Git 配置仓库

在 Spring Cloud Config 的服务端，对于配置仓库的默认实现采用了 Git。Git 非常适用于存储配置内容，它可以非常方便地使用各种第三方工具来对其内容进行管理更新和版本化，同时 Git 仓库的 Hook 功能还可以帮助我们实时地监控配置内容的修改。其中，Git 自身的版本控制功能正是其他一些配置中心所欠缺的，通过 Git 进行存储意味着，一个应用的不同部署实例可以从 Spring Cloud Config 的服务端获取不同的版本配置，从而支持一些特殊的应用场景。

由于 Spring Cloud Config 中默认使用 Git，所以对于 Git 的配置也非常简单，只需在 Config Server 的 `application.properties` 中设置 `spring.cloud.config.server.git.uri` 属性，为其指定 Git 仓库的网络地址和账户信息即可，比如在快速入门一节中的例子：

```
spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/SpringCloud-Learning/
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
```

如果我们将该值通过 `file://`前缀来设置为一个文件地址（在 Windows 系统中，需要使用 `file:///`来定位文件内容），那么它将以本地仓库的方式运行，这样我们就可以脱离 Git 服务端来快速进行调试与开发，比如：

```
spring.cloud.config.server.git.uri=file://${user.home}/config-repo
```

其中，`${user.home}`代表当前用户的所属目录。`file://`配置的本地文件系统方式虽然对于本地开发调试时使用非常方便，但是该方式也仅用于开发与测试，在生产环境中请务必搭建自己的 Git 仓库来存储配置资源。

占位符配置 URI

`{application}`、`{profile}`、`{label}`这些占位符除了用于标识配置文件的规则之外，还可以用于 Config Server 中对 Git 仓库地址的 URI 配置。比如，我们可以通过`{application}`占位符来实现一个应用对应一个 Git 仓库目录的配置效果，具体配置实现如下：

```
spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/{application}
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
```

其中，`{application}`代表了应用名，所以当客户端应用向 Config Server 发起获取配置的请求时，Config Server 会根据客户端的 `spring.application.name` 信息来填充`{application}`占位符以定位配置资源的存储位置，从而实现根据微服务应用的属性动

态获取不同位置的配置。另外，在这些占位符中，{label}参数较为特别，如果 Git 的分支和标签名包含“/”，那么{label}参数在 HTTP 的 URL 中应该使用“(_)”来替代，以避免改变了 URI 含义，指向到其他的 URI 资源。

当我们使用 Git 作为配置中心来存储各个微服务应用配置文件的时候，该功能会变得非常有用，通过在 URI 中使用占位符可以帮助我们规划和实现通用的仓库配置。例如，我们可以对微服务应用做如下规划。

- **代码库**：使用服务名作为 Git 仓库名称，比如会员服务的代码库 http://git.oschina.net/didispace/member-service。
- **配置库**：使用服务名加上-config 后缀作为 Git 仓库名称，比如上面会员服务对应的配置库地址位置 http://git.oschina.net/didispace/member-service-config。

这时，我们就可以使用 spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/{application}-config 配置，来同时匹配多个不同服务的配置仓库。

配置多个仓库

Config Server 除了可以通过 application 和 profile 模式来匹配配置仓库之外，还支持通过带有通配符的表达式来匹配，以实现更为复杂的配置需求。并且当我们有多个匹配规则的时候，还可以用逗号来分割多个{application}/{profile}配置规则，比如：

```
spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/config-repo

spring.cloud.config.server.git.repos.dev.pattern=dev/*
spring.cloud.config.server.git.repos.dev.uri=file://home/git/config-repo

spring.cloud.config.server.git.repos.test=http://git.oschina.net/test/config-repo

spring.cloud.config.server.git.repos.prod.pattern=prod/pp*,online/oo*
spring.cloud.config.server.git.repos.prod.uri=http://git.oschina.net/prod/confi
g-repo
```

上述配置内容通过 spring.cloud.config.server.git.uri 属性，指定了一个默认的仓库位置，当使用{application}/{profile}模式未能匹配到合适的仓库时，就将在该默认仓库位置下获取配置信息。除此之外，还配置了三个仓库，分别是 dev、test、prod。其中，dev 仓库匹配 dev/*的模式，所以无论 profile 是什么，它都能匹配 application 名称为 dev 的应用。并且我们可以注意到，它存储的配置文件位置还采用

了 Config Server 的本地文件系统中的内容。对于此配置，我们可以通过访问 http://localhost:7001/dev/profile 的请求来验证到该仓库的配置内容，其中 profile 可以为任意值。而 test 和 prod 仓库均使用了 Git 仓库的存储，并且 test 仓库未配置匹配规则，所以它只匹配 application 名为 test 的应用；prod 仓库则需要匹配 application 为 prod 并且 profile 为 pp 开头，或者 application 为 online 并且 profile 为 oo 开头的应用和环境。

