day04

dubbo高可用

1）集群容错

- 服务路由：服务路由包含一条路由规则，路由规则决定了服务消费者的调用目标，即规定了服务消费者可调用哪些服务提供者，du'b'bo提供三种服务路由实现，分别为条件路由ConditionRouter、脚本路由ScriptRouter、标签路由TagRouter，本课程重点分析条件路由

  条件路由规则的格式：

  [服务消费者匹配条件] => [服务提供者匹配条件]

  host = 10.20.153.10 => host = 10.20.153.11

  该条规则表示 IP 为 10.20.153.10 的服务消费者**只可**调用 IP 为 10.20.153.11 机器上的服务，不可调用其他机器上的服务。

  如果服务消费者匹配条件为空，表示不对服务消费者进行限制。如果服务提供者匹配条件为空，表示对某些服务消费者禁用服务

  常见路由配置：

  - 白名单：

    host!=10.20.153.10,10.20.153.11=>

  - 黑名单

    host=10.20.153.10,10.20.153.11=>

  - 读写分离

    method=find,*list*,get,*is*=>host=172.22.3.94,172.22.3.95,172.22.3.96
    method!=find,*list*,get,*is*=>host=172.22.3.97,172.22.3.98

  - 前后台分离

    application=front=>host=172.22.3.91,172.22.3.92,172.22.3.93
    application!=front=>host=172.22.3.94,172.22.3.95,172.22.3.96

- 集群容错

  - Failover Cluster 失败自动切换，当出现失败，重试其它服务器。(缺省) 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)。

    重试次数配置如下：

    <dubbo:service retries="2" />

    或

    <dubbo:reference retries="2" />

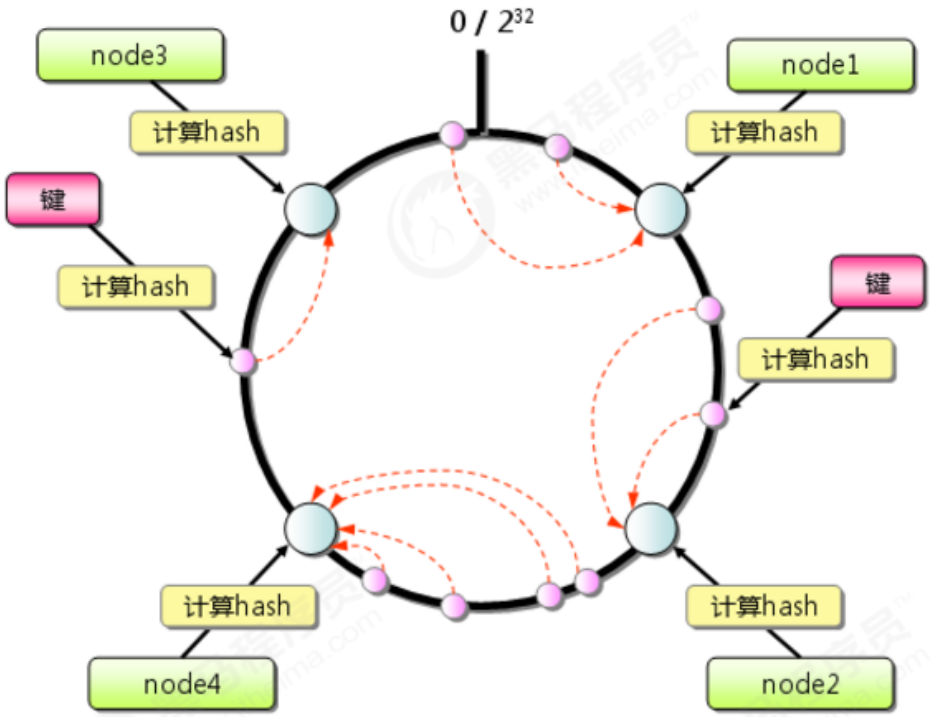
  - Failfast Cluster 快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

  - 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。

- 负载均衡：在集群负载均衡时，Dubbo提供多种均衡策略，缺省random随机调用
  - Random LoadBalance：按照权重设置随机概率，无状态
  - RoundRobin LoadBalance：轮询，有状态
  - LeastActive LoadBalance：最少活跃数随机，方法维度的统计服务调用数
  - ConsistentHash LoadBalance：一致性Hash



2) 服务治理

- 服务降级

  可以通过服务降级功能临时屏蔽某个出错的非关键服务，并定义降级后的返回策略

  可以向注册中心写入动态配置覆盖规则

```
RegistryFactory registryFactory =
ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtensi
on();
Registry registry =
registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));
registry.register(URL.valueOf("override://0.0.0.0/com.foo.BarService?
category=configurators&dynamic=false&application=foo&mock=force:return+null"
));
```

  - `mock=force:return+null` 表示消费方对该服务的方法调用都直接返回 null 值，不发起远程调用。用来屏蔽不重要服务不可用时对调用方的影响。
  - 还可以改为 `mock=fail:return+null` 表示消费方对该服务的方法调用在失败后，再返回 null 值，不抛异常。用来容忍不重要服务不稳定时对调用方的影响。

  dubbo配置服务降级方式

  - 在dubbo-admin中配置
- 整合hystrix

springboot官方提供了对hystrix的集成，直接在pom.xml里加入依赖

```
        <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
        <version>1.4.4.RELEASE</version>
    </dependency>
```

Application启动类中新增@EnableHystrix来启用hystrix starter

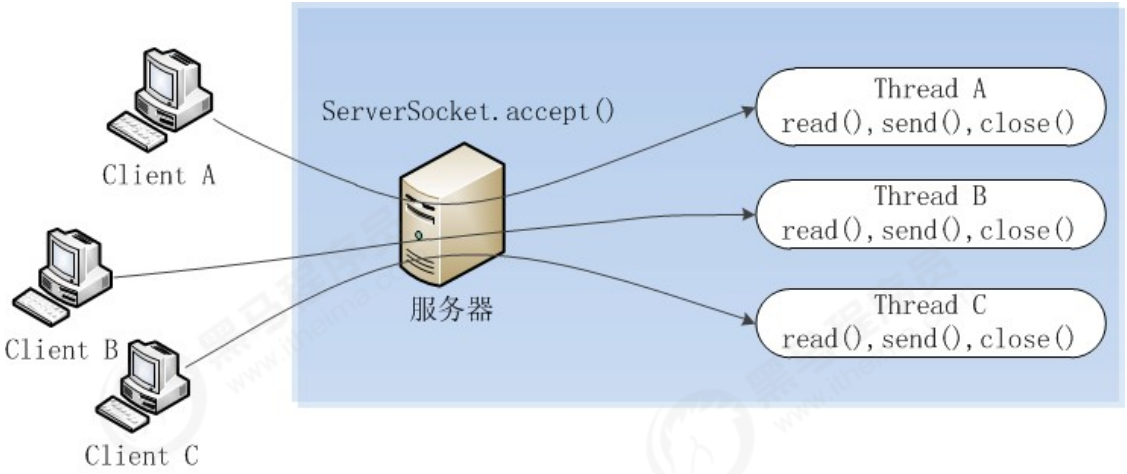
provider增加@HystrixCommand

Consumer method上配置@HystrixCommand(fallbackMethod= methodName))

