

SCA Service Component Architecture

SCA服务构件架构

JAVA通用注解和API规范

满江红 redsaga.com



SCA v1.00, 2007.3.21

SCA Service Component Architecture

Java Common Annotations and APIs

SCA Version 1.00. March 21 2007

Technical Contacts:

Ron Barack	SAP AG
Michael Beisiegel	IBM Corporation
Henning Blohm	SAP AG
Dave Booz	IBM Corporation
Jeremy Boynes	Independent
Ching-Yun Chao	IBM Corporation
Adrian Colyer	Interface21
Mike Edwards	IBM Corporation
Hal Hildebrand	Oracle
Sabin Ielceanu	TIBCO Software, Inc
Anish Karmarkar	Oracle
Daniel Kulp	IONA Technologies plc.
Ashok Malhotra	Oracle
Jim Marino	BEA Systems, Inc.
Michael Rowley	BEA Systems, Inc.
Ken Tam	BEA Systems, Inc
Scott Vorthmann	TIBCO Software, Inc
Lance Waterman	Sybase, Inc.

Copyright Notice

© Copyright BEA Systems, Inc., Cape Clear Software, International Business Machines Corp, Interface21, IONA Technologies, Oracle, Primeton Technologies, Progress Software, Red Hat, Rogue Wave Software, SAP AG., Siemens AG., Software AG., Sybase Inc., TIBCO Software Inc., 2005, 2007. All rights reserved.

License

The Service Component Architecture Specification is being provided by the copyright holders under the following license. By using and/or copying this work, you agree that you have read, understood and will comply with the following terms and conditions:

Permission to copy and display the Service Component Architecture Specification and/or portions thereof, without modification, in any medium without fee or royalty is hereby granted, provided that you include the following on ALL copies of the Service Component Architecture Specification, or portions thereof, that you make:

1. A link or URL to the Service Component Architecture Specification at this location:
 - <http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications>
2. The full text of this copyright notice as shown in the Service Component Architecture Specification.

BEA, Cape Clear, IBM, Interface21, IONA, Oracle, Primeton, Progress Software, Red Hat, Rogue Wave, SAP, Siemens, Software AG., Sun, Sybase, TIBCO (collectively, the "Authors") agree to grant you a royalty-free license, under reasonable, non-discriminatory terms and conditions to patents that they deem necessary to implement the Service Component Architecture Specification.

THE Service Component Architecture SPECIFICATION IS PROVIDED "AS IS," AND THE AUTHORS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, REGARDING THIS SPECIFICATION AND THE IMPLEMENTATION OF ITS CONTENTS, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT OR TITLE.

THE AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE Service Components Architecture SPECIFICATION.

The name and trademarks of the Authors may NOT be used in any manner, including advertising or publicity pertaining to the Service Component Architecture Specification or its contents without specific, written prior permission. Title to copyright in the Service Component Architecture Specification will at all times remain with the Authors.

No other rights are granted by implication, estoppel or otherwise.

Status of this Document

This specification may change before final release and you are cautioned against relying on the content of this specification. The authors are currently soliciting your contributions and suggestions. Licenses are available for the purposes of feedback and (optionally) for implementation.

IBM is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.

BEA is a registered trademark of BEA Systems, Inc.

Cape Clear is a registered trademark of Cape Clear Software

IONA and IONA Technologies are registered trademarks of IONA Technologies plc.

Oracle is a registered trademark of Oracle USA, Inc.

Progress is a registered trademark of Progress Software Corporation

Primeton is a registered trademark of Primeton Technologies, Ltd.

Red Hat is a registered trademark of Red Hat Inc.

Rogue Wave is a registered trademark of Quovadx, Inc

SAP is a registered trademark of SAP AG.

SIEMENS is a registered trademark of SIEMENS AG

Software AG is a registered trademark of Software AG

Sun and Sun Microsystems are registered trademarks of Sun Microsystems, Inc.

Sybase is a registered trademark of Sybase, Inc.

TIBCO is a registered trademark of TIBCO Software Inc.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

目录

Copyright Notice	3
License	3
目录.....	5
1. 通用注解、API、客户程序和实现模型.....	7
1.1. 简介.....	7
1.2. 实现的元数据.....	7
1.2.1. 服务元数据.....	8
1.2.2. @Reference.....	8
1.2.3. @Property.....	9
1.2.4. 实现作用域: @Scope、@Init、@Destroy.....	9
1.3. 接口元数据.....	10
1.3.1. @Remotable.....	10
1.3.2. @Conversational.....	11
1.4. 客户 API.....	11
1.4.1. SCA 构件访问服务.....	11
1.4.2. 非 SCA 构件的实现访问服务.....	11
1.5. 错误处理.....	12
1.6. 异步与会话编程.....	12
1.6.1. @OneWay.....	12
1.6.2. 会话型服务.....	13
1.6.3. 将会话型服务作为参数传递.....	13
1.6.4. 会话型客户程序.....	14
1.6.5. 会话生命周期总结.....	15
1.6.6. Conversations ID.....	15
1.6.7. 回调.....	16
1.6.8. 会话与回调的绑定.....	21
1.7. Java API.....	21
1.7.1. Component Context.....	21
1.7.2. Request Context.....	23
1.7.3. CallableReference.....	24
1.7.4. ServiceReference.....	25
1.7.5 Conversation 会话.....	26
1.7.6 NoRegisteredCallbackException 异常.....	26
1.7.7. Service Runtime Exception.....	26
1.7.8. Service Unavailable Exception.....	27
1.7.9. Conversation Ended Exception.....	27
1.8 Java 注解.....	27
1.8.1. @AllowsPassByReference.....	27
1.8.2. @Callback.....	28
1.8.3. @ComponentName.....	30
1.8.4. @Conversation.....	30

1.8.5. @Constructor.....	31
1.8.6. @Context.....	31
1.8.7. @Conversational	32
1.8.8. @Destroy.....	33
1.8.9. @EagerInit.....	33
1.8.10.@EndsConversation	34
1.8.11.@Init.....	34
1.8.12.@OneWay.....	35
1.8.13.@Property.....	35
1.8.14.@Reference	37
1.8.15.@Remotable	39
1.8.16.@Scope.....	40
1.8.17.@Service	41
1.8.18.@ConversationAttributes	42
1.8.19.@ConversationID.....	43
1.9. WSDL to Java 和 Java to WSDL	43
2. Java 策略注解.....	44
2.1. 通用意图注解.....	44
2.2.特定意图注解.....	46
2.2.1. 如何生成特定意图注解	47
2.3. 意图注解的应用.....	48
2.3.1.继承和注解.....	49
2.4.声明性意图和注解性意图的关系	51
2.5. 策略集注解.....	51
2.6. 安全策略注解.....	52
2.6.1. 安全交互策略.....	52
2.6.2. 安全实现策略.....	54
3. 附录.....	57
3.1 参考文献.....	57

1. 通用注解、API、客户程序和实现模型

1.1. 简介

SCA通用注解、API、客户程序及实现模型为《SCA装配模型规范》[1]中定义的编程理念制定了Java语法规则。它规定了一套“java-based” SCA规范使用的API与注解。

具体的此规范包括如下内容：

- 1、构件服务、引用以及属性的实现元数据
- 2、客户端和构件API J3. 异步或会话服务的元数据。
- 3、回调机制的元数据
- 4、标准构件实现的作用域定义
- 5、Java与WSDL间的映射
- 6、安全策略注解

注意：个别的编程模型视需要可选择用本地的API和本地方言来实现它们自身对装配模型理念的映射。

在此规范中指定注解、API、客户程序以及实现模型，目标是提升一致性，减少各种Java相关的构件实现类型规范之间的重复。本规范中定义的注解、API、客户程序以及实现模型，其设计目的是为了以部分或全部的方式被Java相关的其他SCA规范所使用。

本文档使用J2SE 5的注解技术来实现元数据。当然，SCA也允许用J2SE1.4开发服务的客户程序与实现。所有由注解描述的元数据也可以使用独立的component type文件来表述，component type文件在《SCA装配规范》中定义。

1.2. 实现的元数据

这一节描述基于Java技术的SCA元数据是如何适用于基于Java的实现类型的。

1.2.1. 服务元数据

1.2.1.1. @Service

在Java类上使用 **@Service** 注解以指定实现的服务接口。通常，可以使用以下的任一方式定义服务接口：

- Java接口
- Java类
- 从WSDL[4]（Web Service描述语言）portType产生的Java接口（总是远程接口）

1.2.1.2. 远程服务的 Java 语义

在定义了服务的Java接口上使用 **@Remotable** 注解定义远程服务。远程服务应用于粗粒度服务，且以传值的方式传递参数。

1.2.1.3. 本地服务的 Java 语义

本地服务只能被署在相同地址空间的客户程序访问。

本地接口由不带 **@Remotable** 注解的Java接口或由Java类定义。

如下的代码片段显示了本地服务的Java接口。

```
package services.hello;

public interface HelloService {

    String hello(String message);
}
```

通常，本地接口是细粒度的，用于紧耦合的交互。

本地接口调用的数据交换语义是传引用 (**by-reference**)。这意味着编写代码时必须注意，由客户程序和服务提供者其中一方引起的参数（简单类型除外）改变，对另一方是可见的。

1.2.2. @Reference

通过定义类型为服务接口的属性、**setter**方法参数或构造函数参数，且用 **@Reference** 注解标注上述属性域或方法，以引用注入的方式访问服务。

1.2.3. @Property

实现能通过属性进行配置，属性在《SCA装配规范》[1]中定义。`@Property`注解用于定义SCA属性。

1.2.4. 实现作用域：@Scope、@Init、@Destroy

构件实现要么管理它们自身的状态，要么允许 SCA 运行时来做这些工作。后者情况下，SCA 定义实现作用域，此作用域指定实现与 SCA 运行时的可见性和生命周期合约。对构件所提供服务的调用由 SCA 运行时根据其实现作用域的语义被分发到一个实现的实例上去。

在实现类上使用 `@Scope` 注解指定作用域。

此文档定义了四种基本的作用域：

- STATELESS
- REQUEST
- CONVERSTATION
- COMPOSITE

基于 Java 的实现类型可以选择支持这些作用域中的任意一个，且可以定义特定于该实现类型的新的作用域。

实现类型也允许构件的实现声明生命周期方法，这些方法在实现被实例化或作用域过期时被调用。`@Init` 指示在作用域的生命周期中第一次使用实例时被调用的函数（标注为渴望初始化的 `composite` 作用域的实现除外，参见第 1.2.4.3 节）。`@Destroy` 指定作用域结束时被调用的函数。注意，只有公共的无参数的方法才能注解为生命周期函数。

如下片段展示注解了生命周期函数的服务实现的代码片段。

```
@Init
public void start() {
    ...
}

@Destroy
public void stop() {
    ...
}
```

如下章节给出基于 Java 的实现类型可能支持的四个标准的作用域。

1.2.4.1. 无状态作用域

对于无状态构件，服务请求之间没有隐式的关联。

1.2.4.2. 请求作用域

请求作用域的生命周期是从远程接口的请求进入SCA runtime开始直到一个线程同步处理完成这个请求为止。在生命周期过程中，所有的服务请求将被委托给同一个请求域构件的服务实现实例。

在很多情况下，本地的请求作用域服务被调用时，调用栈中没有此前的远程服务。例如，本地服务被非SCA实体调用的时候。这些情况都会被认为是远程请求，但请求的生命期限依赖于实现。比如，可将一个计时器事件将看作是一个远程请求。

1.2.4.3. 复合构件作用域

在容器复合构件的生命周期中，容器中所有的服务请求都被分发到同一实现实例。容器复合构件的生命周期定义为在运行时中由激活状态变为钝化状态的时间，钝化可以是正常的也可以是异常的。