当配置多个仓库的时候，Config Server 在启动时会直接克隆第一个仓库的配置库，其他的配置库只有在请求时才会克隆到本地，所以对于仓库的排列可以根据配置内容的重要程度有所区分。另外，如果表达式是以通配符开始的，那么需要使用引号将配置内容引起来。

子目录存储

除了支持占位符配置、多仓库配置之外，Config Server 还可以将配置文件定位到 Git 仓库的子目录中。有心的读者，或许还能记得在快速入门中，我们除了配置 spring.cloud.config.server.git.uri 属性之外，还配置了另外一个参数：spring.cloud.config.server.git.searchPaths，通过该参数我们实现了在 http://git.oschina.net/didispace/SpringCloud-Learning/仓库的 spring_cloud_in_action/config-repo 子目录下实现配置的存储。

对于 spring.cloud.config.server.git.searchPaths 参数的配置也支持使用{application}、{profile}和{label}占位符，比如：

```
spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/SpringCloud
-Learning/
spring.cloud.config.server.git.searchPaths={application}
```

通过上面的配置，我们可以实现在 http://git.oschina.net/didispace/SpringCloud-Learning/仓库中，一个应用一个目录的效果。

访问权限

Config Server 在访问 Git 仓库的时候，若采用 HTTP 的方式进行认证，那么我们需要增加 username 和 password 属性来配置账户（快速入门中也是如此实现），比如：

```
spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/application
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
```

若不使用 HTTP 的认证方式，我们也可采用 SSH 的方式，通过生成 Key 并在 Git 仓库中进行配置匹配以实现访问。

SVN 配置仓库

Config Server 除了支持 Git 仓库之外，也能使用 SVN 仓库，只需要做如下配置。

- 在 pom.xml 中引入 SVN 的依赖配置，让 Config Server 拥有读取 SVN 内容的能力：

```
<dependency>
    <groupId>org.tmatesoft.svnkit</groupId>
    <artifactId>svnkit</artifactId>
    <version>1.8.10</version>
</dependency>
```

- 在 application.properties 中使用 SVN 的配置属性来指定 SVN 服务器的位置，以及访问的账户名与密码：

```
spring.cloud.config.server.svn.uri=svn://localhost:443/didispace/config-repo
spring.cloud.config.server.svn.username=username
spring.cloud.config.server.svn.password=password
```

通过上面的配置修改，Config Server 就可以使用 SVN 作为仓库来存储配置文件了，而对于客户端来说，这个过程是透明的，所以不需要做任何变动。

本地仓库

在使用了 Git 或 SVN 仓库之后，文件都会在 Config Server 的本地文件系统中存储一份，这些文件默认会被存储于以 config-repo 为前缀的临时目录中，比如名为 /tmp/config-repo-<随机数>的目录。由于其随机性以及临时目录的特性，可能会有一些不可预知的后果，为了避免将来可能会出现的问题，最好的办法就是指定一个固定的位置来存储这些重要信息。我们只需要通过 spring.cloud.config.server.git.basedir 或 spring.cloud.config.server.svn.basedir 来配置一个我们准备好的目录即可。

本地文件系统

Spring Cloud Config 也提供了一种不使用 Git 仓库或 SVN 仓库的存储方式，而是使用本地文件系统的存储方式来保存配置信息。实现方式也非常简单，我们只需要设置属性 spring.profiles.active=native，Config Server 会默认从应用的 src/main/resource 目录下搜索配置文件。如果需要指定搜索配置文件的路径，我们可以通过 spring.cloud.config.server.native.searchLocations 属性来指定具体的配置文件位置。

虽然 Spring Cloud Config 提供了这样的功能，但是为了支持更好的内容管理和版本控制等强大功能，还是推荐使用 Git 仓库的方式。

健康监测

当使用占位符来配置 URI 的时候，很有可能在控制台中出现类似这样的警告信息（以 spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/{application}-config 配置为例）：

```
    o.s.c.c.s.e.JGitEnvironmentRepository    : Could not fetch remote for master remote:
http://git.oschina.net/didispace/app-config
```

而引起该警告的原因是由于 Spring Cloud Config 的服务端为 spring-boot-actuator 模块的/health 端点实现了对应的健康检测器：org.springframework.cloud.config.server.config.ConfigServerHealthIndicator。在该检测器中，默认构建了一个 application 为 app 的仓库。而根据之前的配置规则：{application}-config.git，该检测器会不断地检查 http://git.oschina.net/didispace/app-config 仓库是否可以连通。此时，我们可以访问配置中心的/health 端点来查看它的健康状态，具体返回信息如下：

```
    "configServer": {
        "status": "DOWN",
        "repository": {
            "application": "app",
            "profiles": "default"
        },
        "error": "org.springframework.cloud.config.server.environment.NoSuchLabelException:
No such label: master"
    }
```

从/health 端点的返回信息中，我们可以看到，由于无法连通 http://git.oschina.net/didispace/app-config 仓库，使得配置中心的可用状态是 DOWN。虽然我们依然可以通过 URI 的方式访问该配置中心，但是当将配置中心服务化使用的时候，该状态将影响它的服务可用性判断。所以，我们需要了解它的健康检测配置，并让它的健康检测器正常工作起来。

