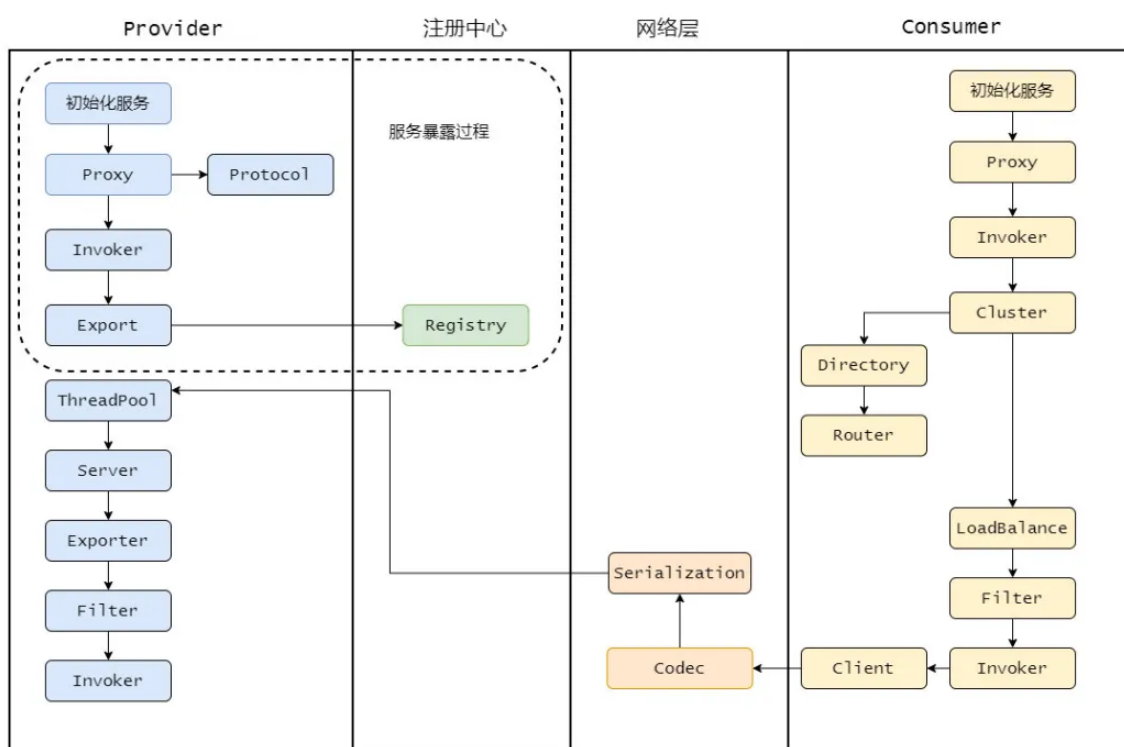
- 引入过程
 - 通过Spring读取配置配置文件配置，将通过各个配置项构建成Map信息
 - 通过Map构建动态代理，最终是通过invoker进行构建，实际是对Invoker的封装
 - 构建URL信息
 - 首先判断是否走injvm协议，本地引用，若是本地引用，则从JVM内存中获取
 - 远程引用分为直连和通过注册中心获取。判断URL是否是直连链接，若不是，则走注册中心引入远程服务
 - 走注册中心，则从注册中心获取Provider的地址等信息，将这些信息拼接到URL对象中。Provider可能有多个，所以可能生成多个URL对象（一个provider对应一个URL）
 - 通过URL构建Invoker
 - 若是只有一个URL，则直接构建URL信息
 - 若是存在多个URL，则先遍历构建多个Invoker，再通过Cluster【进行负载均衡】进行合并，只暴露出一个Invoker。
 - 调用RegistryProtocol#refer，构建Invoker
 - 根据URL，获取注册中心实例【注册中心变更监听通过该注册中心实例同步】
 - 生成消费者的链接，并注册到注册中心
 - 订阅注册中心的Providers目录、configurators目录、routers目录。若订阅的信息发生变化，会同步该Invoker。
 - 创建Invoker，包含netty client（通过网络通信）、通过cluser封装Invoker，使其只暴露一个Invoker。\\

invoker 分为多种：本地引入的invoke、直连的invoker、注册中心集群的invoker。「即使有多个 Provider，服务引入时，也是获取一个invoker」Invoker 代表一个可执行体。Invoker屏蔽了调用细节，暴露出一个统一的执行体。