复合构件作用域的实现也许还会用@EagerInit注解指定急切初始化。当标注了渴望初始化的话，在容器复合构件启动时就会创建复合构件作用域的实例。假如一个方法用@Init注解标注了，实例创建时就会调用此方法。

1.2.4.4. 会话作用域

会话定义为客户与目标服务之间的一系列相关的交互。当第一个服务请求被分发到一个提供了会话服务的实现实例时，会话作用域就开始了。当服务合约定义的一个终止操作被调用，且完成处理或会话过期之后，会话作用域就结束了。会话可以是长时间运行，且SCA运行时也许会选择钝化实现的实例。一旦钝化发生，运行时必须保证实现实例的状态被保存了。

注意，当会话型服务是由一个标注了会话范围的Java类实现时，SCA运行时将会透明地处理实现的状态。当然，也可能由实现自身管理其自己的状态。比如，一个无状态（或其他）作用域的Java类能实现会话型服务。

1.3 接口元数据

这一节为Java接口描述SCA元数据。

1.3.1. @Remotable

用于接口上的@Remotable注解表示该接口是被设计来用于远程通讯的。远程接口就意味着用于粗粒度的服务。操作参数和返回值都是传值的方式传递。

1.3.2. @Conversational

Java 服务接口可使用@Conversational 注解（描述在 SCA 组装规范[1]）来指定其合约是否为会话的。会话服务表示服务的多个请求之间存在某种关联。

若服务接口没有指定@Conversational 注解，该服务的合约就是无状态的。

1.4. 客户 API

这一节讲述 SCA 服务是如何被构件和非受管代码（比如不是作为 SCA 构件来运行的代码）通过编程方式来访问的。

这一节讲述构件与非受管代码如何通过编程方式进行 SCA 服务访问的。

1.4.1. SCA 构件访问服务

SCA构件可通过注入或使用component Context API编码方式获得一个服务的引用。推荐使用引用注入方式访问服务，因为这样产生的代码能在最大程度上减少使用中间件API的使用。只有在不可能引用注入的情况下才应该使用component Context API。

1.4.1.1. 使用 component context API

当构件的实现需要访问一个服务，而此服务的引用在编译期无法知晓时，可以使用构件的ComponentContext来定位该引用。

1.4.2. 非 SCA 构件的实现访问服务

本节描述作为非SCA构件（此构件是SCA复合构件的组成部分）运行的Java代码如何通过引用来访问SCA服务。

1.4.2.1. ComponentContext

非SCA客户代码能通过使用ComponentContext API执行一个SCA域中某个构件的操作。客户代码如何获取一个ComponentContext的引用与运行时相关。以下示例代码给出了非SCA代码使用

ComponentContext API的用法。

```
ComponentContext context = // obtained through host environment-specific means
HelloService helloService = context.getService(HelloService.class, "HelloService");
String result = helloService.hello("Hello World!");
```

1.5. 错误处理

客户调用服务的方法也许会引发业务异常和SCA运行时异常。

被调用的服务方法的实现抛出业务异常，且该业务异常被定义为服务接口上的checked exception。

SCA运行时异常由SCA运行时产生，以指示在管理构件执行和与远程服务交互的过程出现的问题。会用到ServiceRuntimeException和服务UnavailableException SCA运行时异常，它们在1.5节中定义。

1.6. 异步与会话编程

服务的异步编程用于客户调用一个服务，但不需要等待服务执行而自己继续执行的应用场景。通常，被调用的服务会在后续的某个时刻执行。被调用服务的输出，如果有的话，必须通过另外的机制返回给客户程序，因为在服务调用的那个时刻输出还没有产生。与之相对的是调用-返回式的同步合约，被调用的服务执行、并将输出返回到客户程序，客户程序才会继续执行。SCA异步编程模型由非堵塞型方法调用的支持、会话型服务以及回调组成。这些话题都会在后面的章节中讨论。

会话型服务指在客户和服务提供者之间存在持续进行的一系列交互。与简单的无状态交互相比，会话型服务需要维护某些状态数据。尽管不是强制性的，但异步服务经常会伴随着会话的使用。

1.6.1. @OneWay

非堵塞调用是异步编程中最简单的形式。客户程序调用服务之后不等待服务执行，立即继续执行下去。

任何返回 void 类型且无声明异常的方法都可用@OneWay 注解来标注。意味着此方法是非堵塞的，与服务提供者通讯时可用绑定机制缓冲请求，并在后续的某个时刻发送。

对于有返回值的或声明了异常的方法，SCA 尚未定义非堵塞性方法调用的机制。建议服务的设计者尽可能地

定义 one-way 方法，以便为部署者提供最大的绑定灵活度。

1.6.2. 会话型服务

一个服务可以通过@Conversational标注其Java接口来声明为会话型服务。如果一个服务接口未标注@Conversational，则表示为无状态的。

1.6.2.1. ConversationAttributes

基于Java的实现类可用@ConversationAttributes来修饰，此注解用于为会话型实现的实例指定过期规则。

以下是使用@ConversationAttributes的例子：

```
package com.bigbank;
import org.osoa.sca.annotations.Conversation;
import org.osoa.sca.annotations.ConversationID;

@ConversationAttributes(maxAge="30 days");
public class LoanServiceImpl implements LoanService {
}

```

1.6.2.2. @EndsConversation

可以用@EndsConversation注解标注会话接口中的方法。标注了@EndsConversation注解的方法一旦被调用，客户和服务提供者的会话就会终止。也就意味着在同一个会话中不能再调用该服务的其他方法了。这会让客户和服务提供者都释放会话相关的资源。

回调接口（见后文）中的方法也可以标注@EndsConversation注解，以便服务提供者也能选择终止会话。

如果会话已经终止后调用了此会话接口上的一个方法，会抛出ConversationEndedException异常（继承自ServiceRuntimeException），如果在客户和服务提供者间调用各自的@EndsConversation方法存在竞争条件（race condition）也会发生异常。

1.6.3. 将会话型服务作为参数传递

描述一个单一会话的服务引用可以作为参数传递给其它的服务，即便那些其它服务是远程的。为了让一个构件继续由其它构件启动的会话，就会用到它。

服务提供者也可以为其自身创建一个服务引用，并将自己传递给其它服务。服务的实现使用以下方法调用来完成此项工作

```
interface ComponentContext {
    ...
}

```

```

    <B> ServiceReference<B> createSelfReference (Class businessInterface);
    <B> ServiceReference<B> createSelfReference (Class businessInterface,
                                                String serviceName);
}

```

如果构件实现了多个服务，必须使用带有附加serviceName参数的第二种形式。

这种方式可用于支持复杂的回调模式，比如当一个回调仅仅适用于一个的更大会话的子集时。简单回调模式可以通过后面描述的内置回调支持来处理。

1.6.4. 会话型客户程序

会话型服务的客户程序不需要遵守特定的编码规则。客户程序对接口的会话特性的利用，是通过接口中不同方法之间的关系和方法间的数据共享来体现的。如果服务是异步的，客户程序可以通过conversationID等方式跟踪与会话相关的状态数据。

客户程序的开发者通过内省服务合约得知服务是会话型的。

以下代码展示了客户程序如何访问以上所描述的会话型服务：

```

@Reference
LoanService loanService;
// Known to be conversational because the interface is marked as
// conversational

    public void applyForMortgage(Customer customer, HouseInfo houseInfo,
                                int
                                term)
    {
        LoanApplication loanApp;
        loanApp = createApplication(customer, houseInfo);
        loanService.apply(loanApp);
        loanService.lockCurrentRate(term);
    }

    public boolean isApproved() {
        return loanService.getLoanStatus().equals("approved");
    }

    public LoanApplication createApplication(Customer customer,
                                            HouseInfo houseInfo) {
        return ...;
    }

```

1.6.5. 会话生命周期总结

启动会话

发生以下情况之一，会话即在客户端启动：

- 注入了会话型服务的@Reference
- 调用了CompositeContext.getServiceReference

然后可以调用服务的方法，进行会话。

继续会话

客户程序通过以下方式继续某个已存在的会话：

- 持有会话启动时创建的服务引用
- 从其它服务获取作为参数传递的服务引用对象，甚至是远程的服务
- 从某种持久化存储形式装载一个服务引用

结束会话

当发生以下情况时会话结束，任何与会话相关的状态都会被释放：

- 调用标注了@EndsConversation注解的服务操作
- 服务端调用了@Callback引用上的一个标注了@EndsConversation的方法
- 服务端会话生命时限超时
- 客户程序调用Conversation.end()
- 会话中的操作抛出了任何非业务异常

如果在已经调用了@EndsConversation方法后，调用服务引用上的一个方法，那么将自动启动一个新的会话。如果在此@EndsConversation方法调用后，而下一个会话启动前，调用ServiceReference.getConversationID()，将返回null。

如果服务提供者会话超时已经使会话结束，继续使用服务引用会抛出ConversationEndedException异常。为了使用该服务引用建立新会话，必须先调用endConversation ()方法。

1.6.6. Conversations ID

如果在一个protected或public的属性或setter方法上标注了@ConversationID，那么该会话的会话ID就会被注入到属性上。属性的类型不一定是String类型。系统产生的会话ID总是字符串，但应用产生的会话ID可以是其它的复杂数据类型。

1.6.6.1. 应用指定 Conversation ID

当使用由客户提供的会话ID，同样可以利用会话型服务的状态管理功能。但此时客户程序不能使用引用注入，而应该使用ServiceReference.setConversationID() API。

传入该函数（译者注：此处指ServiceReference.setConversationID()函数）的conversation ID应该是一个String或是能序列化为XML的对象实例。此ID对于客户构件整个生命周期都必须是唯一的。如果客户程序是非SCA构件，那么此ID必须是全局唯一的。

不是所有的会话型服务绑定都支持应用指定会话ID，或可能只支持String类型的应用指定会话ID。

1.6.6.2. 客户端访问会话 ID

不论conversation ID是由客户程序选择还是由系统产生的，客户程序都能通过调用serviceReference.getConversationID()来访问conversation ID。

如果conversation ID不是由应用指定的，那么方法ServiceReference.getConversationID()仅仅保证在第一个操作调用之后返回一个有效的值，否则返回null。

1.6.7. 回调

回调服务用于从一个服务提供者回到其客户端的异步通讯，此异步通讯与通过从同步操作返回值的通讯截然不同。回调用于双向服务，双向服务有两个接口：

- 服务的接口
- 回调接口，必须由客户程序提供

回调既能用于远程服务也能用于本地服务。双向服务的两个接口必须同时是远程的或同时是本地的。混用两者是非法的。回调有两种基本的形式：无状态的回调和有状态的回调。

回调接口通过在一个远程服务接口上使用@Callback注解来声明。注解的参数为回调接口的Class对象（译者注：如下面例子中的MyServiceCallback.class）。@Callback注解也能应用于实现的一个方法或属性上。在下节将介绍到，为了实现回调注入而使用@Callback注解。

1.6.7.1. 有状态回调

有状态回调是一个服务的客户构件的特定实现的实例。有状态回调的接口应该用@Conversational标注。

在以下例子中，口定义了一个有状态回调的交互。

```
package somepackage;
import org.osoa.sca.annotations.Callback;
import org.osoa.sca.annotations.Conversational;
import org.osoa.sca.annotations.Remotable;
```

```

    @Remotable
    @Conversational
    @Callback(MyServiceCallback.class)
    public interface MyService {

    public void someMethod(String arg);
}

@Remotable
    public interface MyServiceCallback {

        public void receiveResult(String result);
    }