根据默认配置规则，我们可以直接在 Git 仓库中创建一个名为 app-config 的配置库，让健康检测器能够访问到它。另外，也可以配置一个实际存在的仓库来进行连通检测，比如下面的配置，它实现了通过连接 check-repo-config 仓库来进行健康监测：

```
    spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/{applicatio
n}-config
    spring.cloud.config.server.git.username=username
    spring.cloud.config.server.git.username=password
    spring.cloud.config.server.health.repositories.check.name=check-repo
    spring.cloud.config.server.health.repositories.check.label=master
    spring.cloud.config.server.health.repositories.check.profiles=default
```

由于健康检测的 repositories 是个 Map 对象，所以实际使用时我们可以配置多个。而每个配置中包含了与定位仓库地址时类似的三个元素。

- name：应用名。
- label：分支名。
- profiles：环境名。

如果我们不想使用该健康检测器，也可以通过使用 spring.cloud.config.server.health.enabled=false 参数设置来关闭它。

属性覆盖

Config Server 还有一个“属性覆盖”的特性，它可以让开发人员为所有的应用提供配置属性，只需要通过 spring.cloud.config.server.overrides 属性来设置键值对的参数，这些参数会以 Map 的方式加载到客户端的配置中。比如：

```
spring.cloud.config.server.overrides.name=didi
spring.cloud.config.server.overrides.from=shanghai
```

通过该属性配置的参数，不会被 Spring Cloud 的客户端修改，并且 Spring Cloud 客户端从 Config Server 中获取配置信息时，都会取得这些配置信息。利用该特性可以方便地为 Spring Cloud 应用配置一些共同属性或是默认属性。当然，这些属性并非强制的，我们可以通过改变客户端中更高优先级的配置方式（比如，配置环境变量或是系统属性），来选择是否使用 Config Server 提供的默认值。

安全保护

由于配置中心存储的内容比较敏感，做一定的安全处理是必需的。为配置中心实现安全保护的方式有很多，比如物理网络限制、OAuth2 授权等。不过，由于我们的微服务应用和配置中心都构建于 Spring Boot 基础上，所以与 Spring Security 结合使用会更加方便。

我们只需要在配置中心的 pom.xml 中加入 spring-boot-starter-security 依赖，不需要做任何其他改动就能实现对配置中心访问的安全保护。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

默认情况下，我们可以获得一个名为 user 的用户，并且在配置中心启动的时候，在日志中打印出该用户的随机密码，具体如下：

```
INFO 22028 --- [          main] b.a.s.AuthenticationManagerConfiguration : Using
default security password: 1a32a848-da0c-4590-9c58-e860be8c50dd
```

大多数情况下，我们并不会使用随机生成密码的机制。我们可以在配置文件中指定用户和密码，比如：

```
security.user.name=user
security.user.password=37cc5635-559b-4e6f-b633-7e932b813f73
```

由于我们已经为 config-server 设置了安全保护，如果这时候连接到配置中心的客户端中没有设置对应的安全信息，在获取配置信息时会返回 401 错误。所以，需要通过配置的方式在客户端中加入安全信息来通过校验，比如：

```
spring.cloud.config.username=user
spring.cloud.config.password=37cc5635-559b-4e6f-b633-7e932b813f73
```

加密解密

在微服务架构中，我们通常会采用 DevOps 的组织方式来降低因团队间沟通造成的巨大成本，以加速微服务应用的交付能力。这就使得原本由运维团队控制的线上信息将交由微服务所属组织的成员自行维护，其中将会包括大量的敏感信息，比如数据库的账户与密码等。显然，如果我们直接将敏感信息以明文的方式存储于微服务应用的配置文件中是非常危险的。针对这个问题，Spring Cloud Config 提供了对属性进行加密解密的功能，以保护配置文件中的信息安全。比如下面的例子：

```
spring.datasource.username=didi
spring.datasource.password={cipher}dba6505baa81d78bd08799d8d4429de499bd4c2053c0
5f029e7cfbf143695f5b
```

在 Spring Cloud Config 中通过在属性值前使用{cipher}前缀来标注该内容是一个加密值，当微服务客户端加载配置时，配置中心会自动为带有{cipher}前缀的值进行解密。通过该机制的实现，运维团队就可以放心地将线上信息的加密资源给到微服务团队，而不用担心这些敏感信息遭到泄露了。下面我们来具体介绍如何在配置中心使用该项功能。

使用前提

在使用 Spring Cloud Config 的加密解密功能时，有一个必要的前提需要我们注意。为了启用该功能，我们需要在配置中心的运行环境中安装不限长度的 JCE 版本（Unlimited Strength Java Cryptography Extension）。虽然，JCE 功能在 JRE 中自带，但是默认使用的是有长度限制的版本。我们可以从 Oracle 的官方网站下载到它，它是一个压缩包，解压后可以看到下面三个文件：

```
README.txt
local_policy.jar
US_export_policy.jar
```

