前面一篇dubbo源码分析中，我们对ClusterInvoker和LoadBalance进行了分析，可以知道ClusterInvoker在一批Invoker中选择了一个Invoker来进行调用，而这里的Invoker是通过RegistryDirectory得到的，而RegistryDirectory返回的Invoker实现为：InvokerDelegete（RegistryDirectory的内部类），改类维护了对应的provider url， 同时也包含了一个ListenerInvokerWrapper，InvokerDelegete执行invoke方法时，调用的就是ListenerInvokerWrapper的invoke方法（参考dubbo源码分析-consumer端3-Invoker创建流程）。ListenerInvokerWrapper依然不是真正的调用者，它主要是监听了invoker的创建与销毁事件，它维护的invoker为经过ProtocolFilterWrapper转换过的Invoker，该Invoker在执行前需要先经过filter链的处理，转换代码：

[java] view plain copy

```
1. // ProtocolFilterWrapper中
2. private static <T> Invoker<T> buildInvokerChain(final
Invoker<T> invoker, String key, String group) {
3.     // 这里的invoker跟进使用的protocol不同而不同，如默认使用dubbo
protocol时，此处的invoker即为DubboInvoker
4.     Invoker<T> last = invoker;
5.     // 此处group为"consumer"，因此加载的是consumer对应的Filter
6.     List<Filter> filters =
ExtensionLoader.getExtensionLoader(Filter.class).getActivateExten
sion(invoker.getUrl(), key, group);
7.     if (filters.size() > 0) {
8.         // 如果有Filter则先调用filter.invoke(next, invocation);所
有filter调用完且向后传递才会调用真正的invoker
9.         for (int i = filters.size() - 1; i >= 0; i --) {
10.            final Filter filter = filters.get(i);
11.            final Invoker<T> next = last;
12.            last = new Invoker<T>() {
13.
14.                public Class<T> getInterface() {
15.                    return invoker.getInterface();
16.                }
17.
18.                public URL getUrl() {
19.                    return invoker.getUrl();
```

```
20.                 }
21.
22.                 public boolean isAvailable() {
23.                     return invoker.isAvailable();
24.                 }
25.
26.                 public Result invoke(Invocation invocation)
throws RpcException {
27.                     return filter.invoke(next, invocation);
28.                 }
29.
30.                 public void destroy() {
31.                     invoker.destroy();
32.                 }
33.
34.                 @Override
35.                 public String toString() {
36.                     return invoker.toString();
37.                 }
38.             };
39.         }
40.     }
41.     return last;
42. }
```

我们先来看看Filter是如何加载的，首先通过ExtensionLoader加载所有Filter类型的SPI，并从中过滤出Activate注解中group含consumer的Filter，同时，如果Activate注解中配置了value，则需要对应的url中也配置了相应的值，否则将会被排除掉。

[java] view plain copy

```
1. @Activate(group = Constants.CONSUMER, value =
Constants.GENERIC_KEY, order = 20000)
2. public class GenericImplFilter implements Filter {
3. ...
4. }
```

过滤出合适的Filter以后，还需要对Filter进行排序，使其能够按照正确的顺序执行，其排序的比较器(ActivateComparator)根据Activate注解信息进行处理：

[java] view plain copy

```
1. public @interface Activate {
2.     /**
3.      * Group过滤条件。
4.      * 包含{@link ExtensionLoader#getActivateExtension}的group参数给的值，则返回扩展。
5.      * 如没有Group设置，则不过滤。
6.      */
7.     String[] group() default {};
8. 
9.     /**
10.      * Key过滤条件。包含{@link ExtensionLoader#getActivateExtension}的URL的参数Key中有，则返回扩展。
11.      * 示例：
12.      * 注解的值 @Activate("cache,validatioin")
13.      * 则ExtensionLoader#getActivateExtension的URL的参数有cache或是validatioin则返回扩展。
14.      * <br/>
15.      * 如没有设置，则不过滤。
16.      */
17.     String[] value() default {};
18. 
19.     /**
20.      * 排序信息，可以不提供。表示注解所在的类排在哪些类之前
21.      */
22.     String[] before() default {};
23. 
24.     /**
25.      * 排序信息，可以不提供。表示注解所在的类排在哪些类之后
26.      */
27.     String[] after() default {};
28. 
29.     /**
30.      * 排序信息，可以不提供。order越小，排名越靠前，注意优先级before > after > order
31.      */
32.     int order() default 0;
33. }
```

对于consumer来说，主要的Filter包括：ConsumerContextFilter，MonitorFilter，FutureFilter，ActiveLimitFilter、GenericImplFilter、ValidationFilter。

ConsumerContextFilter：设置consumer调用的上下文，如本地地址，要调用的provider的地址，invoker信息，invocation信息等。RpcContext通过ThreadLocal实现，因此你可以在业务代码中直接通过RpcContext获取上下文信息(在调用对应方法之后才能获取)。需要注意的是RpcContext中的attachments中的内容在后面的远程调用中被传到provider，不建议业务使用，可以考虑traceId之类的数据传递，由于每次调用完成后都会进行清理，因此需要传递的数据每次调用都要重新设置。