```

例中的服务实现使用@Callback来标注需要注入一个有状态的回调。以下例子是实现上述服务的代码片段。在这个例子中，请求被转交给其它构件，因此例中服务起到了一个中介的作用。因为服务是会话作用域的 (conversation scoped)，所以当后端服务回送其异步响应时，回调仍然有效。

```

@Callback
protected MyServiceCallback callback;

@Reference
protected MyService backendService;

public void someMethod(String arg) {
    backendService.someMethod(arg);
}

public void receiveResult(String result) {
    callback.receiveResult(result);
}

```

这个代码片段必须来自提供了两个服务的实现。这两个服务，一个提供给它是客户程序 (MyService)，一个用于从后端接受回调 (MyServiceCallback) (译者注：其实就是实现两个接口，一个是服务接口、一个是回调接口)。该服务的客户程序也要实现MyServiceCallback中定义的方法。

```

private MyService myService;

@Reference
public void setMyService(MyService service){
    myService = service;
}

public void aClientMethod() {
    ...
    myService.someMethod(arg);
}

public void receiveResult(String result) {
    // code to process the result
}

```

通过服务引用作为参数传递的方式（或能力），有状态的回调也支持类似的某些场景用例。关键的不同在于有状态回调不需要向服务操作传递任何附加参数（译者注：一般情况下，需要为每个方法传递一个Callback实例作为参数）。这带来了很大的便利。否则如果服务有很多操作，且其中任何一个操作都可以作为会话的第一个操作，那么给每个可能启动会话的操作都加上回调参数是非常繁琐的。它也比专门设立一个显式操作来传递需要用到的回调对象，然后要求应用开发者去显式调用更自然。

1.6.7.2. 无状态回调

无状态回调接口是没有标注conversational的回调接口。不像有状态服务，使用无状态回调的客户不会有任何回调方法被路由到包含会话相关状态的实例上（译者注：拿Java反射类比，method.invoke(target,args)中的target是不包含任何状态的）。在此情况下，客户程序自身要承担完成持久化状态管理的职能。客户程序必须用到的信息就是回调ID对象。此回调ID对象随着服务的请求被传递，并确保被任何回调返回。

以下是上面客户代码片段的一个重复，但假定MyServiceCallback是无状态的。此情形下，客户程序需要在调用服务之前设置回调ID，当接收响应的时候获取回调ID。

```
private ServiceReference<MyService> myService;

@Reference
public void setMyService(ServiceReference<MyService> service){
    myService = service;
}

public void aClientMethod() {
    String someKey = "1234";
    ...

    myService.setCallbackID(someKey);
    myService.getService().someMethod(arg);
}

public void receiveResult(String result) {
    Object key = myService.getCallbackID();
    // Lookup any relevant state based on "key"
    // code to process the result
}
```

正如有状态回调一样，服务的实现通过用@Callback注解来标注属性域或setter方法来访问回调对象。如下所示：

```
@Callback
protected MyServiceCallback callback;
```

无状态服务的不同就在于，如果构件正服务于除原始客户程序以外的任何请求，回调域都是无效的。所以，前面章节使用的技术，那里存在来自backendService的响应，而backendService作为来自MyService的回

调被传递是无效的，因为当接收到后端系统的消息时回调域还是null。

1.6.7.3. 实现多个双向接口

既然单个实现类可能实现多个服务，也可能为每个实现的服务定义回调。服务的实现能为每个回调包含一个注入域。运行时会根据回调的类型注入回调到相应的属性域中。下面演示了两个属性域的定义，每个属性域都对应于由实现所提供的的一个特定的服务。

```
@Callback
protected MyService1Callback callback1;

@Callback
protected MyService2Callback callback2;
```

如果单个回调有一个类型与多个声明的回调域相兼容，那么所有的回调域都会被设置。

1.6.7.4. 访问回调

除了注入回调服务的引用外，也可以通过@Callback注解一个属性域或函数的方式来获得一个回调实例的引用。

一个实现了回调服务接口的引用可以通过使用CallableReference.getService()来获取。

以下代码片段来自使用回调API的服务实现：

```
@Callback;
protected CallableReference<MyCallback> callback;

public void someMethod() {

    MyCallback myCallback = callback.getCallback();
    ...
    callback.receiveResult(theResult);
}
```

作为备选方案，回调可以通过使用RequestContext API来编码获取。以下片段说明了如何用编程的方式来获取方法中的回调：

```
public void someMethod() {

    MyCallback myCallback = ComponentContext.getRequestContext().getCallback();
    ...
    callback.receiveResult(theResult);
}
```

在客户端，实现了回调的服务可以访问随回调操作返回的回调ID（即引用参数），该回调ID也可通过访问request context取得。如下所示：

```
@Context;
```

```

protected RequestContext requestContext;

void receiveResult(Object theResult) {

    Object refParams = requestContext.getServiceReference().getCallbackID();
    ...
}

```

在客户端，由getServiceReference()函数返回的对象，就是用于发送原始请求的服务引用。getCallbackID()方法返回的对象描述了与回调关联的标识，该标识可以是简单的String类型也可以是一个对象（会在“自定义回调标识”中讲到）

1.6.7.5. 自定义回调

默认情况下，服务的客户构件可以认为是双向服务的回调服务。然而，可以通过使用ServiceReference.setCallback()方法来改变回调。作为回调被传递的对象应该实现了回调定义接口，包括接口上任何附加的SCA语义，比如作用域或是否远程。

因为除客户程序之外，服务也可以用于实现回调，所以如果客户程序并没有实现它引用的回调接口，SCA也不会产生部署时错误。但是，如果调用了某个没有setCallback()方法的引用，那么客户端就会抛出NoRegisteredCallbackException异常。

对于有状态回调接口的回调对象有附加的要求，那就是必须是可序列化的。SCA运行时有可能会需要序列化回调对象并持久化存储之。

回调对象也可以是对其它服务的一个服务引用。该情况下，回调消息直接到达被设置为回调的那个服务。如果回调对象不是一个服务引用，那么回调消息会达到客户程序，然后被路由到注册为回调对象的特定实例。然而，如果回调接口是无状态作用域（stateless scope），那么回调对象必须是一个服务引用。

1.6.7.6. 自定义回调标识

用于标识回调请求的标识默认是由系统产生的。然而，它也可以通过调用ServiceReference.setCallbackID()方法来提供由应用指定的回调标识。它既能用于有状态的回调也可以用于无状态的回调。该标识将被发送给服务提供者，由绑定机制来确保服务提供者在回调方法被调用时把ID发送回来。

回调标识ID和会话ID有同样的限制。它必须是一个字符串或是可以被序列化为XML的对象。由绑定决定具体使用哪种机制去完成标识的传送。当然，特定的绑定还会有更进一步的限制。

1.6.8. 会话与回调的绑定

描述会话型服务的会话ID有很多方式，具体依赖于所用的绑定类型。比如，如果在某个Web service绑定机制中使用了可靠性传输就可能为会话ID选用WS-RM序列ID。如果是WS-Eventing则使用另一种技术（标识头部信息）。WS-Context OASIS TC正是为创建一种通用机制而设立的。

SCA编程模型支持会话，但它交由绑定（binding）去选择在连线（wire）上描述会话ID的方式。

1.7. Java API

本节为SCA的Java API提供参考。

1.7.1. Component Context

如下片段定义ComponentContext

```
package org.osoa.sca;

public interface ComponentContext {

    String getURI();

    <B> B getService(Class<B> businessInterface, String referenceName);

    <B> ServiceReference<B> getServiceReference(Class<B> businessInterface,
String referenceName);

    <B> ServiceReference<B> createSelfReference(Class<B> businessInterface);

    <B> ServiceReference<B> createSelfReference(Class<B> businessInterface,
String serviceName);

    <B> B getProperty(Class<B> type, String propertyName);
```

```

<B, R extends CallableReference<B>> R cast(B target)

throws    IllegalArgumentException;

RequestContext getRequestContext();

<B> ServiceReference<B> cast(B target) throws IllegalArgumentException;

}

```

- **getURI ()** – 返回SCA域中构件的绝对URI
- **getService (Class businessInterface, String referenceName)** – 返回当前构件定义的引用代理
- **getServiceReference (Class businessInterface, String referenceName)** – 返回当前构件定义的服务引用。
- **createSelfReference (Class businessInterface)** – 返回一个服务引用，用于调用此构件上设计的服务。
- **createSelfReference (Class businessInterface, String serviceName)** – 返回一个服务引用，用于调用此构件上设计的服务。*serviceName*显式声明要调用的服务名。
- **getProperty (Class type, String propertyName)** – 返回此构件定义的SCA属性值。
- **getRequestContext ()** – 返回当前SCA服务请求的上下文，如果没有当前请求或上下文无效则返回null。
- **cast (B target)** – 将一个类型安全的引用转换为一个可调用引用（CallableReference）。

构件可以通过定义一个类型为org.osoa.sca.ComponentContext且标注了@Context的protected或public的属性域或public的setter方法来访问其构件上下文（component context）。构件使用ComponentContext.getService(..)访问此目标服务。

如下片段定义了带有**getService()**方法的ComponentContext Java接口。

```

package org.osoa.sca;

public interface ComponentContext {
...
    T getService(Class<T> serviceType, String referenceName);
}

```

`getService()`方法输入参数用于描述客户目标服务的Java类型以及服务引用的名字。其返回一个对象，提供对服务的访问。此返回的对象实现服务的Java接口。

如下例子展示了在Java类中使用`@Context`注解的构件上下文定义。

```
private ComponentContext componentContext;

@Context
public void setContext(ComponentContext context){
    componentContext = context;
}

public void doSomething(){
    HelloWorld service =
        componentContext.getService(HelloWorld.class, "HelloWorldComponent");
    service.hello("hello");
}
```

类似地，非SCA客户代码可以使用`ComponentContext` API来执行SCA域中构件的操作。非SCA客户代码如何获取对`ComponentContext`的引用由运行时指定。

1.7.2. Request Context

如下片段展示了`RequestContext`的Java接口：

```

package org.osoa.sca;

import javax.security.auth.Subject;

public interface RequestContext {

    Subject getSecuritySubject();

    String getServiceName();

    <CB> CallbackReference<CB> getCallbackReference();
    <CB> CB getCallback(); <B> CallableReference<B> getServiceReference();
}

```

RequestContext Java接口有如下方法:

- **getSecuritySubject()** – 返回当前请求的JAAS Subject（译者注：与安全相关的Java 授权与认证主题）
- **getServiceName()** – 返回请求进入的Java实现上的服务名。
- **getCallbackReference()** – 返回调用者指定的对回调的可调用引用。
- **getCallback()** – 返回调用者指定的回调代理。
- **getServiceReference()** – 返回可调用引用，其描述请求上被调用的服务或回调引用。对于服务的实现在一个返回的服务引用上调用setCallback()是非法的。

1.7.3. CallableReference

如下片段定义了CallableReference:

```

package org.osoa.sca;

public interface CallableReference<B> {

    B getService();
    Class<B> getBusinessInterface();
    boolean isConversational();
    Conversation getConversation();
    Object getCallbackID();
}

```

CallableReference Java接口有如下方法:

- **getService()** - 返回一个指向此引用目标的类型安全引用。返回的实例被确保实现此引用的业务接口。返回值是一个对实现了引用相关业务接口目标的代理。

- **getBusinessInterface()** – 返回与引用相关的业务接口的Java类。
- **isConversational()** – 如果引用是会话型的返回true。
- **getConversation()** – 返回与引用相关的会话。如果当前无活性会话则返回null。
- **getCallbackID()** – 返回回调ID。