我们需要将 local_policy.jar 和 US_export_policy.jar 两个文件复制到

$JAVA_HOME/jre/lib/security 目录下，覆盖原来的默认内容。到这里，加密解密的准备工作就完成了。

相关端点

在完成了 JCE 的安装后，可以尝试启动配置中心。在控制台中，将会输出一些配置中心特有的端点，主要包括如下几个。

- /encrypt/status：查看加密功能状态的端点。
- /key：查看密钥的端点。
- /encrypt：对请求的 body 内容进行加密的端点。
- /decrypt：对请求的 body 内容进行解密的端点。

可以尝试通过 GET 请求访问/encrypt/status 端点，我们将得到如下内容：

```
{
  "description": "No key was installed for encryption service",
  "status": "NO_KEY"
}
```

该返回信息说明当前配置中心的加密功能还不能使用，因为没有为加密服务配置对应的密钥。

配置密钥

我们可以通过 encrypt.key 属性在配置文件中直接指定密钥信息（对称性密钥），比如：

```
encrypt.key=didispace
```

加入上述配置信息后，重启配置中心，再访问/encrypt/status 端点，我们将得到如下内容：

```
{
  "status": "OK"
}
```

此时，我们配置中心的加密解密功能就已经可以使用了，不妨尝试访问一下/encrypt 和/decrypt 端点来使用加密和解密的功能。注意，这两个端点都是 POST 请求，加密和解密信息需要通过请求体来发送。比如，以 curl 命令为例，我们可以通过下面的方式调用加密与解密端点：

```
    $ curl localhost:7001/encrypt -d didispace
    3c70a809bfa24ab88bcb5e1df51cb9e4dd4b8fec88301eb7a18177f1769c849ae9c9f29400c9204
80be2c99406ae28c7

    $ curl localhost:7001/decrypt -d
3c70a809bfa24ab88bcb5e1df51cb9e4dd4b8fec88301eb7a18177f1769c849ae9c9f29400c920480be
```

```
2c99406ae28c7
    didispace
```

这里，我们通过配置 encrypt.key 参数来指定密钥的实现方式采用了对称性加密。这种方式实现起来比较简单，只需要配置一个参数即可。另外，我们也可以使用环境变量 ENCRYPT_KEY 来进行配置，让密钥信息外部化存储。

非对称加密

Spring Cloud Config 的配置中心不仅可以使用对称性加密，也可以使用非对称性加密（比如 RSA 密钥对）。虽然非对称性加密的密钥生成与配置相对复杂一些，但是它具有更高的安全性。下面，我们来具体介绍一下如何使用非对称加密。

首先，需要通过 keytool 工具来生成密钥对。keytool 是 JDK 中的一个密钥和证书管理工具。它使用户能够管理自己的公钥/私钥对及相关证书，用于（通过数字签名）自我认证（用户向其他用户/服务认证自己）或数据完整性以及认证服务。在 JDK 1.4 以后的版本中都包含了这一工具，它的位置在%JAVA_HOME%\bin\keytool.exe。

生成密钥的具体命令如下所示：

```
    $ keytool -genkeypair -alias config-server -keyalg RSA -keystore
config-server.keystore
    输入密钥库口令:
    再次输入新口令:
    您的名字与姓氏是什么?
      [Unknown]:  zhaiyongchao
    您的组织单位名称是什么?
      [Unknown]:  company
    您的组织名称是什么?
      [Unknown]:  organization
    您所在的城市或区域名称是什么?
      [Unknown]:  city
    您所在的省/市/自治区名称是什么?
      [Unknown]:  province
    该单位的双字母国家/地区代码是什么?
      [Unknown]:  china
    CN=zhaiyongchao, OU=company, O=organization, L=city, ST=province, C=china 是否正确?
      [否]:  y

    输入 <config-server> 的密钥口令
            (如果和密钥库口令相同, 按回车):
    再次输入新口令:
```

另外，如果不想逐步输入那些提示信息，可以使用-dname 来直接指定，而密钥库口令与密钥口令可使用-storepass 和-keypass 来直接指定。所以，我们可以通过下面的命令直接创建出与上述命令一样的密钥库：

```
$ keytool -genkeypair -alias config-server -keyalg RSA \
  -dname "CN=zhaiyongchao, OU=company, O=organization, L=city, ST=province,
C=china" \
  -keypass 222222 \
  -keystore config-server.keystore \
  -storepass 111111 \
```

默认情况下，使用上述命令创建的密钥只有 90 天有效期。如果想要调整它的有效期，可以通过增加-validity 参数来实现，比如我们可以通过下面的命令，让密钥的有效期延长到一年：

```
$ keytool -genkeypair -alias config-server -keyalg RSA \
  -dname "CN=zhaiyongchao, OU=company, O=organization, L=city, ST=province,
C=china" \
  -keypass 222222 \
  -keystore config-server.keystore \
  -storepass 111111 \
  -validity 365 \
```