3、dubbo线程IO模型

- BIO

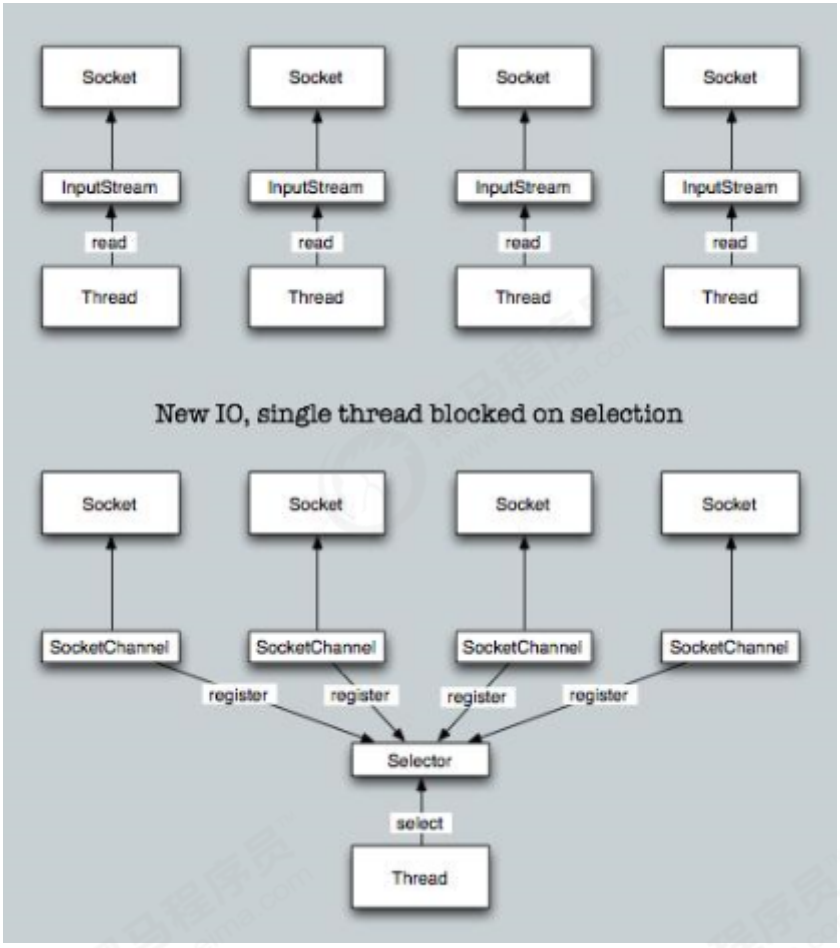
  每个客户端连接过来后，服务端都会启动一个线程去处理该客户端的请求。阻塞I/O的通信模型示意图如下：



在传统的IO模型中，每个连接创建成功之后都需要一个线程来维护。

- NIO

NIO，也叫做new-IO或者non-blocking-IO，可理解为非阻塞IO。NIO编程模型中，新来一个连接不再创建一个新的线程，而是可以把这条连接直接绑定到某个固定的线程，然后这条连接所有的读写都由这个线程来负责，我们用一幅图来对比一下IO与NIO：



如上图所示，IO模型中，一个连接都会创建一个线程，对应一个while死循环，死循环的目的就是不断监测这条连接上是否有数据可以读。但是在大多数情况下，1万个连接里面同一时刻只有少量的连接有数据可读，因此，很多个while死循环都白白浪费掉了，因为没有数据。

而在NIO模型中，可以把这么多的while死循环变成一个死循环，这个死循环由一个线程控制。这就是NIO模型中选择器（Selector）的作用，一条连接来了之后，现在不创建一个while死循环去监听是否有数据可读了，而是直接把这条连接注册到**选择器**上，通过检查这个**选择器**，就可以批量监测出有数据可读的连接，进而读取数据。

举个栗子，在一家餐厅里，客人有点菜的需求，一共有100桌客人，有两种方案可以解决客人点菜的问题：

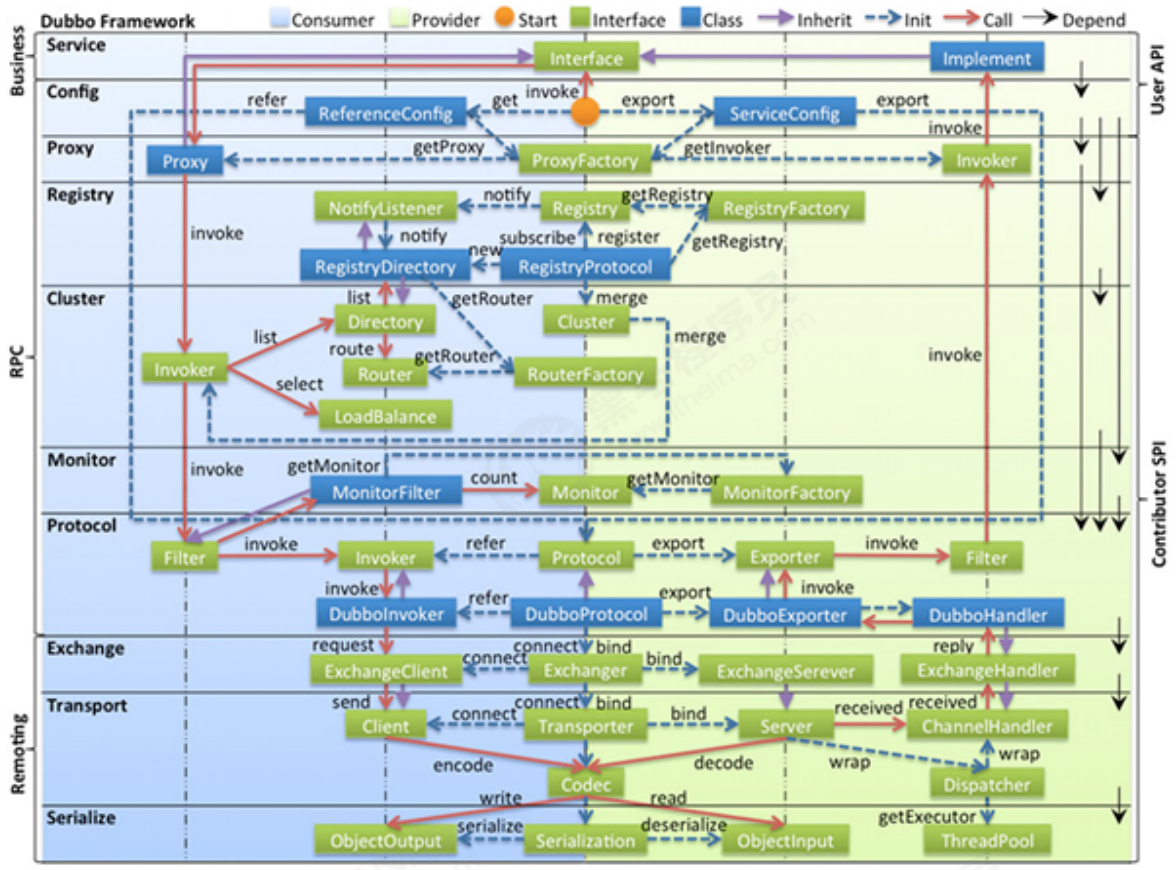
- 方案一：

  每桌客人配一个服务生，每个服务生就在餐桌旁给客人提供服务。如果客人要点菜，服务生就可以立刻提供点菜的服务。那么100桌客人就需要100个服务生提供服务，这就是IO模型，一个连接对应一个线程。

- 方案二：

  一个餐厅只有一个服务生（假设服务生可以忙的过来）。这个服务生隔段时间就询问所有的客人是否需要点菜，然后每一时刻处理所有客人的点菜要求。这就是NIO模型，所有客人都注册到同一个服务生，对应的就是所有的连接都注册到一个线程，然后批量轮询。

这就是NIO模型解决线程资源受限的方案，实际开发过程中，我们会开多个线程，每个线程都管理着一批连接，相对于IO模型中一个线程管理一条连接，消耗的线程资源大幅减少。



l config 配置层：对外配置接口，以 ServiceConfig, ReferenceConfig 为中心，可以直接初始化配置类，也可以通过 spring 解析配置生成配置类

l proxy 服务代理层：服务接口透明代理，生成服务的客户端 Stub 和服务器端 Skeleton, 以 ServiceProxy 为中心，扩展接口为 ProxyFactory

l registry 注册中心层：封装服务地址的注册与发现，以服务 URL 为中心，扩展接口为 RegistryFactory, Registry, RegistryService

l cluster 路由层：封装多个提供者的路由及负载均衡，并桥接注册中心，以 Invoker 为中心，扩展接口为 Cluster, Directory, Router, LoadBalance

monitor 监控层：RPC 调用次数和调用时间监控，以 Statistics 为中心，扩展接口为 MonitorFactory, Monitor, MonitorService

l protocol 远程调用层：封装 RPC 调用，以 Invocation, Result 为中心，扩展接口为 Protocol, Invoker, Exporter

l exchange 信息交换层：封装请求响应模式，同步转异步，以 Request, Response 为中心，扩展接口为 Exchanger, ExchangeChannel, ExchangeClient, ExchangeServer

transport 网络传输层：抽象 mina 和 netty 为统一接口，以 Message 为中心，扩展接口为 Channel, Transporter, Client, Server, Codec

l serialize 数据序列化层：可复用的一些工具，扩展接口为 Serialization, ObjectInput, ObjectOutput, ThreadPool

5、Dubbo 源码解析

类。正因此特性，我们可以很容易的通过 SPI 机制为我们的程序提供拓展功能。SPI 机制在第三方框架中也有所应用，比如 Dubbo 就是通过 SPI 机制加载所有的组件。不过，Dubbo 并未使用 Java 原生的 SPI 机制，而是对其进行了增强，使其能够更好的满足需求。在 Dubbo 中，SPI 是一个非常重要的模块。