[java] view plain copy

```java
@Activate(group = Constants.CONSUMER, order = -10000)
public class ConsumerContextFilter implements Filter {

    public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
        RpcContext.getContext()
                .setInvoker(invoker)
                .setInvocation(invocation)
                .setLocalAddress(NetUtils.getLocalHost(), 0)
// 设置本地的地址
                .setRemoteAddress(invoker.getUrl().getHost(),
// 设置本次调用的provider端的host和端口
                invoker.getUrl().getPort());
        if (invocation instanceof RpcInvocation) {
            ((RpcInvocation)invocation).setInvoker(invoker);
        }
        try {
            return invoker.invoke(invocation);
        } finally {
            // 调用完成后清理attachments中的数据
            RpcContext.getContext().clearAttachments();
        }
    }

}
```

MonitorFilter： 收集consumer或provider的每一次调用信息，将信息保存在内存中进行合并，并定时的讲信息上报到monitor服务。注意monitor需要用户主动的配置才会生效。收集的信息包括：调用耗时、调用

结果（成功/失败）、并发调用数

[java] view plain copy

```
<dubbo:monitor protocol="registry">
        <!-- 2分钟上报一次 -->
        <dubbo:parameter key="interval" value="120000" />
    </dubbo:monitor>
```

FutureFilter: 执行事件通知逻辑，包括调用前（oninvoke）、同步调用后/异步调用完成后（onreturn/onthrow）,他们配置的方式一样：beanId.methodName，以下是官方的demo:

[java] view plain copy

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

ActiveLimitFilter: 用于控制每个consumer调用指定方法的最大并发数，当配置了actives时生效。

[java] view plain copy

```
// value表示只有url中含Constants.ACTIVES_KEY时才会激活此Filter
@Activate(group = Constants.CONSUMER, value =
Constants.ACTIVES_KEY)
public class ActiveLimitFilter implements Filter {

    public Result invoke(Invoker<?> invoker, Invocation
invocation) throws RpcException {
        URL url = invoker.getUrl();
        String methodName = invocation.getMethodName();
        // 获取最大并发数设置，为0时不控制并发数
        int max =
invoker.getUrl().getMethodParameter(methodName,
```

```
Constants.ACTIVES_KEY, 0);
10.         // 获取当前并发数
11.         RpcStatus count =
RpcStatus.getStatus(invoker.getUrl(),
invocation.getMethodName());
12.         if (max > 0) {
13.             // 获取方法的调用超时时间，当并发数已经到阈值时，等待
timeout的时间，如果超时未获取到调用机会则直接报错
14.             long timeout =
invoker.getUrl().getMethodParameter(invocation.getMethodName(),
Constants.TIMEOUT_KEY, 0);
15.             long start = System.currentTimeMillis();
16.             long remain = timeout;
17.             int active = count.getActive();
18.             if (active >= max) {
19.                 // 对count调用wait方法，等待下面count.notify()的
唤醒，
20.                 // 当本次调用因为达到阈值而wait后，会等待之前的调用执
行完成并通过notify唤醒，每次只唤醒一个，
21.                 // 被唤醒后查看本次调用是否超时，如果未超时则判断当前
并发数是否低于阈值，如果低则再次wait。
22.                 // 注意，由于外层没有其他控制手段，当active = max -
1时，如果同时有多个调用运行到这里，并发会超过max，
23.                 // 因此并发限制并不是很精确
24.                 synchronized (count) {
25.                     while ((active = count.getActive()) >=
max) {
26.                         try {
27.                             count.wait(remain);
28.                         } catch (InterruptedException e) {
29.                         }
30.                         long elapsed =
System.currentTimeMillis() - start;
31.                         remain = timeout - elapsed;
32.                         if (remain <= 0) {
33.                             throw new RpcException("Waiting
concurrent invoke timeout in client-side for service:  "
34.                                     +
invoker.getInterface().getName() + ", method: "
35.                                     +
invocation.getMethodName() + ", elapsed: " + elapsed
36.                                     + ",
timeout: " + timeout + ". concurrent invokes: " + active
37.                                     + ". max
concurrent invoke limit: " + max);
```

```
                        }
                    }
                }
            }
        try {
            long begin = System.currentTimeMillis();
            // 增加一个并发数，调用完成或发生RuntimeException时减少一个并发数
            RpcStatus.beginCount(url, methodName);
            try {
                Result result = invoker.invoke(invocation);
                RpcStatus.endCount(url, methodName, System.currentTimeMillis() - begin, true);
                return result;
            } catch (RuntimeException t) {
                RpcStatus.endCount(url, methodName, System.currentTimeMillis() - begin, false);
                throw t;
            }
        } finally {
            if(max>0){
                // 调用完成以后唤起一个wait中的线程
                synchronized (count) {
                    count.notify();
                }
            }
        }
    }

}
```

注意上面的代码，为了RpcStatus.endCount记录成功失败的状态，endCount并没有在finally中执行，也就是在某些特殊的异常下可能会导致并发计数一直上升而不下降，到达阈值后就无法再进行对应的调用了，虽然这种情况很少，但还是觉得这个写法并不是很好。如果你需要扩展的Filter在ActiveLimitFilter之后执行，一定不要抛出Error或者Throwable级的异常！！

GenericImplFilter: 对泛化调用的支持，当consumer没有的api接口时使用，具体用法参考 官方文档； 注意返回调用无法调用回声测试，这个不知道是dubbo的一个bug还是故意这样设置的，如果需要用泛化调用的方式进行回声测试，可以联系我；

ValidationFilter: 用于对调用参数的验证，通过注解设置规则，可参考**hibernate**-validator。

consumer端常用的Filter就讲到这里，如果要自行扩展Filter，请注意@Activate中group的设置及优先级的设置。