上述的三种命令生成方式，最终都会在命令的当前执行目录下生成一个 config-server.keystore 文件。下面，我们需要将它保存在配置中心的文件系统中的某个位置，比如放在当前的用户目录下，然后在配置中心中加入相关的配置信息：

```
encrypt.key-store.location=file://${user.home}/config-server.keystore
encrypt.key-store.alias=config-server
encrypt.key-store.password=111111
encrypt.key-store.secret=222222
```

如果我们将 config-server.keystore 放在配置中心的 src/main/resource 目录下，也可以直接这样配置：encrypt.key-store.location=config-server.keystore。另外，非对称加密的配置信息也可以通过环境变量的方式进行配置，它们对应的具体变量名如下：

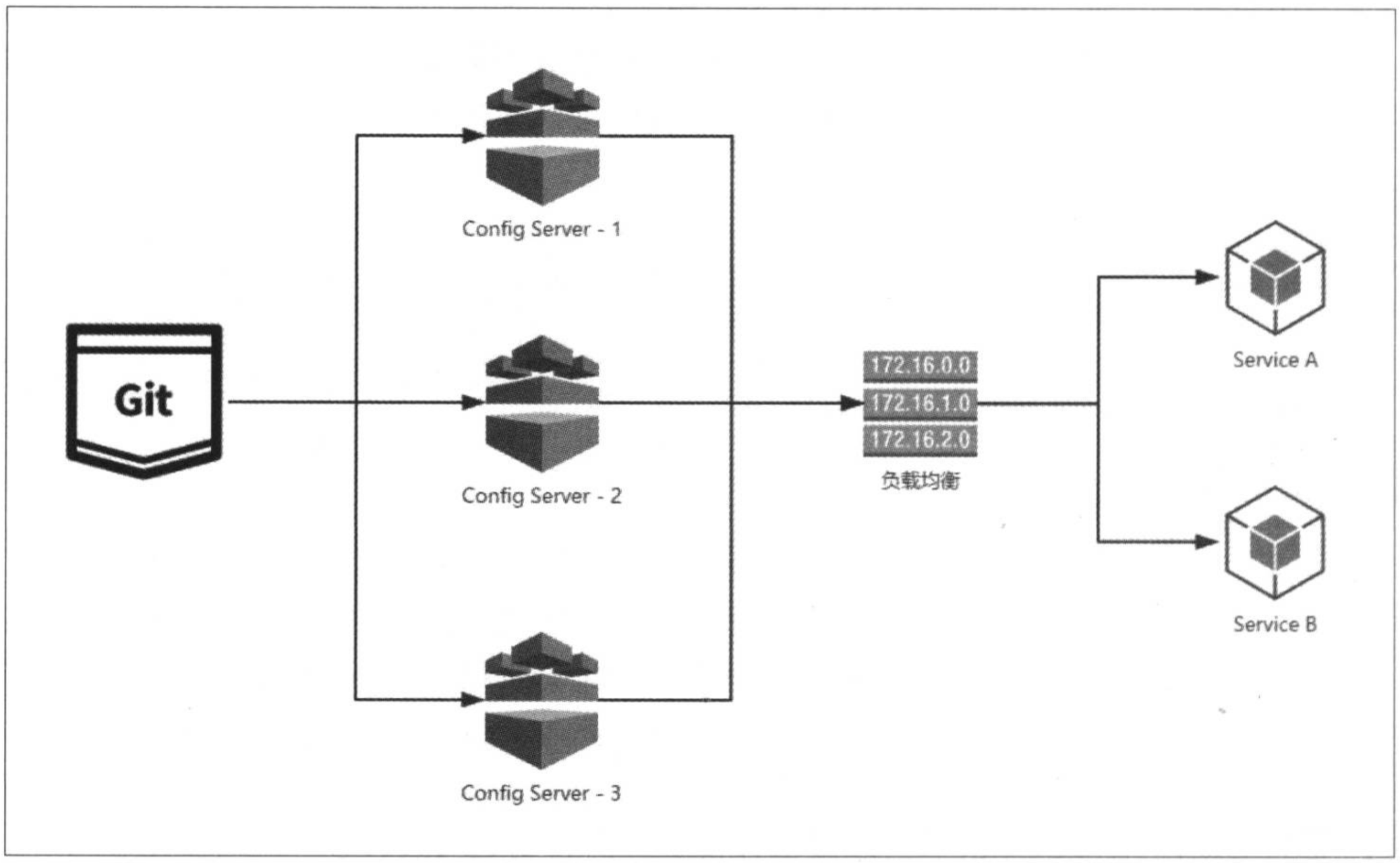
```
ENCRYPT_KEY_STORE_LOCATION
ENCRYPT_KEY_STORE_ALIAS
ENCRYPT_KEY_STORE_PASSWORD
ENCRYPT_KEY_STORE_SECRET
```