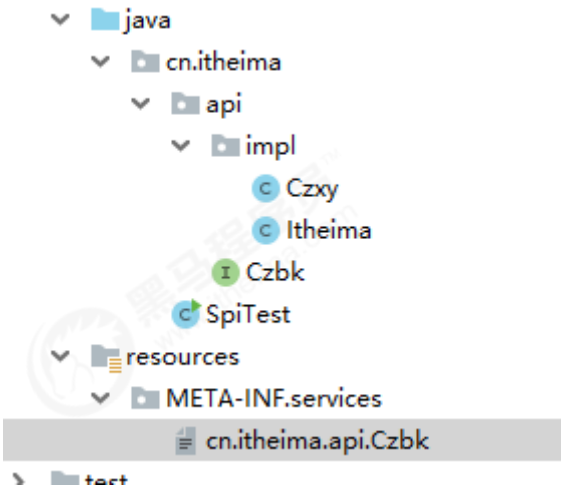
1、Dubbo SPI

1）java 原生的SPI机制

- 新建一个接口Czbk和两个实现类Czxy Itheima

```
//接口
public interface Czbk {
    void service();
}

//两个实现类
public class Czxy implements Czbk {
    @Override
    public void service() {
        System.out.println("上大学，来传智学院，一所不一样的大学，收获不一样的你");
    }
}

public class Itheima implements Czbk {
    @Override
    public void service() {
        System.out.println("学IT，到黑马");
    }
}
```

- 在resources中新建/META-INF/services目录，在新建的目录中新建文件，注意文件名必须是接口的全类名，此处为Czbk的接口全路径，文件内容为接口实现类的全类名，每一个实现类占一行

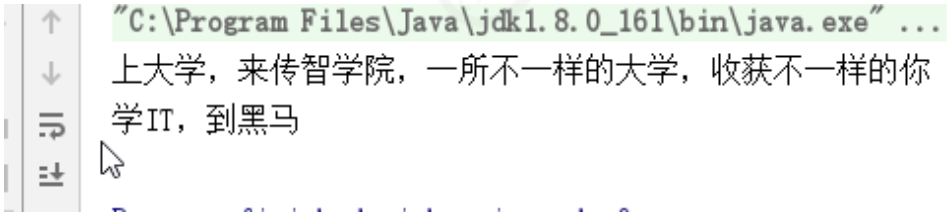



```
cn.itheima.api.impl.Czxy
cn.itheima.api.impl.Itheima
```

- 新建测试类SpiTest，主方法测试

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

        ServiceLoader<Czbk> serviceLoader = ServiceLoader.load(Czbk.class);
        Iterator<Czbk> czbkIterator = serviceLoader.iterator();
        while (czbkIterator.hasNext()) {
            Czbk czbk = czbkIterator.next();
            czbk.service();
        }
    }
}
```

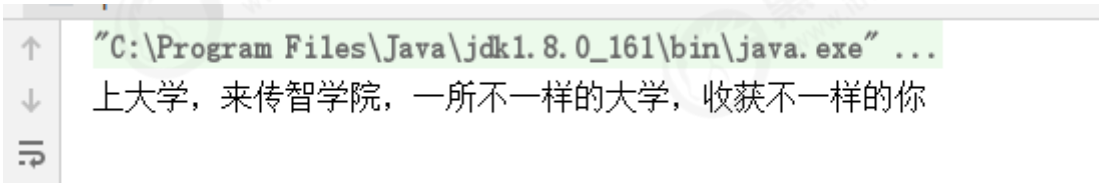
- 输出结果为：



如果在/META-INF/services/目录中的配置文件中修改为

```
cn.itheima.api.impl.Czxy
```

输出结果为



- 源码解析：

  ServiceLoader成员变量

```
// The class or interface representing the service being loaded
private final Class<S> service;

// The class loader used to locate, load, and instantiate providers
private final ClassLoader loader;

// The access control context taken when the ServiceLoader is created
private final AccessControlContext acc;

// Cached providers, in instantiation order
private LinkedHashMap<String,S> providers = new LinkedHashMap<>();

// The current lazy-lookup iterator
private LazyIterator lookupIterator;
```

调用serviceLoader.load方法，实例化loader类加载器、acc访问控制器、providers缓存加载成功的类、lookupIterator迭代器

在serviceLoad调用iterator()方法时，返回Iterator迭代器，迭代器重写hasNext()和next()方法

```
return new Iterator<S>() {

    Iterator<Map.Entry<String,S>> knownProviders
        = providers.entrySet().iterator();

    public boolean hasNext() {
        if (knownProviders.hasNext())
            return true;
        return lookupIterator.hasNext();
    }

    public S next() {
        if (knownProviders.hasNext())
            return knownProviders.next().getValue();
        return lookupIterator.next();
    }

    public void remove() { throw new UnsupportedOperationException(); }

};
```

通过iterator遍历时，调用hasNext()方法中，执行lookupIterator的hasNext()方法，在lookupIterator的hasNext中调用hasNextService方法

```
        if (acc == null) {
            return hasNextService();
        } else {
            PrivilegedAction<Boolean> action = new PrivilegedAction<Boolean>() {
                public Boolean run() { return hasNextService(); }
            };
            return AccessController.doPrivileged(action, acc);
        }
    }

    private boolean hasNextService() {
        if (nextName != null) {
            return true;
        }
        if (configs == null) {
            try {
                String fullName = PREFIX + service.getName();
                if (loader == null)
                    configs = ClassLoader.getSystemResources(fullName);
                else
                    configs = loader.getResources(fullName);
            } catch (IOException x) {
                fail(service, msg: "Error locating configuration files", x);
            }
        }
        while ((pending == null) || !pending.hasNext()) {
            if (!configs.hasMoreElements()) {
                return false;
            }
            pending = parse(service, configs.nextElement());
        }
        nextName = pending.next();
        return true;
    }
```

读取META-INF/services/下的配置文件，获得所有能被实例化的类的全类名，将全类名存入pending集合中，并将值赋值给nextName属性，接下来我们调用next()方法

```
            return nextService();
        } else {
            PrivilegedAction<S> action = new PrivilegedAction<S>() {
                public S run() { return nextService(); }
            };
            return AccessController.doPrivileged(action, acc);
        }
    }

    public void remove() { throw new UnsupportedOperationException(); }

    private S nextService() {
        if (!hasNextService())
            throw new NoSuchElementException();
        String cn = nextName;
        nextName = null;
        Class<?> c = null;
        try {
            c = Class.forName(cn, initialize: false, loader);
        } catch (ClassNotFoundException x) {
            fail(service,
                 msg: "Provider " + cn + " not found");
        }
        if (!service.isAssignableFrom(c)) {
            fail(service,
                 msg: "Provider " + cn  + " not a subtype");
        }
        try {
            S p = service.cast(c.newInstance());
            providers.put(cn, p);
            return p;
        } catch (Throwable x) {
            fail(service,
                 msg: "Provider " + cn + " could not be instantiated",
                 x);
        }
        throw new Error();          // This cannot happen
    }
```

将nextName属性赋值给cn,并创建Class对象，通过反射创建对象，并将实例化的对象缓存到providers中，同时将对对象返回给main方法，main方法获取到实例化的对象，调用其方法就能完成不同实现类的调用

java SPI的问题：

- 无法根据参数来获取所对应的实现类
    - 不能解决IOC、AOP的问题

基于以上问题，dubbo在java原生的SPI机制进行了增强，解决了以上问题。

2）dubbo SPI机制

Dubbo 重新实现了一套功能更强的 SPI 机制，Dubbo SPI 的相关逻辑被封装在了 ExtensionLoader 类中，通过 ExtensionLoader，我们可以加载指定的实现类。Dubbo SPI 所需的配置文件需放置在 META-INF/dubbo 路径下，配置内容如下

```
college=cn.itheima.api.impl.Czxy
shortTrain=cn.itheima.api.impl.Itheima
```

与 Java SPI 实现类配置不同，Dubbo SPI 是通过键值对的方式进行配置，这样我们可以按需加载指定的实现类。另外，在测试 Dubbo SPI 时，需要在 Robot 接口上标注 @SPI 注解。下面来演示 Dubbo SPI 的用法：

```
            ExtensionLoader<Czbk> extensionLoader =
ExtensionLoader.getExtensionLoader(Czbk.class);
        Czbk czbk = extensionLoader.getExtension("college");
        czbk.service();
```

运行结果为