1.7.4. ServiceReference

可以在类型为ServiceReference的protected或public属性域或public setter方法上使用@Reference注解注入服务引用（ServiceReference）。这些方法使用的细节描述在此文档的异步编程中讲到。

如下片段定义了ServiceReference:

```
package org.osoa.sca;

public interface ServiceReference<B> extends CallableReference<B>{

    Object getConversationID();
    void setConversationID(Object conversationId) throws IllegalStateException;
    void setCallbackID(Object callbackID);
    Object getCallback();
    void setCallback(Object callback);
}
```

ServiceReference Java接口包含CallableReference的函数外加如下函数:

- **getConversationID()** - 返回由用户提供的ID，此用户是与会话相关的用户，同时会话通过该引用初始化。
- **setConversationID(Object conversationId)** – 设置id以便关联任何通过此引用启动的会话。如果提供的值为null，那么由实现产生。如果会话当前已经与此引用关联，则抛出IllegalStateException异常。
- **setCallbackID(Object callbackID)** – 设置回调ID。
- **getCallback()** – 返回回调对象。
- **setCallback(Object callback)** – 设置回调对象。

1.7.5 Conversation 会话

如下片段定义了Conversation:

```
package org.osoa.sca;

public interface Conversation {
    Object getConversionID();

    void end();
}
```

ServiceReference Java接口有如下方法:

- **getConversationID()** – 返回此会话的标识符。如果已经为此引用提供了一个用户定义的标识符，那么就返回该值；否则返回会话初始化时由系统产生的标识符。
- **end()** –结束会话。

1.7.6. NoRegisteredCallbackException 异常

如下片段展示了NoRegisteredCallbackException异常。

```
package org.osoa.sca;

public class NoRegisteredCallbackException extends ServiceRuntimeException {
    ...
}
```

1.7.7. Service Runtime Exception

如下片段展示了 ServiceRuntimeException 异常。

```
package org.osoa.sca;

public class ServiceRuntimeException extends RuntimeException {
    ...
}
```

此异常报告 SCA 构件执行管理中的问题。

1.7.8. Service Unavailable Exception

如下展示了 `ServiceRuntimeException` 异常。

```
package org.osoa.sca;

public class ServiceUnavailableException extends ServiceRuntimeException {
    ...
}
```

此异常报告与远程服务交互中的问题。其扩展自 `ServiceRuntimeException`。这些异常是瞬时的，因此可以进行重试。除 `ServiceUnavailableException` 以外的所有 `ServiceRuntimeException` 类型的异常都不可能通过重试操作来解决，因为大多数可能要求人工干预。

1.7.9. Conversation Ended Exception

如下片段展示了 `ConversationEndedException`。

```
package org.osoa.sca;

public class ConversationEndedException extends ServiceRuntimeException {
    ...
}
```

1.8 Java 注解

此节提供所有应用于 SCA 的 Java 注解的定义。

1.8.1. @AllowsPassByReference

如下片段展示了 `@AllowsPassByReference` 注解类型的定义。

```
package org.osoa.sca.annotations;
```

```

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target({TYPE, METHOD})
@Retention(RUNTIME)
public @interface AllowsPassByReference {

}

```

@AllowsPassByReference 注解用在远程接口的实现上，指示与同一地址空间中服务的交互允许使用传引用数据交换语义。实现保证即使参数和返回值实际上是传引用也会维护其传值语义。这意味着服务不会修改任何的操作参数和返回值，甚至在从操作中返回后。实现了远程服务的整个类或单独的远程服务方法实现上都可以使用 **@AllowsPassByReference** 注解标注。

@AllowsPassByReference 没有属性。

如下片段展示了一个样例，这里为Java构件实现类上的一个服务函数的实现定义了 **@AllowsPassByReference**。

```

@AllowsPassByReference
public String hello(String message) {
    ...
}

```

1.8.2. @Callback

如下代码片段展示了 **@Callback** 注解类型的定义：

```

package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE, METHOD, FIELD)

```

```
@Retention(RUNTIME)
public @interface Callback {

    Class<?> value() default Void.class;

}
```

@Callback注解类型用于注解一个带有回调接口的远程服务接口。此远程服务接口将一个回调接口的Java类对象作为参数。

@Callback注解有如下属性：

- **value** – 一个包含了回调接口的Java类文件的名字。

@Callback注解也能用于SCA实现类的一个方法或属性域上，从而完成对回调的注入。

以下代码片段展现了在一个接口上使用callback注解：

```
@Remotable
@Callback(MyServiceCallback.class)
public interface MyService {

    public void someAsyncMethod(String arg);

}
```

@Callback注解声明一个回调接口的使用如下：

```
package somepackage;
import org.osoa.sca.annotations.Callback;
import org.osoa.sca.annotations.Remotable;
@Remotable
@Callback(MyServiceCallback.class)
public interface MyService {

    public void someMethod(String arg);

}

@Remotable
public interface MyServiceCallback {

    public void receiveResult(String result);

}
```

在这个例子中，隐含的component type是：

```
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0" >

<service name="MyService">
    <interface.java interface="somepackage.MyService"
        callbackInterface="somepackage.MyServiceCallback"/>
</service>
</componentType>
```

1.8.3. @ComponentName

如下代码片段展示了@ComponentName 注解类型的定义。

```
package org.osea.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ComponentName {

}
```

@ComponentName 注解类型用于标注一个 Java 类的属性域或用于注入构件名的 setter 方法。如下代码片段展示了一个构件名属性域定义的使用案例。

```
@ComponentName
private String componentName;

@ComponentName
public void setComponentName(String name){
    //...
}
```

1.8.4. @Conversation

如下片段展示了@Conversation 注解类型的定义。

```
package org.osea.sca.annotations;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
```

```

@Target(TYPE)
@Retention(RUNTIME)
public @interface Conversation {
}

```

1.8.5. @Constructor

如下片断展示了@Constructor 注解类型定义。

```

package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.CONSTRUCTOR;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(CONSTRUCTOR)
@Retention(RUNTIME)
public @interface Constructor {
    String[] value() default "";
}

```

@Constructor 注解用于标注初始化 Java 构件实现时要使用的特定构造函数。

@Constructor 注解有如下属性：

- **value (可选)** –标识与每个构造函数参数相对应的 property/reference 名称。数组中的位置决定构造函数参数中的哪一个被命名。

1.8.6. @Context

如下片断展示了@Context 注解类型定义

```

package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;

```

```

import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface Context {
}

```

@Context 注解类型用于标注一个 Java 类的属性域或用于为构件注入 composite context 的 setter 方法。被注入的 context 的类型是由 java 类的属性域的类型或 setter 方法的输入参数决定，此类型可以是 ComponentContext 或 RequestContext。

@Context 注解没有属性。

如下片段展示了 ComponentContext 数据成员定义。

```

@Context
private ComponentContext context;

```

1.8.7. @Conversational

如下片段展示了 @Conversational 注解类型的定义：

```

package org.oesa.sca.annotations;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Conversational {
}

```

@Conversational 注解用于一个 Java 接口上表示一个会话型服务的合约。

@Conversational 注解没有属性。

1.8.8. @Destroy

如下片段展示@Destroy 注解类型的接口。

```

package org.osea.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(METHOD)
@Retention(RUNTIME)
public @interface Destroy {
}

```

@Destroy 注解类型用于标注一个 Java 类的方法。此方法在类实现的本地服务定义的作用域结束时被调用。方法必须有一个 void 返回值且无参数。被标注的方法必须是 public 的。

@Destroy 注解没有属性。

如下代码片段展示了 destroy 方法定义的使用案例。

```

@Destroy
void myDestroyMethod() {
    ...
}

```

1.8.9. @EagerInit

如下代码给出了@EagerInit 注解类型的定义。

```

package org.osea.sca.annotations;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)

```

```
public @interface EagerInit {

}
```

1.8.10. @EndsConversation

如下代码给出了@EndsConversation 的注解类型的定义。

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(METHOD)
@Retention(RUNTIME)
public @interface EndsConversation {
}
```

@EndsConversation 注解类型用于修饰 Java 接口上的一个方法。此方法被调用，会结束一个会话。
@EndsConversation 注解没有属性。

1.8.11. @Init

如下代码给出了@Init 的注解类型的定义。

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(METHOD)
@Retention(RUNTIME)
public @interface Init {
}
```

```
}
```

`@Init` 注解类型用于注释一个 Java 类方法。此方法在类实现的本地服务作用域开始时被调用。此方法必须有一个 `void` 返回值且无参数。被注释的方法必须是 `public` 的。被注释的方法在所有属性和引用注入都完成后被调用。

`@Init` 注解没有属性。

如下代码片段给出了 `init` 方法定义的使用案例。

```
@Init
void myInitMethod() {
    ...
}
```

1.8.12. @OneWay

以下代码给出了 `@OneWay` 注解类型的定义。

```
package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(METHOD)
@Retention(RUNTIME)
public @interface OneWay {
}
```

`@OneWay` 注解类型用于标注一个 Java 接口的方法，指示此调用将以非阻塞方式分发。非阻塞方式在异步编程节有所描述。

`@OneWay` 注解没有属性。

1.8.13. @Property

以下代码给出了 `@Property` 注解类型的定义。

```

package org.osoa.sca.annotations;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target({METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface Property {

    public String name() default "";
    public boolean required() default false;
}

```

@Property 注解类型用于注释一个 Java 类的属性域或用于注入 SCA 属性值的 setter 方法。被注入的属性类型，可以是简单 Java 类型也可以是复杂 Java 类型，由 Java 类的属性域或 setter 方法的输入参数的类型决定。

@Property 注解也能用在 protected 或 public 的属性域或 setter 方法或一个构造函数上。

即使不存在 @Property 注解，也能通过 public 的 setter 方法注入属性。然而，为了把属性注入到一个非 public 的属性域中，必须使用 @Property 注解。在没有 @Property 注解的情况下，属性名与属性域或 setter 的名字相同。

如果某个属性既有 setter 方法又有数据成员，使用 setter 方法。

@Property 注解有如下属性：

- **name (optional)** –属性名，默认为java类的属性域名
- **required (optional)** –指定是否必须注入，默认为false

以下代码片段展示了一个属性的数据成员定义的案例

```

@property(name="currency", required=true)
protected String currency;

```

以下代码片段展示了一个属性的 setter 定义的案例

```

@property(name="currency", required=true)
public void setCurrency( String theCurrency );

```

如果属性是定义为一个数组或 java.util.Collection，那么隐含的 component type 有一个附带 many 属性设置为 true 的 property。

以下片段展示了为集合类型使用 @Property 注解的配置 property 的定义。

```

...
private List<String> helloConfigurationProperty;

@property(required=true)
public void setHelloConfigurationProperty(List<String> property){
    helloConfigurationProperty = property;
}
...