通过环境变量来配置密钥库相关信息可以获得更好的安全性，所以我们将敏感的口令信息存储在配置中心的环境变量中是一种不错的选择。

高可用配置

当要将配置中心部署到生产环境中时，与服务注册中心一样，我们也希望它是一个高可用的应用。Spring Cloud Config 实现服务端的高可用非常简单，主要有以下两种方式。

- **传统模式**：不需要为这些服务端做任何额外的配置，只需要遵守一个配置规则，将所有的 Config Server 都指向同一个 Git 仓库，这样所有的配置内容就通过统一的共享文件系统来维护。而客户端在指定 Config Server 位置时，只需要配置 Config Server 上层的负载均衡设备地址即可，就如下图所示的结构。



- **服务模式**：除了上面这种传统的实现模式之外，我们也可以将 Config Server 作为一个普通的微服务应用，纳入 Eureka 的服务治理体系中。这样我们的微服务应用就可以通过配置中心的服务名来获取配置信息，这种方式比起传统的实现模式来说更加有利于维护，因为对于服务端的负载均衡配置和客户端的配置中心指定都通过服务治理机制一并解决了，既实现了高可用，也实现了自维护。由于这部分的实现需要客户端的配合，具体示例读者可详细阅读“客户端详解”一节中的“服务化配置中心”小节。

客户端详解

在学习了关于 Spring Cloud Config 服务端的大量配置和使用细节之后，接下来我们将

通过本节内容继续学习 Spring Cloud Config 客户端的使用与配置。

URI 指定配置中心

Spring Cloud Config 的客户端在启动的时候，默认会从工程的 classpath 中加载配置信息并启动应用。只有当我们配置 `spring.cloud.config.uri` 的时候，客户端应用才会尝试连接 Spring Cloud Config 的服务端来获取远程配置信息并初始化 Spring 环境配置。同时，我们必须将该参数配置在 `bootstrap.properties`、环境变量或是其他优先级高于应用 Jar 包内的配置信息中，才能正确加载到远程配置。若不指定 `spring.cloud.config.uri` 参数的话，Spring Cloud Config 的客户端会默认尝试连接 `http://localhost:8888`。

在之前的快速入门示例中，我们就是以这种方式实现的，就如下面的配置，其中 `spring.application.name` 和 `spring.cloud.config.profile` 用于定位配置信息。

```
spring.application.name=didispace
spring.cloud.config.profile=dev

spring.cloud.config.uri=http://localhost:7001/
```

服务化配置中心

在第 3 章中，我们已经学会了如何构建服务注册中心、如何发现与注册服务。那么 Config Server 是否也能以服务的方式注册到服务中心，并被其他应用所发现来实现配置信息的获取呢？答案是肯定的。在 Spring Cloud 中，我们也可以把 Config Server 视为微服务架构中与其他业务服务一样的一个基本单元。

下面，我们就来详细介绍如何将 Config Server 注册到服务中心，并通过服务发现来访问 Config Server 并获取 Git 仓库中的配置信息。下面的内容将基于快速入门中实现的 `config-server` 和 `config-client` 工程来进行改造实现。

服务端配置

- 在 `config-server` 的 `pom.xml` 中增加 `spring-cloud-starter-eureka` 依赖，以实现将分布式配置中心加入 Eureka 的服务治理体系。

```
……
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
    </dependency>
    <dependency>
```

```
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
</dependencies>
……
```

- 在 application.properties 中配置参数 eureka.client.serviceUrl.defaultZone 以指定服务注册中心的位置，详细内容如下：

```
spring.application.name=config-server
server.port=7001
# 配置服务注册中心
eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
# Git 管理配置
spring.cloud.config.server.git.uri=http://git.oschina.net/didispace/SpringCloud
-Learning/
spring.cloud.config.server.git.searchPaths=spring_cloud_in_action/config-repo
spring.cloud.config.server.git.username=username
spring.cloud.config.server.git.password=password
```

- 在应用主类中，新增@EnableDiscoveryClient 注解，用来将 config-server 注册到上面配置的服务注册中心上去。

```
@EnableDiscoveryClient
@EnableConfigServer
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

- 启动该应用，并访问 http://localhost:1111/，可以在 Eureka Server 的信息面板中看到 config-server 已经被注册了。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONFIG-SERVER	n/a (1)	(1)	UP (1) - PC-201602152056:config-server:7001

客户端配置

- 在 config-client 的 pom.xml 中新增 spring-cloud-starter-eureka 依

赖，以实现客户端发现 config-server 服务，具体配置如下：

```
……
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-config</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-eureka</artifactId>
    </dependency>
</dependencies>
……
```

- 在 bootstrap.properties 中，按如下配置：

```
spring.application.name=didispace
server.port=7002

eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/

spring.cloud.config.discovery.enabled=true
spring.cloud.config.discovery.serviceId=config-server
spring.cloud.config.profile=dev
```

其中，通过 eureka.client.serviceUrl.defaultZone 参数指定服务注册中心，用于服务的注册与发现；再将 spring.cloud.config.discovery.enabled 参数设置为 true，开启通过服务来访问 Config Server 的功能；最后利用 spring.cloud.config.discovery.serviceId 参数来指定 Config Server 注册的服务名。这里的 spring.application.name 和 spring.cloud.config.profile 如之前通过 URI 的方式访问的时候一样，用来定位 Git 中的资源。

- 在应用主类中，增加@EnableDiscoveryClient 注解，用来发现 config-server 服务，利用其来加载应用配置：