```
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
上大学，来传智学院，一所不一样的大学，收获不一样的你
```

修改main方法getExtension的参数为

```
Czbk czbk = extensionLoader.getExtension("shortTrain");
```

运行结果为

```
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
学IT，到黑马
```

dubbo SPI源码解析：

通过 ExtensionLoader 的 getExtensionLoader 方法获取一个 ExtensionLoader 实例，该方法方法先从缓存中获取与拓展类对应的 ExtensionLoader，若缓存未命中，则创建一个新的实例

```
} else {
    ExtensionLoader<T> loader = (ExtensionLoader)EXTENSION_LOADERS.get(type);
    if (loader == null) {
        EXTENSION_LOADERS.putIfAbsent(type, new ExtensionLoader(type));
        loader = (ExtensionLoader)EXTENSION_LOADERS.get(type);
    }
```

```
        Holder<Object> holder = this.getOrCreateHolder(name);
        Object instance = holder.get();
        if (instance == null) {
            synchronized(holder) {
                instance = holder.get();
                if (instance == null) {
                    instance = this.createExtension(name);
                    holder.set(instance);
                }
            }
        }

        return instance;
```

createExtension方法包含如下步骤

- 通过 getExtensionClasses 获取所有的拓展类
- 通过反射创建拓展对象
- 向拓展对象中注入依赖
- 将拓展对象包裹在相应的 Wrapper 对象中

```
private T createExtension(String name) {
    // 从配置文件中加载所有的拓展类，可得到“配置项名称”到“配置类”的映射关系表
    Class<?> clazz = getExtensionClasses().get(name);
    if (clazz == null) {
        throw findException(name);
    }
    try {
        T instance = (T) EXTENSION_INSTANCES.get(clazz);
        if (instance == null) {
            // 通过反射创建实例
            EXTENSION_INSTANCES.putIfAbsent(clazz, clazz.newInstance());
            instance = (T) EXTENSION_INSTANCES.get(clazz);
        }
        // 向实例中注入依赖
        injectExtension(instance);
        Set<Class<?>> wrapperClasses = cachedWrapperClasses;
        if (wrapperClasses != null && !wrapperClasses.isEmpty()) {
            // 循环创建 Wrapper 实例
            for (Class<?> wrapperClass : wrapperClasses) {
                // 将当前 instance 作为参数传给 Wrapper 的构造方法，并通过反射创建
Wrapper 实例。
                // 然后向 Wrapper 实例中注入依赖，最后将 Wrapper 实例再次赋值给
instance 变量
                instance = injectExtension(
                    (T)
wrapperClass.getConstructor(type).newInstance(instance));
            }
        }
        return instance;
```

```
    }
}
```

此处injectExtension方法依赖注入实现原理：Dubbo 首先会通过反射获取到实例的所有方法，然后再遍历方法列表，检测方法名是否具有 setter 方法特征。若有，则通过 ObjectFactory 获取依赖对象，最后通过反射调用 setter 方法将依赖设置到目标对象中。

```
private T injectExtension(T instance) {
    try {
        if (objectFactory != null) {
            // 遍历目标类的所有方法
            for (Method method : instance.getClass().getMethods()) {
                // 检测方法是否以 set 开头，且方法仅有一个参数，且方法访问级别为 public
                if (method.getName().startsWith("set")
                    && method.getParameterTypes().length == 1
                    && Modifier.isPublic(method.getModifiers())) {
                    // 获取 setter 方法参数类型
                    Class<?> pt = method.getParameterTypes()[0];
                    try {
                        // 获取属性名，比如 setName 方法对应属性名 name
                        String property = method.getName().length() > 3 ?
                            method.getName().substring(3, 4).toLowerCase() +
                                method.getName().substring(4) : "";
                        // 从 ObjectFactory 中获取依赖对象
                        Object object = objectFactory.getExtension(pt,
property);
                        if (object != null) {
                            // 通过反射调用 setter 方法设置依赖
                            method.invoke(instance, object);
                        }
                    } catch (Exception e) {
                        logger.error("fail to inject via method...");
                    }
                }
            }
        }
    } catch (Exception e) {
        logger.error(e.getMessage(), e);
    }
    return instance;
}
```

第一个步骤中获取所有的扩展类方法getExtensionClasses，该方法先检查缓存，若缓存未命中，则通过 synchronized 加锁。加锁后再次检查缓存，并判空。此时如果 classes 仍为 null，则通过 loadExtensionClasses 加载拓展类

```
private Map<String, Class<?>> getExtensionClasses() {
    // 从缓存中获取已加载的拓展类
    Map<String, Class<?>> classes = cachedClasses.get();
    // 双重检查
    if (classes == null) {
        synchronized (cachedClasses) {
```

```
                // 加载拓展类
                classes = loadExtensionClasses();
                cachedClasses.set(classes);
            }
        }
    }
    return classes;
}
```

loadExtensionClasses 方法做了两件事，一是解析SPI注解，二是调用 loadDirectory 方法加载指定文件夹配置文件

```
private Map<String, Class<?>> loadExtensionClasses() {
    // 获取 SPI 注解，这里的 type 变量是在调用 getExtensionLoader 方法时传入的
    final SPI defaultAnnotation = type.getAnnotation(SPI.class);
    if (defaultAnnotation != null) {
        String value = defaultAnnotation.value();
        if ((value = value.trim()).length() > 0) {
            // 对 SPI 注解内容进行切分
            String[] names = NAME_SEPARATOR.split(value);
            // 检测 SPI 注解内容是否合法，不合法则抛出异常
            if (names.length > 1) {
                throw new IllegalStateException("more than 1 default extension
name on extension...");
            }

            // 设置默认名称，参考 getDefaultExtension 方法
            if (names.length == 1) {
                cachedDefaultName = names[0];
            }
        }
    }

    Map<String, Class<?>> extensionClasses = new HashMap<String, Class<?>>();
    // 加载指定文件夹下的配置文件
    loadDirectory(extensionClasses, DUBBO_INTERNAL_DIRECTORY);
    loadDirectory(extensionClasses, DUBBO_DIRECTORY);
    loadDirectory(extensionClasses, SERVICES_DIRECTORY);
    return extensionClasses;
}
```

loadDirectory 方法获取classLoader ， 通过classLoader获取URL资源信息，遍历URL通过loadResource加载资源

```
private void loadDirectory(Map<String, Class<?>> extensionClasses, String dir) {
    // fileName = 文件夹路径 + type 全限定名
    String fileName = dir + type.getName();
    try {
        Enumeration<java.net.URL> urls;
        ClassLoader classLoader = findClassLoader();
        // 根据文件名加载所有的同名文件
        if (classLoader != null) {
            urls = classLoader.getResources(fileName);
```

```
        }
        if (urls != null) {
            while (urls.hasMoreElements()) {
                java.net.URL resourceURL = urls.nextElement();
                // 加载资源
                loadResource(extensionClasses, classLoader, resourceURL);
            }
        }
    } catch (Throwable t) {
        logger.error("...");
    }
}
```

loadResource 方法用于读取和解析配置文件，并通过反射加载类，最后调用 loadClass 方法