• BeanFactory、FactoryBean、ObjectFactory的区别

- BeanFactory、ObjectFactory都是Bean工厂，实际就是IOC容器，都通过它获取Bean对象。ObjectFactory用于延迟创建对象的场景，从这个类获取对象时，才触发对象的创建、延迟实例化。
- FactoryBean是一个Bean，封装了对象的创建细节，从IOC容器获取Bean，最终都会调到 FactoryBean#getObject方法，实质上就是包了一层呢个，避免了创建对象的细节。

5、Dubbo 服务调用过程



1. 调用某个接口的方法，调用服务引入时生成的代理类Proxy。Proxy调用invoke方法，需要从本地缓存获取可远程调用的invoke列表。然后会从cluster中经过路由的过滤、负载均衡机制选择一个 invoker发起远程调用，再通过Client进行数据传输。传输过程还需要Codec进行做协议改造，再序列化。此时会记录该请求和请求ID等待服务端的响应。
2. 服务端接收到请求后，通过参数获取到之前暴露的map，得到对应的exporter，然后最终调用真正的实现类，再组装好结果返回，响应会携带上请求时的ID。
3. 消费者收到响应后，通过该请求ID，找到对应的请求，将响应放在对应的Future，唤醒等待的线程，并且响应。

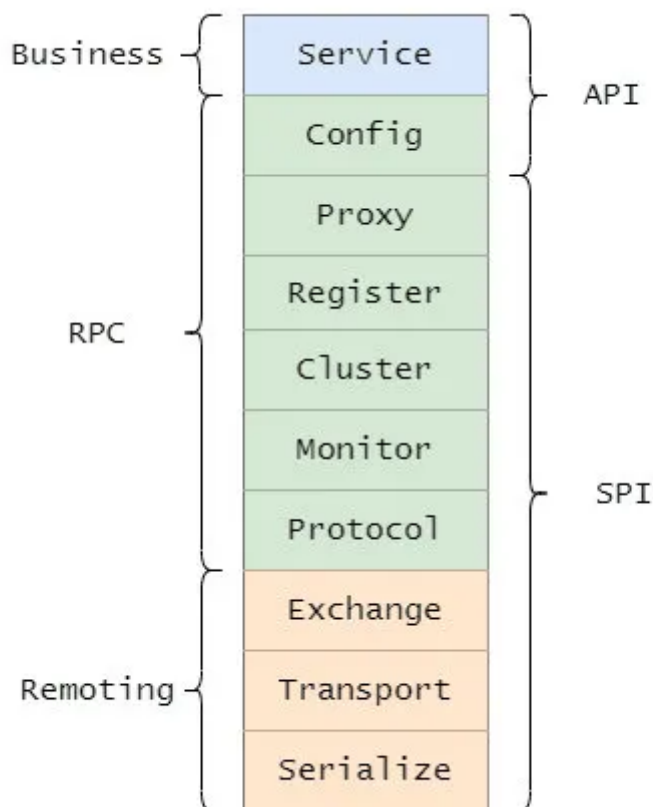
• Dubbo协议

- 使用的handler+body的方式。协议头16字节，包含协议版本、接口名字、接口版本、方法名字、requestID等信息

• 序列化协议

- dubbo默认采用hessian2 序列化协议。数据紧凑，占用字节少，传输快。缺点：调试困难，读取需接口特殊工具转换
- 调用流程（客户端）
 - 请求过程
 - 调用具体的接口会生成代理类，代理类会生成一个RpcInvocation对象，调用Invoker方法。通过反射调用该方法。
 - 通过方法名获取提供支持的Invokers，再通过负载均衡的计算，得到一个Invoker。若是配置了服务降级，则会获取到mockInvoker，执行特定的逻辑。
 - 通过Invoker构建request，dubbo内部默认是异步调用，调用请求，返回Future
 - 请求成功后，如何根据相应，找到对应的请求？请求会携带requestId，返回时也根据该requestId进行对应。有一个currentHashMap，将requestId和请求的future进行存储，方便对应。
 - 响应过程
 - 根据响应response携带的requestId从currentHashMap中找到对应的future。
 - 将future塞入response，唤醒等待线程
- 调用流程（服务端）
 - 解析request，获取requestId和请求数据信息
 - 获取对应的provider中的Invoker。【通过请求的接口全类名和端口号拼接成serviceKey，在map中获取Invoker。该map是在启动暴露时，存再map中的】