```

1.8.14. @Reference

以下代码给出了@Reference 注解类型的定义

```

package org.osoa.sca.annotations;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target({METHOD, FIELD, PARAMETER})
@Retention(RUNTIME)
public @interface Reference {

    public String name() default "";
    public boolean required() default true;
}

```

@Reference 注解类型用于标注一个 Java 类的数据成员或用于注入一个解析引用的服务的 setter 方法。被注入服务的接口由 Java 类的属性域类型或 setter 方法的输入参数类型定义。

即使不存在@Reference 注解，也能通过 public 的 setter 方法注入引用。然而，为了注入引用到一个非 public 数据成员上，必须使用@Reference 注解。在没有@Reference 注解的情况下，引用的名字与数据成员或 setter 的名字相同。

对于某个引用，既有 setter 方法又有数据成员，则使用 setter 方法。

@Reference 注解有如下属性：

- **name(可选的)**—引用的名字，默认为Java类的数据成员名
- **required(可选的)**—是否必须注入服务。默认为true

以下代码片段展示通过数据成员注入引用。

```
@Reference(name="stockQuote", required=true)
protected StockQuoteService stockQuote;
```

以下代码片段展示通过 setter 注入引用。

```
@Reference(name="stockQuote", required=true)
public void setStockQuote( StockQuoteService theSQService );
```

以下来自构件实现的代码片段展示了通过@Reference 注解注入服务引用。引用名为“helloService”，且其类型为 HelloService。clientMethod()方法调用服务的“hello”操作，引用“helloService” 是此服务的引用。

```
private HelloService helloService;

@Reference(name="helloService", required=true)
public setHelloService(HelloService service){
    helloService = service;
}

public void clientMethod() {
    String result = helloService.hello("Hello World!");
    ...
}
```

@Reference 注解的出现在 componentType 信息中被反射出来。此 componentType 信息通过实现类上的反射由运行时产生。如下片段展示了上述构件实现片段的 component type 信息。

```
<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.oesa.org/xmlns/sca/1.0">

    <!--Any services offered by the component would be listed here Æ
    <reference name="helloService" multiplicity="1..1">
        <interface.java interface="services.hello.HelloService"/>
    </reference>

</componentType>
```

如果引用不是数组或集合，那么隐含的component type有一个带有multiplicity属性为0..1或1..1的引用，其multiplicity属性依赖于@Reference的required属性值，如果required=true则应用1..1。

如果引用定义为一个数组或java.util.Collection，则隐含的component type有一个带有multiplicity属性为1..n或0..n的引用，依赖于@Reference注解的required属性是设置为true还是false - 如果required=true则应用1..n。

如下构件实现的片段展示了一个服务引用的定义样例，其在java.util.List上使用@Reference注解。引用名为“helloServices”，类型为 HelloService。clientMethod()方法调用所有服务的“hello”操作，引用“helloService”引用了这些服务。这种情形下，至少应该存在一个HelloService，所以required为true。

```
@Reference(name="helloService", required=true)
```

```

protected List<HelloService> helloServices;
public void clientMethod() {
    ...
    HelloService helloService = (HelloService)helloServices.get(index);
    String result = helloService.hello("Hello World!");
    ...
}

```

如下片段展示了反射自上述构件实现片段的 component type 的 XML 描述。这种情形没有书写 component type，因为可以从 Java 类反射得来。

```

<?xml version="1.0" encoding="ASCII"?>
<componentType xmlns="http://www.osoa.org/xmlns/sca/1.0">

    <!--Any services offered by the component would be listed here Æ
    <reference name="helloService" multiplicity="1..n">
        <interface.java interface="services.hello.HelloService"/>
    </reference>
</componentType>

```

1.8.15. @Remotable

如下片段展示了 @Remotable 注解类型的定义。

```

package org.osoa.sca.annotations;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Remotable {

}

```

@Remotable 注解类型用于将一个 Java 服务接口标注为远程的。一个远程服务可以对外公布为一个服务，且必须转换为 WSDL portType 类型。

@Remotable 注解没有属性。

如下片段展示了带有 @Remotable 注解的远程服务 Java 接口。

```

package services.hello;
import org.osoa.sca.annotations.*;

```

```
@Remotable
public interface HelloService {
    String hello(String message);
}
```

远程接口的风格一般是粗粒度的，趋于松耦合交互。远程服务接口不允许使用方法的重载。

通过远程接口的复杂数据类型交互必须与服务绑定使用的 `marshalling` 技术相兼容。比如，如果服务打算通过标准的 `web service` 绑定暴露，那么参数必须是服务数据对象（SDO）2.0 [2]或 JAXB[3]类型。

无论对远程服务的调用来自包含该服务的复合构件的外部，还是来自同属一个复合构件的另一构件，数据交换语境都是传值。

远程服务的实现可以在调用过程中或调用发生后修改输入数据，且可以在调用后修改返回的数据。如果远程服务是被本地或远程调用的，SCA 容器负责确保对输入数据的修改，或调用后对返回数据的修改，不为调用者所见。

```
package services.hello;
```

```
import org.osoa.sca.annotations.*;
```

```
@Remotable
public interface HelloService {
    String hello(String message);
}
```

```
package services.hello;
```

```
import org.osoa.sca.annotations.*;
```

```
@Service(HelloService.class)
@AllowsPassByReference
public class HelloServiceImpl implements HelloService {
    public String hello(String message) {
        ...
    }
}
```

1.8.16.@Scope

如下片段展示@Scope 注解类型的定义。

```
package org.osoa.sca.annotations;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
```

```
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Scope {

    String value() default "STATELESS";
}
```

@Scope注解类型用于一个服务的接口定义之上或服务实现类上。

@Scope注解有如下属性：

- **value** – 作用域的名称。
默认值为'STATELESS'。对于'STATELESS'的实现，不同请求可由不同的实现示例提供服务。实现实例可以是新创建的或从实例池里获得。

如下片段展示了作用域服务接口定义的样例。

```
package services.shoppingcart;
import org.osoa.sca.annotations.Scope;

@Scope("CONVERSATION")
public interface ShoppingCartService {
    void addToCart(Item item);
}
```

1.8.17.@Service

如下片段展示了@Service注解类型的定义。

```
package org.osoa.sca.annotations;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface Service {
    Class<?>[] interfaces() default {};
    Class<?> value() default Void.class;
}
```

@Service注解类型用于构件实现类上指定实现所提供的SCA服务。类不需要声明实现所有服务暗指的接口，但服务接口的所有方法都必须存在。一个用作服务实现的类不要求有@Service注解。如果一个类没有@Service注解，那么提供哪些服务以及那些服务有哪些接口的规则由相关的实现类型决定。

@Service注解有如下属性：

- **interfaces** – 该值是一个应该暴露为此构件服务的接口或类对象的数组。
- **value** – 当类只提供了一个单一服务接口时的快捷方式。两个属性只能指定其中一个。

没有属性的@Service注解是没有意义的，就如同根本没有注解一样。

服务的名称默认为接口或类名，不包括包名。

如果一个Java实现需要实现相同接口的两个服务，那么要通过接口的子类化完成。子接口必须不能增加任何方法。两个接口都列入Java实现类的@Service注解中。

1.8.18. @ConversationAttributes

如下片段展示了@ConversationAttributes注解类型的定义。

```
package org.osoa.sca.annotations;
import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;

@Target(TYPE)
@Retention(RUNTIME)
public @interface ConversationAttributes {
    public String maxIdleTime() default "";
    public String maxAge() default "";
    public boolean singlePrincipal() default false;
}
```

@ConversationAttributes注解类型用于定义应用于服务或Java类引用的会话型接口的一系列属性。此注解有如下属性：

- **maxIdleTime (可选)** – 可以在单一会话中的操作间传递的最大时间。如果超过此时间，容器可能会终止会话。
- **maxAge (可选)** – 整个会话可以保持激活态的最大时间。如果超过此时间，容器可能会终止会话。
- **singlePrincipal (可选)** – 如果为true，只有启动会话的用户有权限继续会话。默认为false。

在两个取值为时间的属性中，时间表述为整数开始的字符串，后接空格，然后"seconds", "minutes", "hours", "days" or "years"其中一个。

未指定超时意味着超时由SCA运行时的实现定义，且选择这种方式。

如下片段给出了构件名属性域定义的样例。

```
package service.shoppingcart;
```

```
import org.osoa.sca.annotations.*

@ConversationAttributes (maxAge="30 days");
public class ShoppingCartServiceImpl implements ShoppingCartService {
    ...
}
```

1.8.19. @ConversationID

如下片段展示了@ConversationID 注解类型的定义。

```
package org.osoa.sca.annotations;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface ConversationID {
}
```

ConversationID 注解类型用于标注一个 Java 类属性域或 setter 方法，以便注入会话 ID。系统产生的会话 ID 总是字符串类型，但应用产生的会话 ID 可以是其它复杂类型。

如下片段展示了会话 ID 属性域定义的样例。

```
@ConversationID
private String ConversationID;
```

此属性域的类型可以不是 String 类型。

1.9. WSDL to Java 和 Java to WSDL

SCA的Java客户程序与实现模型应用 *WSDL to Java*和*Java to WSDL*映射规则（此映射规则在JAX-WS规范[4]中定义）从WSDL portType产生远程Java接口，反之亦然。

从Java类型到XML schema类型映射，SCA支持SDO 2.0[2]映射和JAXB[3]映射。

JAX-WS映射具有如下约束：

- 不支持holder

注意：此规范需要更多的关于JAX-WS的客户异步模型如何使用的案例与探讨。

2. Java 策略注解

SCA为在SCA装配集上附加策略相关元数据提供了便利，这些策略相关元数据将在运行时影响实现、服务以及引用的行为。策略工具在《SCA策略框架规范》[5]中描述。具体来说，便利包括意图与策略集，意图表达抽象的高层策略需求，而策略集则表达低层详细的具体策略。

策略元数据可以通过放入composite文档与component type文档中的声明性语句来添加到SCA装配集中。这些注解完全与实现代码无关，允许在应用开发的装配与部署阶段应用这些策略。

然而，将策略元数据直接附加到实现代码中是有用且也是更自然的方式。这在代码本身依赖于特定策略的地方特别重要。来自安全域的一个案例是实现代码希望运行在一个特定的安全角色下，且实现上被调用的任意服务操作都必须是授权的以保证客户有正确的权限来使用关注的操作。通过使用相应的策略元数据标注代码，开发者可以放心元数据在装配和开发阶段不被遗忘。

SCA Java通用注解规范提供了一系列注解，它们为开发者提供了将策略信息附加于Java实现代码的能力。这些相关的注解首先为将SCA意图与策略集附加于Java代码提供了一般性的便利。其次，更进一步还有特定的注解，为某个策略域，比如安全，处理特定的策略意图。

SCA Java通用注解规范支持Java平台通用注解规范(JSR-250) [6]的使用。采用此规范（这里指Java平台通用注解规范）暗示着SCA Java规范支持一致的注解与Java类继承关系。

2.1. 通用意图注解

SCA为将任意意图附加于Java类、Java接口或类以及接口中的元素，如方法、属性域提供了@Requires注解。

@Requires注解可以在一条语句中附加一个或多个意图。

每个意图都表述为一个字符串。意图是XML QName，由命名空间URI加上意图名组成。准确的字符串表述形式遵从javax.xml.namespace.QName类的定义，如下：

```
"{" + Namespace命名空间 URI + "}" + 意图名
```

意图可以是限定的，在这种情形下，由基本意图名，后接一个"."号，再接限定名组成字符串。也可能有多个限定级别。

这个描述是非常冗长的，所以我们希望为此字符串的命名空间部分，以及Java代码使用的每个意图定义

重用的字符串常量。SCA为意图定义常量如下：

```
public static final String SCA_PREFIX="{http://www.osea.org/xmlns/sca/1.0}";
public static final String CONFIDENTIALITY = SCA_PREFIX + "confidentiality";
public static final String CONFIDENTIALITY_MESSAGE = CONFIDENTIALITY + ".message";
```

注意，约定上来说，限定的意图将限定符作为常量名的一部分包含在内，由下划线分隔。这些意图常量在定义了相应意图注解的文件中定义（意图注解以及这些常量的正式定义见下一节）。

多个意图（限定的或未限定的）表述为一个数组声明中的分隔字符串。

带有2个限定意图（来自安全域）的@Requires注解的例子如下：

```
@Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

其附加了"confidentiality.message"和"integrity.message"意图。

如下是要求支持机密性引用的例子：