```
@EnableDiscoveryClient
@SpringBootApplication
public class Application {

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

```
        new SpringApplicationBuilder(Application.class).web(true).run(args);
    }

}
```

- 沿用之前我们创建的 Controller 来加载 Git 中的配置信息：

```
@RefreshScope
@RestController
public class TestController {

    @Value("${from}")
    private String from;

    @RequestMapping("/from")
    public String from() {
        return this.from;
    }

}
```

- 完成上述配置之后，我们启动该客户端应用。若启动成功，访问 http://localhost:1111/，可以在 Eureka Server 的信息面板中看到该应用已经被注册成功。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CONFIG-SERVER	n/a (1)	(1)	UP (1) - PC-201602152056:config-server:7001
DIDISPACE	n/a (1)	(1)	UP (1) - PC-201602152056:didispace:7002

- 访问客户端应用提供的服务 http://localhost:7002/from，此时，我们会返回在 Git 仓库中 didispace-dev.properties 文件中配置的 from 属性内容："git-dev-1.0"。

失败快速响应与重试

Spring Cloud Config 的客户端会预先加载很多其他信息，然后再开始连接 Config Server 进行属性的注入。当我们构建的应用较为复杂的时候，可能在连接 Config Server 之前花费较长的启动时间，而在一些特殊场景下，我们又希望可以快速知道当前应用是否能顺利地从 Config Server 获取到配置信息，这对在初期构建调试环境时，可以减少很多等待启动的时间。要实现客户端优先判断 Config Server 获取是否正常，并快速响应失败内容，只需在 bootstrap.properties 中配置参数 spring.cloud.config.failFast= true 即可。

我们可以实验一下，在未配置该参数前，不启动 Config Server，直接启动客户端应用，可以获得下面的报错信息。同时，在报错之前，可以看到客户端应用已经加载了很多内容，比如 Controller 的请求等。

```
org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'scopedTarget.testController': Injection of autowired dependencies failed;
```

加上 spring.cloud.config.failFast=true 参数之后，再启动客户端应用，可以获得下面的报错信息，并且前置的加载内容少了很多，这样通过该参数有效避免了当 Config Server 配置有误时，不需要多等待前置的一些加载时间，实现了快速返回失败信息。

```
java.lang.IllegalStateException: Could not locate PropertySource and the fail fast property is set, failing
```

上面，我们演示了当 Config Server 宕机或是客户端配置不正确导致连接不到而启动失败的情况，快速响应的配置可以发挥比较好的效果。但是，若只是因为网络波动等其他间歇性原因导致的问题，直接启动失败似乎代价有些高。所以，Config 客户端还提供了自动重试的功能，在开启重试功能前，先确保已经配置了 spring.cloud.config.failFast=true，再进行下面的操作。

- 在客户端的 pom.xml 中增加 spring-retry 和 spring-boot-starter-aop 依赖，具体如下：

```
<dependencies>
……
    <dependency>
        <groupId>org.springframework.retry</groupId>
        <artifactId>spring-retry</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-aop</artifactId>
    </dependency>
……
</dependencies>
```

- 不需要再做其他任何配置，启动客户端应用，在控制台中可以看到如下内容。客户端在连接 Config Server 失败之后，会继续尝试，直到第 6 次失败后，才返回错误信息。通过这样的重试机制，可以避免一些间歇性问题引起的失败导致客户端应用无法启动的情况。

```
c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at: http://PC-201602152056:7001/
c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at: http://PC-201602152056:7001/
```

```
    c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at:
http://PC-201602152056:7001/
    c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at:
http://PC-201602152056:7001/
    c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at:
http://PC-201602152056:7001/
    c.c.c.ConfigServicePropertySourceLocator : Fetching config from server at:
http://PC-201602152056:7001/
    o.s.boot.SpringApplication               : Application startup failed
    java.lang.IllegalStateException: Could not locate PropertySource and the fail fast
property is set, failing
```

若对默认的最大重试次数和重试间隔等设置不满意，还可以通过下面的参数进行调整。

- spring.cloud.config.retry.multiplier：初始重试间隔时间（单位为毫秒），默认为 1000 毫秒。
- spring.cloud.config.retry.initial-interval：下一间隔的乘数，默认为 1.1，所以当最初间隔是 1000 毫秒时，下一次失败后的间隔为 1100 毫秒。
- spring.cloud.config.retry.max-interval：最大间隔时间，默认为 2000 毫秒。
- spring.cloud.config.retry.max-attempts：最大重试次数，默认为 6 次。