```
private void loadResource(Map<String, Class<?>> extensionClasses,
    ClassLoader classLoader, java.net.URL resourceURL) {
    try {
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(resourceURL.openStream(), "utf-8"));
        try {
            String line;
            // 按行读取配置内容
            while ((line = reader.readLine()) != null) {
                // 定位 # 字符
                final int ci = line.indexOf('#');
                if (ci >= 0) {
                    // 截取 # 之前的字符串，# 之后的内容为注释，需要忽略
                    line = line.substring(0, ci);
                }
                line = line.trim();
                if (line.length() > 0) {
                    try {
                        String name = null;
                        int i = line.indexOf('=');
                        if (i > 0) {
                            // 以等于号 = 为界，截取键与值
                            name = line.substring(0, i).trim();
                            line = line.substring(i + 1).trim();
                        }
                        if (line.length() > 0) {
                            // 加载类，并通过 loadClass 方法对类进行缓存
                            loadClass(extensionClasses, resourceURL,
                                    Class.forName(line, true, classLoader),
name);
                        }
                    } catch (Throwable t) {
                        IllegalStateException e = new
IllegalStateException("Failed to load extension class...");
                    }
                }
            }
        } finally {
            reader.close();
        }
```

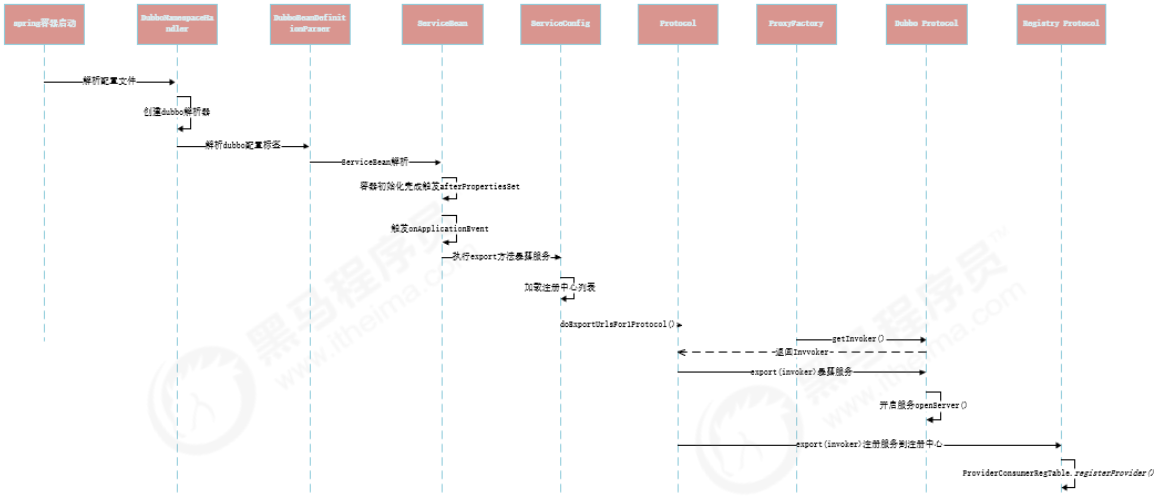
```
    }
}
```

loadClass方法主要用用用于操作缓存

```
private void loadClass(Map<String, Class<?>> extensionClasses, java.net.URL
resourceURL,
    Class<?> clazz, String name) throws NoSuchMethodException {

    if (!type.isAssignableFrom(clazz)) {
        throw new IllegalStateException("...");
    }

    // 检测目标类上是否有 Adaptive 注解
    if (clazz.isAnnotationPresent(Adaptive.class)) {
        if (cachedAdaptiveClass == null) {
            // 设置 cachedAdaptiveClass缓存
            cachedAdaptiveClass = clazz;
        } else if (!cachedAdaptiveClass.equals(clazz)) {
            throw new IllegalStateException("...");
        }

    // 检测 clazz 是否是 Wrapper 类型
    } else if (isWrapperClass(clazz)) {
        Set<Class<?>> wrappers = cachedWrapperClasses;
        if (wrappers == null) {
            cachedWrapperClasses = new ConcurrentHashSet<Class<?>>();
            wrappers = cachedWrapperClasses;
        }
        // 存储 clazz 到 cachedWrapperClasses 缓存中
        wrappers.add(clazz);

    // 程序进入此分支，表明 clazz 是一个普通的拓展类
    } else {
        // 检测 clazz 是否有默认的构造方法，如果没有，则抛出异常
        clazz.getConstructor();
        if (name == null || name.length() == 0) {
            // 如果 name 为空，则尝试从 Extension 注解中获取 name，或使用小写的类名作为
name
            name = findAnnotationName(clazz);
            if (name.length() == 0) {
                throw new IllegalStateException("...");
            }
        }
        // 切分 name
        String[] names = NAME_SEPARATOR.split(name);
        if (names != null && names.length > 0) {
            Activate activate = clazz.getAnnotation(Activate.class);
            if (activate != null) {
                // 如果类上有 Activate 注解，则使用 names 数组的第一个元素作为键，
                // 存储 name 到 Activate 注解对象的映射关系
                cachedActivates.put(names[0], activate);
            }
            for (String n : names) {
```

```
                    cachedNames.put(clazz, n);
                }
                Class<?> c = extensionClasses.get(n);
                if (c == null) {
                    // 存储名称到 Class 的映射关系
                    extensionClasses.put(n, clazz);
                } else if (c != clazz) {
                    throw new IllegalStateException("...");
                }
            }
        }
    }
}
```

2、dubbo服务暴露



spring容器启动，会加载BeanDefinitionParser类来解析配置文件，dubbo配置文件的加载依赖实现类DubboBeanDefinitionParser，DubboBeanDefinitionParser解析器会将配置文件中不同的标签解析成不同的xxxConfig，<dubbo:service/> 、<dubbo:reference/>分别解析成serviceBean和referenceBean

```
public void init() {
    this.registerBeanDefinitionParser( elementName: "application", new DubboBeanDefinitionParser(ApplicationConfig.class, required: true));
    this.registerBeanDefinitionParser( elementName: "module", new DubboBeanDefinitionParser(ModuleConfig.class, required: true));
    this.registerBeanDefinitionParser( elementName: "registry", new DubboBeanDefinitionParser(RegistryConfig.class, required: true));
    this.registerBeanDefinitionParser( elementName: "config-center", new DubboBeanDefinitionParser(ConfigCenterBean.class, required: true));
    this.registerBeanDefinitionParser( elementName: "metadata-report", new DubboBeanDefinitionParser(MetadataReportConfig.class, required: true))
    this.registerBeanDefinitionParser( elementName: "monitor", new DubboBeanDefinitionParser(MonitorConfig.class, required: true));
    this.registerBeanDefinitionParser( elementName: "metrics", new DubboBeanDefinitionParser(MetricsConfig.class, required: true));
    this.registerBeanDefinitionParser( elementName: "provider", new DubboBeanDefinitionParser(ProviderConfig.class, required: true));
    this.registerBeanDefinitionParser( elementName: "consumer", new DubboBeanDefinitionParser(ConsumerConfig.class, required: true));
    this.registerBeanDefinitionParser( elementName: "protocol", new DubboBeanDefinitionParser(ProtocolConfig.class, required: true));
    this.registerBeanDefinitionParser( elementName: "service", new DubboBeanDefinitionParser(ServiceBean.class, required: true));
    this.registerBeanDefinitionParser( elementName: "reference", new DubboBeanDefinitionParser(ReferenceBean.class, required: false));
    this.registerBeanDefinitionParser( elementName: "annotation", new AnnotationBeanDefinitionParser());
}
```

```
<!-- 使用zookeeper注册中心暴露服务地址 -->
<dubbo:registry address="zookeeper://127.0.0.1:2181" />
<!-- 用dubbo协议在20880端口暴露服务 -->
<dubbo:protocol name="dubbo" port="20880" />
<!-- 声明需要暴露的服务接口 -->
<dubbo:service interface="com.itheima.demo.DemoService" ref="demoService" />
<!-- 和本地bean一样实现服务 -->
<bean id="demoService" class="com.itheima.demo.DemoServiceImpl" />
```

serviceBean实现了InitializingBean和ApplicationListener接口，在afterPropertiesSet方法中主要将配置文件中的属性依次配置到对应的bean中，在 Spring 上下文刷新事件后会回调onApplicationEvent方法

```
public void onApplicationEvent(ContextRefreshedEvent event) {
    if (!isExported() && !isUnexported()) {
        if (logger.isInfoEnabled()) {
            logger.info( msg: "The service ready on spring started. service: " + getInterface())
        }
        export();
    }
}
```

服务暴露导出

调用父类ServiceConfig对象的export方法，检查延迟和是否导出，执行doExportUrls方法

```
protected synchronized void doExport() {
    if (unexported) {
        throw new IllegalStateException("The service " + interfaceClass.getName() + "
    }
    if (exported) {
        return;
    }
    exported = true;

    if (StringUtils.isEmpty(path)) {
        path = interfaceName;
    }
    doExportUrls();
}
```

导出URL

doExportUrls方法首先是通过 loadRegistries 加载注册中心链接，后再遍历 ProtocolConfig 集合导出每个服务。并在导出服务的过程中，将服务注册到注册中心。下面，我们先来看一下 loadRegistries 方法的逻辑

```
private void doExportUrls() {
    List<URL> registryURLs = loadRegistries( provider: true);
    for (ProtocolConfig protocolConfig : protocols) {
        String pathKey = URL.buildKey(getContextPath(protocolConfig).map(p -> p + "/" + path)
        ProviderModel providerModel = new ProviderModel(pathKey, ref, interfaceClass);
        ApplicationModel.initProviderModel(pathKey, providerModel);
        doExportUrlsForlProtocol(protocolConfig, registryURLs);
    }
}
```