6、Dubbo的分层设计



将分层分为业务层（Business层）、RPC层、Remoting层，层次分明，则**职责边界清晰方便维护和拓展**

又可统称为API层和SPI层。通过微内核设计+SPI拓展。若实现定制化需求，则通过SPI拓展即可。

- Service：业务层，技术开发的业务逻辑层
- Config：配置层，根据ServiceConfig和ReferenceConfig，初始化配置信息
- Proxy：代理层，服务提供者还是消费者都会生成一个代理类，使得服务接口透明化，代理层做远程调用和返回结果
- Register：注册层，向注册中心注册和发现
- Cluster：路由和集群容错层，服务选取具体调用的节点信息，处理特殊的调用要求和负责远程调用失败的容错措施
- Monitor：监控层，负责向监控统计调用时间、次数、可用率等信息
- Portocol：远程调用层，主要是封装RPC调用，负责管理Invoker
- Exchange：信息交换层，用来封装请求响应，同步转异步
- Transport：网络传输层，抽象了网络传输的统一接口，Netty、Mina等
- Serialize：序列化层，将数据序列化成二进制流和反序列化

7、Dubbo 服务 负载均衡策略

消费者调用的invoker是ClusterInvoker（所有的Invoker）。相同的服务在一个服务目录中，服务目录是Invoker的集合。监听注册中心的Invoker变化，刷新本地invokers集合。

- 获取invoker集合，将所有的Invoker放在 Map<String,List<Invoker>> 中
- 监听注册中心的变化，实现NotifyListener 接口，服务引入时会订阅
- 刷新invoker，获取<方法名, Invoker 列表> 映射关系，并将相同服务的invoker进行合并，只返回一个invoker，保存在上面的Map中

路由：将通过负载均衡决定调用哪天服务提供者，这是由消费者决定的。

Dubbo判断哪台机器时，也会进行预热。就像缓存有预热、JIT有预热、服务器提供服务也需要预热。在负载均衡时，则降低刚启动机器的权重。

1. 平衡加权轮询算法。

1. 加了权重的轮询，例如 A、B权重是3:1，那么顺序则是A、B、A、A、B、A..... 打算顺序的3:1

2. 加权随机算法（默认策略）

1. 提供相同服务的机器 S1、S2，S1的权重是7，S2的权重3。S1->[0,7), S2->[7,10)。则计算随机数计算落在区间。

```
public class RandomLoadBalance extends AbstractLoadBalance {

    public static final String NAME = "random";

    @Override
    protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {

        int length = invokers.size();
        boolean sameweight = true;
        int[] weights = new int[length];
```

```

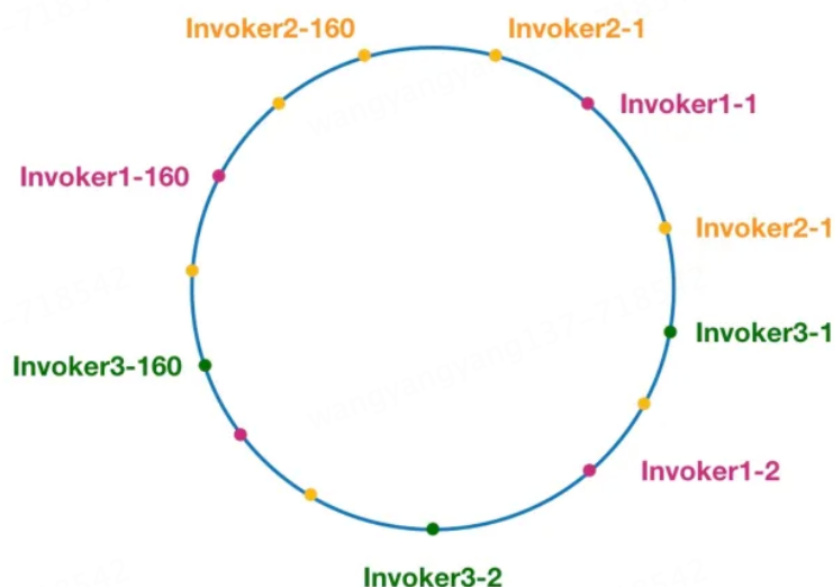
    int totalWeight = 0;
    // 下面这个for循环的主要作用就是计算所有该服务的提供者的权重之和
    totalWeight(),
    // 除此之外，还会检测每个服务提供者的权重是否相同
    for (int i = 0; i < length; i++) {
        int weight = getweight(invokers.get(i), invocation);
        totalWeight += weight;
        weights[i] = totalWeight;
        if (sameweight && totalWeight != weight * (i + 1)) {
            sameweight = false;
        }
    }
    if (totalWeight > 0 && !sameweight) {
        // 随机生成一个 [0, totalWeight) 区间内的数字
        int offset =
        ThreadLocalRandom.current().nextInt(totalWeight);
        // 判断会落在哪个服务提供者的区间
        for (int i = 0; i < length; i++) {
            if (offset < weights[i]) {
                return invokers.get(i);
            }
        }
    }

    return
    invokers.get(ThreadLocalRandom.current().nextInt(length));
}
}