```
package org.osea.sca.annotation;
import static org.osea.sca.annotation.Confidentiality.*;

public class Foo {
    @Requires(CONFIDENTIALITY)
    @Reference
    public void setBar(Bar bar)
    ...
}
```

用户也可以选择只使用 QName 的命名空间部分的常量，所以他们可以不必定义新的常量而添加新的意图。那种情形下，上例中的定义会变成：

```
package org.osea.sca.annotation;
import static org.osea.sca.Constants.*;

public class Foo {
    @Requires(SCA_PREFIX+"confidentiality")
    @Reference
    public void setBar(Bar bar)
}
```

@Requires 注解的正式语法如下：

```
@Requires(“qualifiedIntent” | {“qualifiedIntent” [, “qualifiedIntent”]})
```

where

```
qualifiedIntent ::= QName | QName.qualifier | QName.qualifier1.qualifier2
```

如下展示了@Requires 注解的正式定义：

```

package org.osoa.sca.annotation;

import static java.lang.annotation.ElementType.TYPE;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import java.lang.annotation.Inherited;

@Inherited
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})

public @interface Requires {
    String[] value() default "";
}

```

SCA_NS 常量在 Constants 接口中定义:

```

package org.osoa.sca;

public interface Constants {
    public static final String SCA_NS=
        "http://www.osoa.org/xmlns/sca/1.0";
    public static final String SCA_PREFIX = "{"+SCA_NS+"}";
}

```

2.2.特定意图注解

除了上述由@Requires 注解提供的一般意图注解外，还可以有与特定策略意图相关的 Java 注解。SCA 提供了许多这样的特定意图注解，并且可以为任意意图创建新的特定的意图注解。

特定意图注解的一般形式是注解名派生自意图名的一个注解。如果此意图为限定意图，限定符以一个字符串或字符串数组的形式作为此注解的属性提供。

比如，在“通用意图注解”节中用@Requires(CONFIDENTIALITY)意图描述的 SCA confidentiality 意图也可以用特定的@Confidentiality 意图注解指定。完整性安全意图的特定意图注解为：

```
@Integrity
```

对于"authentication"意图，其特定的限定意图例子为：

```
@Authentication( { "message", "transport" } )
```

此意图附加了限定意图对: "authentication.message"和"authentication.transport"。(假定这些情形下, sca 命名空间为"http://www.osoa.org/xmlns/sca/1.0")。

特定意图注解的一般形式为:

```
@<Intent>[(qualifiers)]
```

这里 Intent 是一个 NCName, 指定一个特定的意图类型。

```
Intent ::= NCName
```

```
qualifiers ::= "qualifier" | { "qualifier" [, "qualifier"] }
```

```
qualifier ::= NCName | NCName/qualifier
```

2.2.1. 如何生成特定意图注解

SCA通过提供一个@Intent注解来标识意图相对应的注解。此@Intent注解必须用在一个意图注解的定义中。

@Intent注解有单个参数, 像@Requires注解一样是一个注解的QName的字符串形式。作为意图定义的部分, 为命名空间、意图以及意图的限定版本创建字符串常量是很好的(尽管不是必须的)。然后这些字符串常量可以用于@Requires注解, 且将这些字符串的一个或多个用作@Intent注解的参数也是可以的。

可选地, 意图的QName可以用targetNamespace和localPart各自的参数来指定, 如下:

```
@Intent(targetNamespace=SCA_NS, localPart="confidentiality").
```

@Intent 注解的定义如下:

```
package org.osoa.sca.annotation;
import static java.lang.annotation.ElementType.ANNOTATION_TYPE;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import java.lang.annotation.Inherited;
```

```

@Retention(RUNTIME)
@Target(ANNOTATION_TYPE)
public @interface Intent {

    String value() default "";
    String targetNamespace() default "";
    String localPart() default "";

}

```

当一个意图可以限定时，将注解的第一个参数定义成承载一个或多个限定符的字符串（或字符串数组）是很好的实践。

在这种情形下，应该用@Qualifier 注解标注属性的定义。@Qualifier 告诉 SCA 应该将属性值看作是由整个注解描述的意图的限定符。如果在一个注解中指定了多个限定符的值，则意味着要求有多个限定形式。比如：

```
@Confidentiality({"message", "transport"})
```

暗示着限定意图“confidentiality.message”和“confidentiality.transport”被设置为附加于 confidentiality 意图的元素。

```

package org.osoa.sca.annotation;

import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.RetentionPolicy.RUNTIME;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import java.lang.annotation.Inherited;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Qualifier {

}

```

特定意图注解的定义中的@Intent 和@Qualifier 注解的使用例子在“处理安全交互策略”节中展示。

2.3. 意图注解的应用

SCA意图注解可以应用于如下Java元素：

- Java class
- Java interface

- Method
- Field

这里多个意图注解（一般或特定的）应用于同一个Java元素，它们是叠加效果。多个策略注解一起使用的例子如下：

```
@Authentication
@Requires({CONFIDENTIALITY_MESSAGE, INTEGRITY_MESSAGE})
```

在这种情形下，有效的意图为"authentication", "confidentiality.message" 以及 "integrity.message"。

如果在类/接口级和方法或属性域级都指定了注解，那么方法或属性域级注解完全覆盖同类型的类级注解。

在SCA service上添加注解策略时，意图注解可以应用于类或类方法。用引用注入的形式应用一个意图到 setter方法上允许在引用时定义意图。

2.3.1.继承和注解

注解的继承规则与 JSR250 通用注解规范一致。

如下例子展示了类、操作以及超类上意图的继承关系。

```
package services.hello;
import org.osoa.sca.annotations.Remotable;
import org.osoa.sca.annotations.Integrity;
import org.osoa.sca.annotations.Authentication;

@Remotable
@Integrity("transport")
@Authentication

public class HelloService {
    @Integrity
    @Authentication("message")
    public String hello(String message) {...}

    @Integrity
    @Authentication("transport")
    public String helloThere() {...}
}
```

```
package services.hello;
import org.osoa.sca.annotations.Remotable;import
org.osoa.sca.annotations.Confidentiality;
import org.osoa.sca.annotations.Authentication;

@Remotable
@Confidentiality("message")
```

```

public class HelloChildService extends HelloService {
    @Confidentiality("transport")
    public String hello(String message) {...}
    @Authentication
    String helloWorld(){...}
}

```

例Example 2a. Usage example of annotated policy and inheritance.

helloWorld 方法上有效的意图注解是 Integrity("transport"), @Authentication, 以及 @Confidentiality("message").

HelloChildService 的 hello 方法上的有效意图注解是 @Integrity("transport")、@Authentication 和 @Confidentiality("transport")。

HelloChildService 的 helloThere 方法上的有效意图注解是 @Integrity、@Authentication("transport"), 与 HelloService 类相同。

HelloService 的 hello 方法上的有效意图注解是 @Integrity 和 @Authentication("message")。

如下的列表包含了与例 2a 中的 Java 接口、类对应的 HelloService 以及 HelloChildService 实现等效的声明性安全交互策略。

```

<?xml version="1.0" encoding="ASCII"?>
<composite xmlns="http://www.oesa.org/xmlns/sca/1.0"
           name="HelloServiceComposite" >
  <service name="HelloService" requires="integrity/transport
    authentication">
    ...
  </service>
  <service name="HelloChildService" requires="integrity/transport
    authentication confidentiality/message">
    ...
  </service>
  ...
  <component name="HelloServiceComponent">*
    <implementation.java class="services.hello.HelloService"/>
    <operation name="hello" requires="integrity
      authentication/message"/>
    <operation name="helloThere" requires="integrity
      authentication/transport"/>
  </component>
  <component name="HelloChildServiceComponent">*
    <implementation.java class="services.hello.HelloChildService" />
    <operation name="hello" requires="confidentiality/transport"/>
    <operation name="helloThere" requires="integrity/transport
      authentication"/>
    <operation name="helloWorld" requires="authentication"/>
  </component>
  ...
</composite>

```

例2b. 声明性意图, 等效于例2a中的注解性意图

2.4. 声明性意图和注解性意图的关系

Java 类上的注解性意图既不能被将此 Java 类作为实现的 composite 文档中的声明性意图覆盖，也不能被与此 Java 类相关的 component Type 文档中的语句覆盖。此规则遵从意图的一般性规则，代表了对实现的基本要求。

以 Java 类中的注解的形式表达的非限定的意图，可由 composite 文档中的声明性意图作出限定。

2.5. 策略集注解

SCA策略框架使用策略集来获取细节的低层具体策略（比如，在加密消息时使用特定通讯协议来将服务连接具体策略）。

使用 **@PolicySets** 注解可以直接将策略集应用于 Java 实现。PolicySets 注解既可以将单个策略集的 QName 设置作为一个字符串，也可以将两个或多个策略集合名称设置为一个字符串数组。

```
@PolicySets( "<policy set QName>" |
            { "<policy set QName>" [, "<policy set QName>"] })
```

像意图一样，策略集名是 QName，形式为“{Namespace-URI}localPart”。

@PolicySets 注解的例子如下：

```
@Reference(name="helloService", required=true)
@PolicySets({ MY_NS + "WS_Encryption_Policy",
             MY_NS + "WS_Authentication_Policy" })
public setHelloService(HelloService service){
    . . .
}
```

这种情形下，策略集 WS_Encryption_Policy 和 WS_Authentication_Policy 被应用，两个都使用了常量 MY_NS 定义的命名空间。

根据策略框架规范[\[5\]](#)中定义的规则，当两个都存在时，策略集必须满足实现表述的意图。

SCA 策略集注解可以应用于如下 Java 元素：

- Java class
- Java interface
- Method
- Field

2.6. 安全策略注解

本节介绍 SCA 的安全意图注解，安全意图注解见 SCA 策略框架规范[5]中的定义。

2.6.1. 安全交互策略

如下的交互策略意图和限定符是为安全策略定义的，它们应用于实现的服务与引用的操作上。

- @Integrity
- @Confidentiality
- @Authentication

这三个意图的全部都有相同的限定符：

- message
- transport

如下片段展示了@Integrity、@Confidentiality 以及@Authentication 注解类型的定义

```
package org.osoa.sca.annotation;
import java.lang.annotation.*;
import static org.osoa.sca.Constants.SCA_NS;

@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD,
        ElementType.FIELD ,ElementType.PARAMETER})
@Intent(Integrity.INTEGRITY)
public @interface Integrity {
    public static final String INTEGRITY = SCA_NS+"integrity";
    public static final String INTEGRITY_MESSAGE = INTEGRITY+".message";
    public static final String INTEGRITY_TRANSPORT = INTEGRITY+".transport";
    @Qualifier
    String[] value() default "";
}

package org.osoa.sca.annotation;
import java.lang.annotation.*;
import static org.osoa.sca.Constants.SCA_NS;
```

```

@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE,ElementType.METHOD,
        ElementType.FIELD ,ElementType.PARAMETER})
@Intent(Confidentiality.CONFIDENTIALITY)
public @interface Confidentiality {
    public static final String CONFIDENTIALITY = SCA_NS+"confidentiality";
    public static final String CONFIDENTIALITY_MESSAGE =

        CONFIDENTIALITY+".message";
    public static final String CONFIDENTIALITY_TRANSPORT =
        CONFIDENTIALITY+".transport";
    @Qualifier
    String[] value() default "";
}

package org.osoa.sca.annotation;

import java.lang.annotation.*;
import static org.osoa.sca.Constants.SCA_NS;