```
protected List<URL> loadRegistries(boolean provider) {
```

```
        if (CollectionUtils.isNotEmpty(registries)) {
            for (RegistryConfig config : registries) {
                String address = config.getAddress();
                if (StringUtils.isEmpty(address)) {
                 // 若 address 为空，则将其设为 0.0.0.0
                    address = ANYHOST_VALUE;
                }
                if (!RegistryConfig.NO_AVAILABLE.equalsIgnoreCase(address)) {
                    Map<String, String> map = new HashMap<String, String>();
                    appendParameters(map, application);
                    appendParameters(map, config);
                    map.put(PATH_KEY, RegistryService.class.getName());
                    appendRuntimeParameters(map);
                    if (!map.containsKey(PROTOCOL_KEY)) {
                        map.put(PROTOCOL_KEY, DUBBO_PROTOCOL);
                    }
                    // 解析得到 URL 列表，address 可能包含多个注册中心 ip，
                    // 因此解析得到的是一个 URL 列表
                    List<URL> urls = UrlUtils.parseURLs(address, map);

                    for (URL url : urls) {
                       // 将 URL 协议头设置为 registry
                        url = URLBuilder.from(url)
                                .addParameter(REGISTRY_KEY, url.getProtocol())
                                .setProtocol(REGISTRY_PROTOCOL)
                                .build();
                        // 通过判断条件，决定是否添加 url 到 registryList 中，条件如
下：
                        // (服务提供者 && register = true 或 null)
                        //    || (非服务提供者 && subscribe = true 或 null)
                        if ((provider && url.getParameter(REGISTER_KEY, true))
                                || (!provider && url.getParameter(SUBSCRIBE_KEY,
true))) {
                            registryList.add(url);
                        }
                    }
                }
            }
        }
        return registryList;
    }
```

doExportUrlsFor1Protocol方法主要将版本、时间戳、方法名以及各种配置对象的字段信息放入到 map 中，map 中的内容将作为 URL 的查询字符串。构建好 map 后，紧接着是获取上下文路径、主机名以及端口号等信息，最后将 map 和主机名等数据传给 URL 构造方法创建 URL 对象

```
// export service
String host = this.findConfigedHosts(protocolConfig, registryURLs, map);
Integer port = this.findConfigedPorts(protocolConfig, name, map);
URL url = new URL(name, host, port, getContextPath(protocolConfig).map(p -> p + "/" + path).orElse(path), map);
```

导出服务

```
// scope=none 不做处理
if (!SCOPE_NONE.equalsIgnoreCase(scope)) {
```

```
    // scope !=remote 导出本地服务
    if (!SCOPE_REMOTE.equalsIgnoreCase(scope)) {
        exportLocal(url);
    }
    // export to remote if the config is not local (export to local only when
config is local)
    // scope!=local 导出远程服务
    if (!SCOPE_LOCAL.equalsIgnoreCase(scope)) {
        if (!isOnlyInJvm() && logger.isInfoEnabled()) {
            logger.info("Export dubbo service " + interfaceClass.getName() + "
to url " + url);
        }
        if (CollectionUtils.isNotEmpty(registryURLs)) {
            for (URL registryURL : registryURLs) {
                //if protocol is only injvm ,not register
                if (LOCAL_PROTOCOL.equalsIgnoreCase(url.getProtocol())) {
                    continue;
                }
                url = url.addParameterIfAbsent(DYNAMIC_KEY,
registryURL.getParameter(DYNAMIC_KEY));
                URL monitorUrl = loadMonitor(registryURL);
                if (monitorUrl != null) {
                    url = url.addParameterAndEncoded(MONITOR_KEY,
monitorUrl.toFullString());
                }
                if (logger.isInfoEnabled()) {
                    logger.info("Register dubbo service " +
interfaceClass.getName() + " url " + url + " to registry " + registryURL);
                }

                // For providers, this is used to enable custom proxy to
generate invoker
                String proxy = url.getParameter(PROXY_KEY);
                if (StringUtils.isNotEmpty(proxy)) {
                    registryURL = registryURL.addParameter(PROXY_KEY, proxy);
                }
                //为服务提供类(ref)生成Invoker
                Invoker<?> invoker = PROXY_FACTORY.getInvoker(ref, (Class)
interfaceClass, registryURL.addParameterAndEncoded(EXPORT_KEY,
url.toFullString()));
                DelegateProviderMetaDataInvoker wrapperInvoker = new
DelegateProviderMetaDataInvoker(invoker, this);
                //导出服务，生成Exporter
                Exporter<?> exporter = protocol.export(wrapperInvoker);
                exporters.add(exporter);
            }
        } else {//不存在注册中心，仅导出服务
            Invoker<?> invoker = PROXY_FACTORY.getInvoker(ref, (Class)
interfaceClass, url);
            DelegateProviderMetaDataInvoker wrapperInvoker = new
DelegateProviderMetaDataInvoker(invoker, this);

            Exporter<?> exporter = protocol.export(wrapperInvoker);
            exporters.add(exporter);
        }
        /**
```

```
         */
        MetadataReportService metadataReportService = null;
        if ((metadataReportService = getMetadataReportService()) != null) {
            metadataReportService.publishProvider(url);
        }
    }
}
```

无论是导出服务到本地还是远程都需要创建Invoker对象，Invoker是ProxyFactory 代理工厂创建的对象，invoke封装了调用实体。然后根据 scope 参数，决定导出服务到本地还是导出到远程。在这里我们重点讨论到导出服务到远程，其中包含服务导出和服务注册两个过程。

```
 public <T> Exporter<T> export(final Invoker<T> originInvoker) throws
RpcException {
        //获取注册中心 URL
        URL registryUrl = getRegistryUrl(originInvoker);
        // url to export locally
        URL providerUrl = getProviderUrl(originInvoker);

        // Subscribe the override data
        // FIXME When the provider subscribes, it will affect the scene : a
certain JVM exposes the service and call
        //  the same service. Because the subscribed is cached key with the name
of the service, it causes the
        //  subscription information to cover.
        final URL overrideSubscribeUrl = getSubscribedOverrideUrl(providerUrl);
        final OverrideListener overrideSubscribeListener = new
OverrideListener(overrideSubscribeUrl, originInvoker);
        overrideListeners.put(overrideSubscribeUrl, overrideSubscribeListener);

        providerUrl = overrideUrlWithConfig(providerUrl,
overrideSubscribeListener);
        //export invoker 导出服务
        final ExporterChangeableWrapper<T> exporter =
doLocalExport(originInvoker, providerUrl);

        // url to registry 根据 URL 加载 Registry 实现类
        final Registry registry = getRegistry(originInvoker);
        final URL registeredProviderUrl = getRegisteredProviderUrl(providerUrl,
registryUrl);
        //注册服务
        ProviderInvokerWrapper<T> providerInvokerWrapper =
ProviderConsumerRegTable.registerProvider(originInvoker,
                registryUrl, registeredProviderUrl);
        //to judge if we need to delay publish
        boolean register = registeredProviderUrl.getParameter("register", true);
        if (register) {
            register(registryUrl, registeredProviderUrl);
            providerInvokerWrapper.setReg(true);
        }

        // Deprecated! Subscribe to override rules in 2.6.x or before.
        registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener);

        exporter.setRegisterUrl(registeredProviderUrl);
```

```
        return new DestroyableExporter<>(exporter);
    }
```

在doLocalExport方法中导出服务，其中包含创建DubboExporter 和openServer

```
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    URL url = invoker.getUrl();

    // export service.
    String key = serviceKey(url);
    DubboExporter<T> exporter = new DubboExporter<T>(invoker, key, exporterMap);
    exporterMap.put(key, exporter);

    //export an stub service for dispatching event
    Boolean isStubSupportEvent = url.getParameter(STUB_EVENT_KEY,
DEFAULT_STUB_EVENT);
    Boolean isCallbackservice = url.getParameter(IS_CALLBACK_SERVICE, false);
    if (isStubSupportEvent && !isCallbackservice) {
        String stubServiceMethods = url.getParameter(STUB_EVENT_METHODS_KEY);
        if (stubServiceMethods == null || stubServiceMethods.length() == 0) {
            if (logger.isWarnEnabled()) {
                logger.warn(new IllegalStateException("consumer [" +
url.getParameter(INTERFACE_KEY) +
                        "], has set stubproxy support event ,but no stub methods
founded."));
            }