```

3. 一致性哈希算法

1. 就是哈希一致性算法。根据请求参数进行哈希计算。相同参数的请求总是发到同一个服务提供者。



4. 最小活跃数算法

1. 活跃数：初始状态下，所有的活跃数为0。收到请求，活跃数+1，结束请求，活跃数-1。「提供方 每个方法都只有单独的活跃数」
2. 若活跃数相等，则走加权随机算法。注意，不是最小活跃数的几个Invoker走加权随机算法，而是所有Provider走加权随机算法。

```
public class LeastActiveLoadBalance extends AbstractLoadBalance {

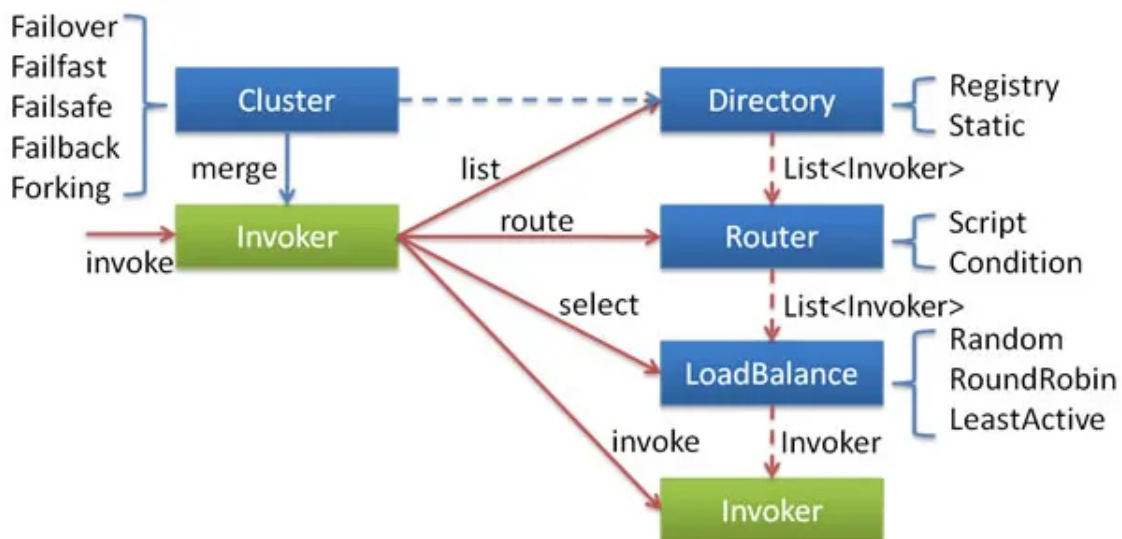
    public static final String NAME = "leastactive";

    @Override
    protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL
url, Invocation invocation) {
        int length = invokers.size();
        int leastActive = -1;
        int leastCount = 0;
        int[] leastIndexes = new int[length];
        int[] weights = new int[length];
        int totalWeight = 0;
        int firstWeight = 0;
        boolean sameweight = true;
        // 这个 for 循环的主要作用是遍历 invokers 列表，找出活跃数最小的 Invoker
        // 如果有多个 Invoker 具有相同的最小活跃数，还会记录下这些 Invoker 在
invokers 集合中的下标，并累加它们的权重，比较它们的权重值是否相等
        for (int i = 0; i < length; i++) {
            Invoker<T> invoker = invokers.get(i);
            // 获取 invoker 对应的活跃(active)数
            int active = RpcStatus.getStatus(invoker.getUrl(),
invocation.getMethodName()).getActive();
            int afterWarmup = getWeight(invoker, invocation);
            weights[i] = afterWarmup;
            if (leastActive == -1 || active < leastActive) {
                leastActive = active;
                leastCount = 1;
                leastIndexes[0] = i;
                totalWeight = afterWarmup;
                firstWeight = afterWarmup;
                sameweight = true;
            } else if (active == leastActive) {
                leastIndexes[leastCount++] = i;
                totalWeight += afterWarmup;
                if (sameweight && afterWarmup != firstWeight) {
                    sameweight = false;
                }
            }
        }
        // 如果只有一个 Invoker 具有最小的活跃数，此时直接返回该 Invoker 即可
        if (leastCount == 1) {
            return invokers.get(leastIndexes[0]);
        }
        // 如果有多个 Invoker 具有相同的最小活跃数，但它们之间的权重不同
        // 这里的处理方式就和 RandomLoadBalance 一致了
        if (!sameweight && totalWeight > 0) {
            int offsetWeight =
ThreadLocalRandom.current().nextInt(totalWeight);
```

```

        for (int i = 0; i < leastCount; i++) {
            int leastIndex = leastIndexes[i];
            offsetWeight -= weights[leastIndex];
            if (offsetWeight < 0) {
                return invokers.get(leastIndex);
            }
        }
    }
    return
    invokers.get(leastIndexes[ThreadLocalRandom.current().nextInt(leastCount
    )]);
}
}