@Inherited
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE,ElementType.METHOD,
        ElementType.FIELD ,ElementType.PARAMETER})
@Intent(Authentication.AUTHENTICATION)
public @interface Authentication {
    public static final String AUTHENTICATION = SCA_NS+"authentication";
    public static final String AUTHENTICATION_MESSAGE =
        AUTHENTICATION+".message";
    public static final String AUTHENTICATION_TRANSPORT =
        AUTHENTICATION+".transport";
    @Qualifier
    String[] value() default "";
}

```

如下例子展示了将意图应用于 setter 方法的例子，此 setter 方法用来注入一个引用。访问引用的 HelloService 的 hello 操作要求有"integrity.message"和"authentication.message"意图。

```

//Interface for HelloService
public interface service.hello.HelloService {
    String hello(String helloMsg);
}

// Interface for ClientService
public interface service.client.ClientService {
    public void clientMethod();
}

// Implementation class for ClientService
package services.client;

import services.hello.HelloService;

```

```

import org.osoa.sca.annotations.*;

@Service(ClientService.class)
public class ClientServiceImpl implements ClientService {

    private HelloService helloService;

    @Reference(name="helloService", required=true)

    @Integrity("message")
    @Authentication("message")
    public void setHelloService(HelloService service){
        helloService = service;
    }

    public void clientMethod() {
        String result = helloService.hello("Hello World!");
        ...
    }
}

```

Example 1. Usage of annotated intents on a reference.

2.6.2. 安全实现策略

SCA定义了许多安全策略注解，用作实现自身的策略。这些注解大多数与认证和安全标识有关。SCA支持如下的认证与安全标识注解（在JSR250中定义）：

- RunAs

带有一个字符串参数，此参数为安全角色的名字。

用此注解标注的代码将以此标识角色的安全权限执行。

- RolesAllowed

带有单个字符串的参数或一个字符串数组参数，描述一个或多个角色名。当存在时，实现只能被与@roles属性列出的角色名之一相应的责任人访问。角色名是如何映射到安全责任人的与实现相关（SCA没有定义这个）。

eg. @RolesAllowed({"Manager", "Employee"})

- PermitAll

没有参数。当存在时，授权所有角色都能访问。

- DenyAll

没有参数。当存在时，拒绝所有角色访问。

- DeclareRoles

带有一个字符串参数或一个字符串数组参数，标识一个或多个角色名，组成实现所使用的一系列角色的集合。

eg. `@DeclareRoles({"Manager", "Employee", "Customer"})`

(所有这些都Java包`javax.annotation.security`中声明)

这些意图的完整解释，请查阅策略框架规范[\[5\]](#)。

2.6.2.1. Annotated Implementation Policy 例子

如下例子展示了被注解的安全实现策略：

```
package services.account;

@Remotable
public interface AccountService{
    public AccountReport getAccountReport(String customerID);
}
```

如下是 `AccountServiceImpl` 类的完整代码，展示了其实现的服务，外加其服务引用和可设置的属性，连同一系列实现策略注解：

```
package services.account;
import java.util.List;
import commonj.sdo.DataFactory;
import org.osoa.sca.annotations.Property;
import org.osoa.sca.annotations.Reference;
import org.osoa.sca.annotations.RolesAllowed;
import org.osoa.sca.annotations.RunAs;
import org.osoa.sca.annotations.PermitAll;
import services.accountdata.AccountDataService;
import services.accountdata.CheckingAccount;
import services.accountdata.SavingsAccount;
import services.accountdata.StockAccount;
import services.stockquote.StockQuoteService;
@RolesAllowed("customers")
```

```

@RunAs("accountants" )
public class AccountServiceImpl implements AccountService {

    @Property
    protected String currency = "USD";

    @Reference
    protected AccountDataService accountDataService;
    @Reference
    protected StockQuoteService stockQuoteService;

    @RolesAllowed({"customers", "accountants"})
    public AccountReport getAccountReport(String customerID) {

        DataFactory dataFactory = DataFactory.INSTANCE;
        AccountReport accountReport =
            (AccountReport)dataFactory.create(AccountReport.class);
        List accountSummaries = accountReport.getAccountSummaries();

        CheckingAccount checkingAccount =
            accountDataService.getCheckingAccount(customerID);
        AccountSummary checkingAccountSummary =
            (AccountSummary)dataFactory.create(AccountSummary.class);
        checkingAccountSummary.setAccountNumber(checkingAccount.getAccountNumber());
        checkingAccountSummary.setAccountType("checking");
        checkingAccountSummary.setBalance(fromUSDollarToCurrency
            (checkingAccount.getBalance()));
        accountSummaries.add(checkingAccountSummary);

        SavingsAccount savingsAccount =
            accountDataService.getSavingsAccount(customerID);
        AccountSummary savingsAccountSummary =
            (AccountSummary)dataFactory.create(AccountSummary.class);
        savingsAccountSummary.setAccountNumber(savingsAccount.getAccountNumber());

        savingsAccountSummary.setAccountType("savings");
        savingsAccountSummary.setBalance(fromUSDollarToCurrency
            (savingsAccount.getBalance()));
        accountSummaries.add(savingsAccountSummary);

        StockAccount stockAccount = accountDataService.getStockAccount(customerID);
        AccountSummary stockAccountSummary =
            (AccountSummary)dataFactory.create(AccountSummary.class);
        stockAccountSummary.setAccountNumber(stockAccount.getAccountNumber());
        stockAccountSummary.setAccountType("stock");
        float balance= (stockQuoteService.getQuote(stockAccount.getSymbol()))*
            stockAccount.getQuantity();
        stockAccountSummary.setBalance(fromUSDollarToCurrency(balance));
        accountSummaries.add(stockAccountSummary);