        } else {
            stubServiceMethodsMap.put(url.getServiceKey(), stubServiceMethods);
        }
    }

    openServer(url);
    optimizeSerialization(url);

    return exporter;
}
```

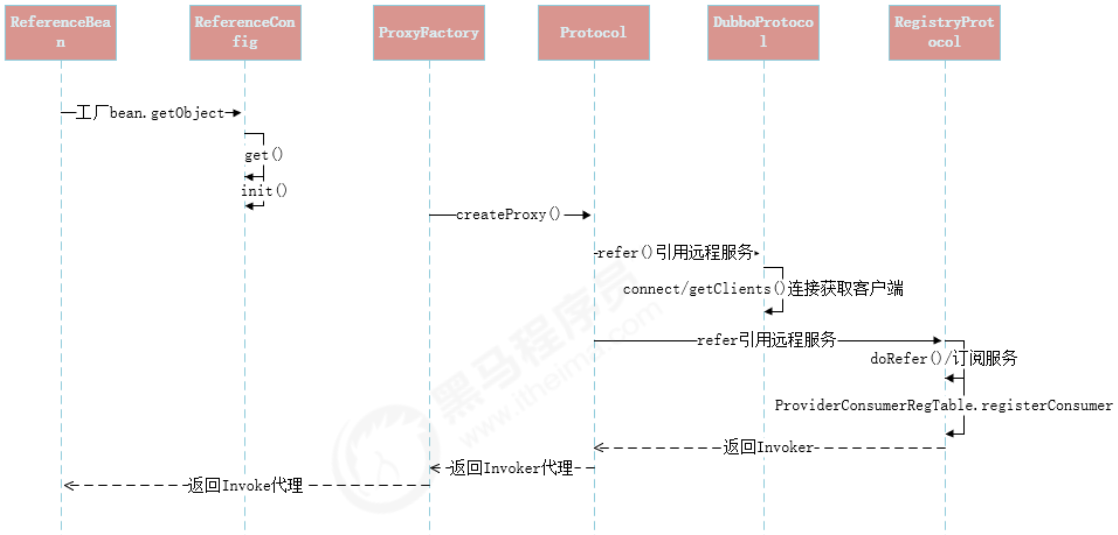
openServer方法中通过createServer方法创建服务实例

```
String key = url.getAddress();
//client can export a service which's only for server to invoke
boolean isServer = url.getParameter(IS_SERVER_KEY, defaultValue: true);
if (isServer) {
    ExchangeServer server = serverMap.get(key);
    if (server == null) {
        synchronized (this) {
            server = serverMap.get(key);
            if (server == null) {
                serverMap.put(key, createServer(url));
            }
        }
    } else {
        // server supports reset, use together with override
        server.reset(url);
    }
```

dubbo服务实例默认使用NettyServer，参考源码分析

服务注册调用getRegistry方法，创建连接注册中心，调用register方法注册服务，如果是zookeeper做为注册中心，调用zookeeper客户端创建服务节点，服务注册成功，在ZooInspector可以查看注册服务的节点数据

3、服务引入



服务引入原理

- dubbo服务引入时机

Dubbo 服务引用的时机有两个，第一个是在 Spring 容器调用 ReferenceBean 的 afterPropertiesSet 方法时引用服务，第二个是在 ReferenceBean 对应的服务被注入到其他类中时引用。这两个引用服务的时机区别在于，第一个是饿汉式的，第二个是懒汉式的。默认情况下，Dubbo 使用懒汉式引用服务。如果需要使用饿汉式，可通过配置 dubbo:reference 的 init 属性开启

当我们的服务被注入到其他类中时，Spring 会第一时间调用 getObject 方法，并由该方法执行服务引用逻辑。按照惯例，在进行具体工作之前，需先进行配置检查与收集工作。接着根据收集到的信息决定服务用的方式，有三种，第一种是引用本地 (JVM) 服务，第二是通过直连方式引用远程服务，第三是通过注册中心引用远程服务，管是哪种引用方式，最后都会得到一个 Invoker 实例。如果有多个注册中心，多个服务提供者，这个时候会得到一组 Invoker 实例，此时需要通过集群管理类 Cluster 将多个 Invoker 合并成一个实例。合并后的 Invoker 实例已经具备调用本地或远程服务的能力了，但并不能将此实例暴露给用户使用，这会对用户业务代码造成侵入。此时框架还需要通过代理工厂类 (ProxyFactory) 为服务接口生成代理类，并让代理类去调用 Invoker 逻辑。避免了 Dubbo 框架代码对业务代码的侵入。

服务引用的入口方法为 ReferenceBean 的 getObject 方法，该方法定义在 Spring 的 FactoryBean 接口中，ReferenceBean 实现了这个方法

```
    public Object getObject() {
        return get();
    }

    public synchronized T get() {
        checkAndUpdateSubConfigs();

        if (destroyed) {
            throw new IllegalStateException("The invoker of ReferenceConfig(" +
url + ")             has already destroyed!");
        }
        if (ref == null) {
            init();
        }
        return ref;
    }
```

Dubbo 在引用或导出服务时，首先会对这些配置进行检查和处理，以保证配置的正确性。配置解析逻辑封装在 ReferenceConfig 的 init 方法中

```
private void init() {
    //避免重复初始化
    if (initialized) {
        return;
    }
    checkStubAndLocal(interfaceClass);
    checkMock(interfaceClass);
    Map<String, String> map = new HashMap<String, String>();
    //添加 side、协议等信息到 map 中
    map.put(SIDE_KEY, CONSUMER_SIDE);

    appendRuntimeParameters(map);
    if (!isGeneric()) {
        String revision = Version.getVersion(interfaceClass, version);
        if (revision != null && revision.length() > 0) {
            map.put(REVISION_KEY, revision);
        }

        String[] methods = Wrapper.getWrapper(interfaceClass).getMethodNames();
        if (methods.length == 0) {
```

```
            map.put(METHODS_KEY, ANY_VALUE);
        } else {
            map.put(METHODS_KEY, StringUtils.join(new HashSet<String>
(Arrays.asList(methods)), COMMA_SEPARATOR));
        }
    }
    map.put(INTERFACE_KEY, interfaceName);
    // 将 ApplicationConfig、ConsumerConfig、ReferenceConfig 等对象的字段信息添加到
map 中
    appendParameters(map, metrics);
    appendParameters(map, application);
    appendParameters(map, module);
    // remove 'default.' prefix for configs from ConsumerConfig
    // appendParameters(map, consumer, Constants.DEFAULT_KEY);
    appendParameters(map, consumer);
    appendParameters(map, this);
    Map<String, Object> attributes = null;
    if (CollectionUtils.isNotEmpty(methods)) {
        attributes = new HashMap<String, Object>();
        //遍历 MethodConfig 列表
        for (MethodConfig methodConfig : methods) {
            appendParameters(map, methodConfig, methodConfig.getName());
            String retryKey = methodConfig.getName() + ".retry";
            //// 检测 map 是否包含 methodName.retry
            if (map.containsKey(retryKey)) {
                String retryValue = map.remove(retryKey);
                if ("false".equals(retryValue)) {
                    // 添加重试次数配置 methodName.retries
                    map.put(methodConfig.getName() + ".retries", "0");
                }
            }

            attributes.put(methodConfig.getName(),
            convertMethodConfig2AyncInfo(methodConfig));
        }
    }
    //获取服务消费者ip地址
    String hostToRegistry = ConfigUtils.getSystemProperty(DUBBO_IP_TO_REGISTRY);
    if (StringUtils.isEmpty(hostToRegistry)) {
        hostToRegistry = NetUtils.getLocalHost();
    } else if (isInvalidLocalHost(hostToRegistry)) {
        throw new IllegalArgumentException("Specified invalid registry ip from
property:" + DUBBO_IP_TO_REGISTRY + ", value:" + hostToRegistry);
    }
    map.put(REGISTER_IP_KEY, hostToRegistry);
    //创建代理类
    ref = createProxy(map);

    String serviceKey = URL.buildKey(interfaceName, group, version);
    ApplicationModel.initConsumerModel(serviceKey,
buildConsumerModel(serviceKey, attributes));
    initialized = true;
}
```