```



- 首先在服务引入的时候，将多个远程调用都塞入 Directory 中，然后通过 Cluster 来封装这个目录，封装的同时提供各种容错功能，比如 FailOver、FailFast 等等，最终暴露给消费者的就是一个 invoker。
- 然后消费者调用的时候会目录里面得到 invoker 列表，会经过路由的过滤，得到这些 invokers 之后再由 loadBalance 来进行负载均衡选择一个 invoker，最终发起调用。
- 这种过程其实是在 Cluster 的内部发起的，所以能在发起调用出错的情况下，用上容错的各种措施。

8、zk如何存储Dubbo生产者和消费者信息

zk一个多叉树，类似文件目录结构。

生产者消费者启动，都是在zk中注册

- key：接口全类名
- value：Provider和Consumer

- Provider包含多个URL，即多个不同的提供方

zk会对Provider和Consumer加入watcher，可以拉取所有值。且都是临时提供者，若是Provider1宕机，则通过心跳监测到该机器无响应，则删除Provider1的URL信息。

9、Spring Cloud 和Dubbo的区别

- 底层协议：SpringCloud基于http协议，dubbo基于Tcp协议，dubbo的性能相对较好
- 注册中心：Spring Cloud 使用的eureka，dubbo推荐使用zookeeper「eureka AP 是高可用性的，zk CP保证了数据的强一致性，zk要求数据超过半数的节点是一致的，否则会进入崩溃恢复」CAP
- 模型定义：dubbo将一个接口定义为一个服务，SpringCloud 则是将一个应用定义为一个服务
- SpringCloud提供了一个生态，而Dubbo是SpringCloud中关于服务调用的一种解决方案（服务治理）

10、zk集群挂掉，发布者和订阅者还能通信吗，若是发布者呢

可以，启动dubbo容器时，消费者会去zookeeper拉取注册的发布者的信息，进行本地缓存，调用时以负载均衡策略，从本地获取某一台机器，但zookeeper挂掉后，后续新的生产者，无法被消费者调用

- 若服务发布者，无状态，则RPC调用失败后，会调用其他机器，后续路由选择时，不会再次选中该台机器
- 若服务发布者全部宕机，服务消费者无法使用，并且无限次重连，等待服务者恢复

11、为什么用 javassist 而不用 jdk 动态代理是

因为 javassist 快。

RPC和Dubbo的区别