        return accountReport;
    }

    @PermitAll
    public float fromUSDollarToCurrency(float value){

```

```
    if (currency.equals("USD")) return value; else
    if (currency.equals("EURO")) return value * 0.8f; else
    return 0.0f;
}
}
```

Example 3. Usage of annotated security implementation policy for the java language.

例子3中，注解了整个实现类：

- `@RolesAllowed("customers")` - 指明customers可以访问整个实现
- `@RunAs("accountants")` - 指明实现的代码以accountants权限运行。

`getAccountReport(..)`方法用`@RolesAllowed({"customers", "accountants"})`标注，指示此方法可以被customers和accountants调用。

`fromUSDollarToCurrency()`方法用`@PermitAll`标识，意味着此方法可以被任意角色调用。

3. 附录

3.1 参考文献

[1] SCA Assembly Specification

http://www.osoa.org/download/attachments/35/SCA_AssemblyModel_V100.pdf

[2] SDO 2.0 Specification

<http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf>

[3] JAXB Specification

<http://www.jcp.org/en/jsr/detail?id=31>

[4] WSDL Specification

WSDL 1.1: <http://www.w3.org/TR/wsd1>

WSDL 2.0: <http://www.w3.org/TR/wsdl20/>

[6] Common Annotation for Java Platform specification (JSR-250)

<http://www.jcp.org/en/jsr/detail?id=250>

后记：SCA 中文规范项目

声明

SCA相关规范的中文文档得到 OSOA Chinese Community (OSOA中文社区) 的直接授权和有力的支持,其目的是在中文世界推广优秀的SOA相关技术标准。本次翻译活动由满江红开放技术研究组织(<http://www.redsaga.com>)和OSOA中文社区(<http://www.osoa.org/pages/viewpage.action?pageId=416>)共同发起、组织,将翻译SCA的绝大多数规范,并得到goCom社区(<http://www.gocom.cc/>)的赞助。我们在此郑重宣布,本次翻译遵循原Service Component Architecture Specification的授权协议。在完整保留全部文本包括本版权页,并不违反Service Component Architecture Specification协议的前提下,允许和鼓励任何人进行全文转载及推广。我们在此宣布所有参与人员放弃除署名权外的一切权利。

致谢

SCA 规范的翻译对很多人来说都是一个挑战,再次感谢所有参与翻译、评审工作的同学们,没有你们的辛勤劳动,也就不会有中文规范的面世。

参与人员

规范分配情况,最后的校对和统稿由管理员统一完成。

Specification	翻译	一审	二审
SCA Assembly Model V1.00	ligang1111	wangfeng	wangfeng
SCA Policy Framework V1.00	liang_ma,max	pesome	needle
SCA Transaction Policy V1.00	max	jiangxd	guangh
SCA Java Common Annotations and APIs V1.00	ligang1111	HiugongGwok	dc
SCA Java Component Implementation V1.00	ligang1111	pinelygao	nekesai
SCA Spring Component Implementation V1.00	ligang1111	pinelygao	Caozw

SCA BPEL Client and Implementation V1.00	jiangxd	hongsoft	hongsoft
SCA C++ Client and Implementation V1.00	SpringWater	SpringWater s	SpringWater s
SCA Web Services Binding V1.00	xichengmylove	hongsoft	hongsoft
SCA JMS Binding V1.00	YuLimin/Echo	hongsoft	pegasus
SCA EJB Session Bean Binding V1.00	max	hongsoft	yanfei
SCA JCA Binding V1.00 (暂不发布)	bsspirit	jiangxd	benja

参与人员列表:

网名	姓名	网名	姓名
billytree	冯博	guangh	光华
ligang1111	李刚	Caozw	小曹
xichengmylove	严永华	yanfei	晏斐
pesome	张俊	benja	老贾
max	孙浩	HiugongGwok	郭晓刚
needle	needle	bsspirit	张丹
YuLimin	俞丽敏	Echo	杨春花
wangfeng	王锋	pinelygao	高松
hongsoft	洪波	liang_ma	马亮
jiangxd	姜晓东	Dc	王葱权
SpringWater	张世富	pegasus	马捷
chrischengzh	chris	nekesai	蔡永保
jiaoly	老焦	keqiang	克强

下列人员报名,但由于种种原因没有参与,在此感谢:

网名	姓名	网名	姓名
jetyang_1	杨文明	cenwenchu	岑文初
lafay	叶江	Eric	孟庆余
laodingshan	丁跃斌		

项目历程

SCA 规范一期 翻译项目进度汇总

2008年03月17日

3月17日，SCA 规范一期的翻译项目正式启动，Wiki 中增加了相应的 SCA 板块和对应的几个栏目。在 JavaEye 和 goCom 上发了招募贴，在满江红邮件列表中发了封邮件。

希望有更多的同学能够参加到这次的项目中，积极报名，争取能早日高质量地完成所有的工作，让更多人能享受到我们的劳动成果~

2008年03月22日

相当郁闷，我在 JavaEye 上的帖子竟然被隐藏了，也忒欺负人了。经过沟通之后，终于又恢复了，又在 JavaEye 发了一遍新闻，呵呵。

2008年03月28日

通过这段调查，发现 SCA 规范和 Spring 比较，相当于 阳春白雪 VS 下里巴人，研究规范的都是比较专业的人士。看来我们的翻译工作很有必要啊，在国内推广 SOA。

2008年04月02日

昨天是愚人节，结果自己被愚弄了一把，看到 Spring 被微软收购了，真是“很傻很天真”。

鉴于这次规范的翻译的特殊性，有相当多的规范已经有中文草稿，所以没有采取其他文档翻译按章节划分的做法。

截至今天为止参加人数已经有 10 几个人，一期计划翻译的 9 个规范都已经被领走了，ligang1111 同学一个人承包了 5 个规范，而且都是分量最重，而且内容最多的部分。最后一个 JMS binding 规范翻译是被 俞丽敏夫妇领走的，革命伴侣，令人羡慕！

希望有更多的人参与进来，现在正在招募一审，二审人员。

2008年04月06日

CVS 服务器已经设置好，上传了规范英文文档，相关翻译人员分配了 CVS 帐号，大家可以开始干活了，呵呵。

2008年04月08日

刚同学已经完成 SCA Spring Component Implementation Specification V100 翻译，在这里向他的辛勤劳动表示祝贺！

他为我们已经开了一个好头！

其他已经领取任务的同志们加油啊！

2008年04月10日

昨天满江红网站出了故障访问不了，还好今天已经正常了。

xichengmylove 同学已经完成 SCA Web Services Binding V1.00 翻译，max 已经完成领取任务的 30%。在这里向他们们的辛勤劳动表示祝贺！

其他已经领取任务的同志们加油啊！请及时上报翻译进度。

2008年04月11日

我们又有新鲜血液注入啦： 让我们欢迎 jiangxd, SpringWater 。他们分别承担了 SCA BPEL client, SCA C++ Client 的翻译工作，鼓掌！

SCA BPEL Client and Implementation V1.00	jiangxd		已领取	15	2008.4.11
SCA C++ Client and Implementation V1.00	SpringWater		已领取	70	2008.4.11

2008年04月21日

前面一周出差了，回来发现很多文档都翻译完了，但是有些翻译文档没有上传，请大家及时上传，有问题直接和我联系。

众人拾柴火焰高！原定二期翻译的 SCA C++ Client and Implementation V1.00 规范已经完成 70%的翻译，向 SpringWater 致敬！

目前比较头疼的是，一审二审工作，还望群策群力。

2008年04月24日

昨天下午去听了 SOA 中国的关键任务 上海站 报告，会上有个仁兄替我们 SCA 翻译作了个小广告，弱弱的问下，是那位啊？

感谢李刚同学！ 翻译完成 [SCA Assembly Model V1.00](#) ，分量最重的这块。

让我们欢迎 teamlet！他将负责 [SCA Assembly Model V1.00](#) 的一审工作！

2008年04月28日

SpringWater 今天将 SCA_ClientAndImplementationModel_Cpp-V100 翻译文档发给我了，我看了一下，其实我们计划的翻译阶段即将完成了！

再次感谢所有参与者的辛苦劳动！

现在面临的问题是 翻译文档的一审、二审工作，我将联系 OSOA 中文社区、Tuscany 中文社区、goCom 和 JavaEye 社区的 SOA 爱好者，也向所有的 SOA 爱好者发出邀请，希望大家热情参与！

2008年04月29日

laodingshan 同学今天领取了 SCA ComponentImplementation_V100 和 SCA_SpringComponentImplementationSpecification-V100 的一审工作，虽然他是个老同志，但是参与 SOA 的热情很高！

对他的加入表示热烈欢迎!

2008年05月15日

今天有了个想法,这次翻译工作我们也能拉到一部分赞助,因此我想征求大家的意见,设立一个伯乐奖,对于推荐熟悉 SOA 相关技术,参与标准翻译"千里马"的"伯乐"也给与这次翻译活动的纪念品,希望大家积极推荐!

2008年05月19日

沉痛悼念汶川大地震死难同胞!

2008年05月20日

liang_ma 同学主动承担了 **SCA Policy Framework V1.00** 规范的翻译工作,这个规范比较长,有 46 页,让我们为他加油打气!

2008年05月27日

SCA Java Common Annotations and APIs V1.00 已经翻译完毕,祝贺 ligang1111;

HiugongGwok 同学将承担一审工作,欢迎 HiugongGwok !

bsspirit 同学将承担 SCA_JCABindings v 1.0 的翻译,欢迎!

2008年06月05日

这一周,大家很忙啊,进度没有及时汇报,兄弟们要加油啊,一鼓作气!

提前预祝端午节快乐!

2008年06月12日

这是第一个法定的端午节假期,我们勤劳的 jiangxd 同志加班加点,完成了 **SCA Transaction Policy V1.00** 的一审工作。在此表示祝贺!

bsspirit 同学承担 SCA_JCABindings v 1.0 的翻译进度也很快,工作量已完成 70%了。

2008年06月17日

bsspirit 同学完成 **SCA JCA Binding V1.00** 的翻译工作,在此表示祝贺!

由于 laodingshan 同志自身工作很忙,所以他负责的 SCA ComponentImplementation_V100 和 SCA_SpringComponentImplementationSpecification-V100 一审工作现在由 pinelygao 接替。

pinelygao 同志有着 8 年的 J2ee 开发、架构经历,将会给予我们很大的支援!

2008年06月20日

在 **SCA Policy Framework V1.00** 规范翻译中,liang_ma 同学可能遇到了困难。进度比较缓慢。

我们不得不承认，这对他是一个挑战，而且他已经翻译了 40%了 !!!

这个规范是最后一个正在翻译的规范了。我们惊喜的发现，现在翻译的规范数比我们原来计划的要多很多了。

在我们为他加油打气的同时，我们也希望，期待援助：圈内人士，如果有对此感兴趣，欢迎积极加入，SCA Policy Framework V1.00 是一个很重要的规范，也是下一步研究的热点。规范总共 46 页。

希望大家踊跃认领，支持！

补充：伟大的 jiangxd 同志又承担了 **SCA JCA Binding V1.00** 一审工作，我已经不知道说什么感谢好了。。。。。

2008 年 06 月 23 日

max 同志将承担起 **SCA JCA Binding V1.00** 的剩余翻译工作。这样我们的工作将可以 7 月份完成。届时组织一次集体规范的二审工作。

接下来，SCA 相关规范的中文版将面世。

请 liang_ma 同学尽快将你已翻译的部分上传或者发给我。

SCA Policy Framework V1.00 一审已经被 pesome 领取了，感谢 pesome !!!

2008 年 06 月 25 日

今天把 liang_ma 负责的 SCA_Policy_Framework_V100 翻译上传了，他已经翻译到了 1.4 节。

max 同志终于可以开始工作了。

感谢天，感谢地，看来原定计划 不会难产了。

2008 年 06 月 26 日

今天是个好日子。

喜报！

hongsoft 同学负责的 **SCA BPEL Client and Implementation V1.00**，**SCA EJB Session Bean Binding V1.00** 一审完毕。

pinelygao 同学负责的 **SCA Java Component Implementation V1.00** 一审完毕

而且，hongsoft 同学还将接替 needle,完成 **SCA Web Services Binding V1.00** 和 **SCA JMS Binding V1.00** 的一审工作。

向勤劳的 hongsoft 大厨表示致敬!!!

2008 年 07 月 07 日

新的规范出来了，**SCA Java EE Integration Specification V1.00**，看了下，ORACLE 和 BEA 的人加起来比 IBM 还要多。没有看到 Primeton 的人，看来我们中国厂商要从标准跟随者到领导者，还要多多努力。

SCA Policy Framework V1.00 的翻译工作非常迅速，感谢 max 的辛苦工作，今天他告诉我只剩下 1.4 节和 liang_ma 交界的地方了。

今天是鄙人的生日，假公济私一下，哈哈。。

2008 年 07 月 10 日

HiugongGwok 承担的 SCA Java Common Annotations and APIs V1.00 一审工作已经完成。

而且，max 同志承担的 **SCA JCA Binding V1.00** 翻译已经高质量的完成，细心的他将在今晚自己审查后，上传。对 max 同志的高度责任心，一丝不苟的认真态度，值得我们学习。

再次感谢 2 位同志，胜利的曙光已经越来越近了。

还有一个好消息，我已经联系了相关赞助，将为我们这次的 SOA 规范翻译活动出专门的纪念 T 恤，以为各位同志的辛勤工作给少许心理上的安慰。

2008 年 07 月 11 日

max 同志承担的 **SCA Policy Framework** 已经上传 CVS，pesome 同志要接着进行“火炬”传递，完成一审工作。

SCA JCA Binding V1.00 一审工作在伟大的 jiangxd 同志的辛苦工作下已经完成，他自己本身工作也很忙，在经常加班的情况下，还认真审阅了规范的翻译，进行了大量认真、细致的修改、校正工作。

胜利的曙光。。。。。

2008 年 07 月 14 日

wangfeng 同学已经完成了 **SCA Assembly Model V1.00** 这个最长规范的一审工作，鉴于该同志一如既往地兢兢业业，经研究决定，对该同志进行通报表扬，呵呵！

2008 年 07 月 21 日

pesome 同学负责的 **SCA Policy Framework V1.00** 一审工作已经完成 50%，预计将于本周末全部完成。

这样我们将在 8 月 2 号组织一次集中的二审工作，这样整个一期规范将于 8 月中旬正式发布。

2008 年 07 月 23 日

昨天当了一次黄世仁，呵呵。

感谢 pinelygao 同学提交了 **SCA Java Component Implementation V1.00**，**SCA Spring Component Implementation V1.00** 一审文档。

2008 年 07 月 28 日

pesome 同学负责的 **SCA Policy Framework V1.00** 一审工作已经完成,祝贺阿

一个不好的消息是 SCA JCA 规范需要因为翻译质量不过关，需要重新翻译

好消息是二审工作已经全面铺开了

2008年08月18日

刘翔退赛了。

我们的 SCA 规范翻译二审工作还在继续。。。

SCA 规范目前正在进行紧张的二审工作，在欣赏到众多志愿者翻译成果的同时，也发现了不少的问题。

在此，我们诚征志愿者，对有问题的中文规范进行修改，我们提供中文版规范和翻译中问题的批注供您参考。并且你将会署名在发布的规范中。

另外，对于参与翻译工作的人员，我们提供了一些纪念品。目前的想法包括：

1. 按翻译页数奖励 goCom 币，每页奖励 goCom 币 10 分。

(用此 G 币可以在 goShop 商城购买商品，凡卓越网上能看到的商品都可以在 goCom 上通过获得的 G 币进行购买)

2. 打造成为 goCom 专家组成员。

(goCom 近期将推出专家组概念，专家组成员将在 goCom 网站专家页面享有独立页面用以展示其个人内容，goShop 商城中享受更高折扣，各级评比中享受 10% 的加分，北京用户免费参与 goCom 户外活动)

3. 邀请参加 goCom 在线技术日活动，每次活动可奖励 1500goCom 币。

4. 赠送最新银弹杂志与 goCom 纪念 T 恤。

2008年08月29日

日志好久没有更新了。二审工作确实比较辛苦，跟每个规范的翻译质量有很大关系，而且每个人对规范的理解也不一样。

郁闷。。。。

我们需要专业的 SCA 人员，作校对和最后的把关！

- 但是不管怎样，我们还要前行，接下来，将陆续发布规范，首先是装配规范和 java 注解和实现规范。*

希望大家多提宝贵意见，在拍砖的同时能够体会我们翻译者的苦与痛。

讨论、建议和勘误

进行规范的翻译确实是一个挑战，错误和翻译不当之处在所难免，请在尊重作者辛勤劳动的基础上，提出你们的宝贵建议。

我们在 goCom OSOA 中文专区 (<http://www.gocom.cc/modules/osoal/>) 上专门开辟了一个板块，欢迎指出错误和讨论。

技术圈子

欢迎您加入 SOA 技术圈子 <http://groups.google.com/group/SOAer>

SOAer 是一个 GoogleGroup, 聚集了国内 SOA 方面的架构师、咨询师、技术专家和技术爱好者。

主要活动包括:

- 1 SOA 相关开源的运作;
- 2 SOA 技术讨论;
- 3 SOA 相关热点问题的讨论;
- 4 SOA 规范: SCA/SDO 的翻译;
- 5 SOA 相关技术的高级培训;
- 6 特邀讲座及报道;
- 7 SOA 相关规范及书籍翻译;

我们本着“talk u like and do u like!”、“一起分享, 一起成长!”的宗旨, 一起探讨 SOA 的方方面面。

满江红

满江红开放技术研究组织(www.redsaga.com)长期以来致力于推动开放源代码事业在中国的发展。我们欢迎有能力、有热情的你加入我们, 一起达到从未想过的高度!

招募

欢迎大家参与SCA、SDO后续规范的翻译。敬请关注: <http://groups.google.com/group/SOAer>