```
private T createProxy(Map<String, String> map) {
        //本地引入服务
        if (shouldJvmRefer(map)) {
            URL url = new URL(LOCAL_PROTOCOL, LOCALHOST_VALUE, 0,
interfaceClass.getName()).addParameters(map);
            invoker = REF_PROTOCOL.refer(interfaceClass, url);
            if (logger.isInfoEnabled()) {
                logger.info("Using injvm service " + interfaceClass.getName());
            }
        } else {//远程引入服务
            urls.clear(); // reference retry init will add url to urls, lead to
OOM
            //直连
            if (url != null && url.length() > 0) { // user specified URL, could
be peer-to-peer address, or register center's address.
                String[] us = SEMICOLON_SPLIT_PATTERN.split(url);
                if (us != null && us.length > 0) {
                    for (String u : us) {
                        URL url = URL.valueOf(u);
                        if (StringUtils.isEmpty(url.getPath())) {
                            url = url.setPath(interfaceName);
                        }
                        if (REGISTRY_PROTOCOL.equals(url.getProtocol())) {
                            urls.add(url.addParameterAndEncoded(REFER_KEY,
StringUtils.toQueryString(map)));
                        } else {
                            urls.add(ClusterUtils.mergeUrl(url, map));
                        }
                    }
                }
            } else { // 加载注册中心url assemble URL from register center's
configuration
                // if protocols not injvm checkRegistry
                if (!LOCAL_PROTOCOL.equalsIgnoreCase(getProtocol())){
                    checkRegistry();
                    List<URL> us = loadRegistries(false);
                    if (CollectionUtils.isNotEmpty(us)) {
                        for (URL u : us) {
                            URL monitorUrl = loadMonitor(u);
                            if (monitorUrl != null) {
                                map.put(MONITOR_KEY,
URL.encode(monitorUrl.toFullString()));
                            }
                            urls.add(u.addParameterAndEncoded(REFER_KEY,
StringUtils.toQueryString(map)));
                        }
                    }
                    //为配置注册中心，抛出异常
                    if (urls.isEmpty()) {
                        throw new IllegalStateException("No such any registry to
reference " + interfaceName + " on the consumer " + NetUtils.getLocalHost() + "
use dubbo version " + Version.getVersion() + ", please config <dubbo:registry
address=\"...\" /> to your spring config.");
                    }
```

```
            //单注册中心
            if (urls.size() == 1) {
                invoker = REF_PROTOCOL.refer(interfaceClass, urls.get(0));
            } else {//多注册中心
                List<Invoker<?>> invokers = new ArrayList<Invoker<?>>();
                URL registryURL = null;
                for (URL url : urls) {
                    invokers.add(REF_PROTOCOL.refer(interfaceClass, url));
                    if (REGISTRY_PROTOCOL.equals(url.getProtocol())) {
                        registryURL = url; // use last registry url
                    }
                }
                if (registryURL != null) { // registry url is available
                    // use RegistryAwareCluster only when register's CLUSTER is
available
                    URL u = registryURL.addParameter(CLUSTER_KEY,
RegistryAwareCluster.NAME);
                    // The invoker wrap relation would be:
RegistryAwareClusterInvoker(StaticDirectory) ->
FailoverClusterInvoker(RegistryDirectory, will execute route) -> Invoker
                    invoker = CLUSTER.join(new StaticDirectory(u, invokers));
                } else { // not a registry url, must be direct invoke.
                    invoker = CLUSTER.join(new StaticDirectory(invokers));
                }
            }
        }
        if (shouldCheck() && !invoker.isAvailable()) {
            throw new IllegalStateException("Failed to check the status of the
service " + interfaceName + ". No provider available for the service " + (group
== null ? "" : group + "/") + interfaceName + (version == null ? "" : ":" +
version) + " from the url " + invoker.getUrl() + " to the consumer " +
NetUtils.getLocalHost() + " use dubbo version " + Version.getVersion());
        }
        if (logger.isInfoEnabled()) {
            logger.info("Refer dubbo service " + interfaceClass.getName() + "
from url " + invoker.getUrl());
        }
        /**
         * @since 2.7.0
         * ServiceData Store
         */
        MetadataReportService metadataReportService = null;
        if ((metadataReportService = getMetadataReportService()) != null) {
            URL consumerURL = new URL(CONSUMER_PROTOCOL,
map.remove(REGISTER_IP_KEY), 0, map.get(INTERFACE_KEY), map);
            metadataReportService.publishConsumer(consumerURL);
        }
        // create service proxy 创建服务代理对象
        return (T) PROXY_FACTORY.getProxy(invoker);
```

现在我们重点来看创建Invoker实例的过程，Invoker 是 Dubbo 的核心模型，代表一个可执行体。在服务提供方，Invoker 用于调用服务提供类。在服务消费方，Invoker 用于执行远程调用，Invoker 是由 Protocol 实现类DubboProtocol调用refer方法

```
        // create rpc invoker.
        DubboInvoker<T> invoker = new DubboInvoker<T>(serviceType, url,
getClients(url), invokers);
        invokers.add(invoker);

        return invoker;
```

Dubbo 使用 NettyClient 进行通信，getClients逻辑如下：

```
private ExchangeClient[] getClients(URL url) {
        // whether to share connection

        boolean useShareConnect = false;

        int connections = url.getParameter(CONNECTIONS_KEY, 0);
        List<ReferenceCountExchangeClient> shareClients = null;
        // if not configured, connection is shared, otherwise, one connection
for one service
        if (connections == 0) {
            useShareConnect = true;

            /**
             * The xml configuration should have a higher priority than
properties.
             */
            String shareConnectionsStr = url.getParameter(SHARE_CONNECTIONS_KEY,
(String) null);
            connections =
Integer.parseInt(StringUtils.isBlank(shareConnectionsStr) ?
ConfigUtils.getProperty(SHARE_CONNECTIONS_KEY,
                    DEFAULT_SHARE_CONNECTIONS) : shareConnectionsStr);
            shareClients = getSharedClient(url, connections);
        }

        ExchangeClient[] clients = new ExchangeClient[connections];
        for (int i = 0; i < clients.length; i++) {
            if (useShareConnect) {
                clients[i] = shareClients.get(i);

            } else {
                clients[i] = initClient(url);
            }
        }

        return clients;
    }
```

根据 connections 数量决定是获取共享客户端还是创建新的客户端实例，默认情况下，使用共享客户端实例。getSharedClient 方法中也会调用 initClient 方法。

ExchangeClient 实例。initClient 方法首先获取用户配置的客户端类型，默认为 netty。然后检测用户配置的客户端类型是否存在，不存在则抛出异常。最后根据 lazy 配置决定创建什么类型的客户端。这里的 LazyConnectExchangeClient 会在 request 方法被调用时通过 Exchangers 的 connect 方法创建 ExchangeClient 客户端

connect连接方法中，getExchanger 会通过 SPI 加载 HeaderExchangeClient 实例，Transporters 的 connect方法中调用getTransporter().connect(url, handler)，getTransporter 方法是自适应扩展类，默认加载NettyTransporter，调用该类的connect 方法，往下就是通过Netty API创建Netty客户端了。

接下来就是为服务接口生成代理对象，代理对象生成的入口方法为 ProxyFactory 的 getProxy方法

```
@Override
public <T> T getProxy(Invoker<T> invoker) throws RpcException {
    return getProxy(invoker, generic: false);
}
```

```
    public <T> T getProxy(Invoker<T> invoker, boolean generic) throws
RpcException {
        Class<?>[] interfaces = null;
        String config = invoker.getUrl().getParameter(INTERFACES);
        if (config != null && config.length() > 0) {
            String[] types = COMMA_SPLIT_PATTERN.split(config);
            if (types != null && types.length > 0) {
                interfaces = new Class<?>[types.length + 2];
                interfaces[0] = invoker.getInterface();
                interfaces[1] = EchoService.class;
                for (int i = 0; i < types.length; i++) {
                    // TODO can we load successfully for a different
classloader?.
                    interfaces[i + 2] = ReflectUtils.forName(types[i]);
                }
            }
        }
        if (interfaces == null) {
            interfaces = new Class<?>[]{invoker.getInterface(),
EchoService.class};
        }

        if (!GenericService.class.isAssignableFrom(invoker.getInterface()) &&
generic) {
            int len = interfaces.length;
            Class<?>[] temp = interfaces;
            interfaces = new Class<?>[len + 1];
            System.arraycopy(temp, 0, interfaces, 0, len);
            interfaces[len] =
com.alibaba.dubbo.rpc.service.GenericService.class;
        }

        return getProxy(invoker, interfaces);
    }
```

最后getProxy(Invoker, Class<?>[]) 这个方法是一个抽象方法，实现类 JavassistProxyFactory 对该方法的实现

```
    // 生成 Proxy 子类（Proxy 是抽象类）。并调用 Proxy 子类的 newInstance 方法创建
Proxy 实例
    return (T) Proxy.getProxy(interfaces).newInstance(new
InvokerInvocationHandler(invoker));
}
```

以上代码通过 Proxy 的 getProxy 方法获取 Proxy 子类，然后创建 InvokerInvocationHandler 对象，并将该对象传给 newInstance 生成 Proxy 实例。InvokerInvocationHandler 实现自 JDK 的 InvocationHandler 接口。