获取远程配置

在入门示例中，我们对{application}、{profile}、{label}这些参数已经有了一定的了解。在 Git 仓库中，一个形如{application}-{profile}.properties 或{application}-{profile}.yml 的配置文件，通过 URL 请求和客户端配置的访问对应可以总结如下。

- 通过向 Config Server 发送 GET 请求以直接的方式获取，可用下面的链接形式。
  - 不带{label}分支信息，默认访问 master 分支，可使用：
    - /{application}-{profile}.yml
    - /{application}-{profile}.properties
  - 带{label}分支信息，可使用：
    - /{label}/{application}-{profile}.yml
    - /{application}/{profile}[/{label}]
    - /{label}/{application}-{profile}.properties

- 通过客户端配置方式加载的内容如下所示。
  - spring.application.name：对应配置文件中的{application}内容。
  - spring.cloud.config.profile：对应配置文件中{profile}内容。
  - spring.cloud.config.label：对应分支内容，如不配置，默认为master。

动态刷新配置

有时候，我们需要对配置内容做一些实时更新，那么 Spring Cloud Config 是否可以实现呢？答案显然是可以的。下面，我们以快速入门中的示例作为基础，看看如何进行改造来实现配置内容的实时更新。

首先，回顾一下，当前我们已经实现了哪些内容。

- config-repo：定义在 Git 仓库中的一个目录，其中存储了应用名为 didispace 的多环境配置文件，配置文件中有一个 from 参数。
- config-server：配置了 Git 仓库的服务端。
- config-client：指定了 config-server 为配置中心的客户端，应用名为 didispace，用来访问配置服务器以获取配置信息。该应用中提供了一个/from 接口，它会获取 config-repo/didispace-dev.properties 中的 from 属性返回。

在改造程序之前，我们先将 config-server 和 config-client 都启动起来，并访问客户端提供的 REST 接口 http://localhost:7002/from 来获取配置信息，获得的返回内容为 git-dev-1.0。接着，我们可以尝试使用 Git 工具修改当前配置的内容，比如，将 config-repo/didispace-dev.properties 中的 from 的值从 from=git-dev-1.0 修改为 from=git-dev-2.0，再访问 http://localhost:7002/from，可以看到其返回内容还是 git-dev-1.0。

接下来，我们将在 config-client 端做一些改造以实现配置信息的动态刷新。

- 在 config-client 的 pom.xml 中新增 spring-boot-starter-actuator 监控模块。其中包含了/refresh 端点的实现，该端点将用于实现客户端应用配置信息的重新获取与刷新。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

- 重新启动 config-client，访问一次 http://localhost:7002/from，可以看到当前的配置值。
- 修改 Git 仓库 config-repo/didispace-dev.properties 文件中 from 的值。
- 再访问一次 http://localhost:7002/from，可以看到配置值没有改变。
- 通过 POST 请求发送到 http://localhost:7002/refresh，我们可以看到返回内容如下，代表 from 参数的配置内容被更新了：

```
[
  "from"
]
```

- 再访问一次 http://localhost:7002/from，可以看到配置值已经是更新后的值了。

通过上面的介绍，大家不难想到，该功能还可以同 Git 仓库的 Web Hook 功能进行关联，当有 Git 提交变化时，就给对应的配置主机发送/refresh 请求来实现配置信息的实时更新。但是，当我们的系统发展壮大之后，维护这样的刷新清单也将成为一个非常大的负担，而且很容易犯错，那么有什么办法可以解决这个复杂度呢？后续我们将介绍如何通过 Spring Cloud Bus 来实现以消息总线的方式进行配置变更的通知，并完成集群上的批量配置更新。

第 9 章

消息总线：Spring Cloud Bus

在微服务架构的系统中，我们通常会使用轻量级的消息代理来构建一个共用的消息主题让系统中所有微服务实例都连接上来，由于该主题中产生的消息会被所有实例监听和消费，所以我们称它为消息总线。在总线上的各个实例都可以方便地广播一些需要让其他连接在该主题上的实例都知道的消息，例如配置信息的变更或者其他一些管理操作等。

由于消息总线在微服务架构系统中被广泛使用，所以它同配置中心一样，几乎是微服务架构中的必备组件。Spring Cloud 作为微服务架构综合性的解决方案，对此自然也有自己的实现，这就是本章我们将要具体介绍的 Spring Cloud Bus。通过使用 Spring Cloud Bus，可以非常容易地搭建起消息总线，同时实现了一些消息总线中的常用功能，比如，配合 Spring Cloud Config 实现微服务应用配置信息的动态更新等。

在本章中，我们将从消息代理的基础开始，由浅入深地介绍如何使用 Spring Cloud Bus 构建微服务架构中的消息总线。

消息代理

消息代理（Message Broker）是一种消息验证、传输、路由的架构模式。它在应用程序之间起到通信调度并最小化应用之间的依赖的作用，使得应用程序可以高效地解耦通信过程。消息代理是一个中间件产品，它的核心是一个消息的路由程序，用来实现接收和分发消息，并根据设定好的消息处理流来转发给正确的应用。它包括独立的通信和消息传递协议，能够实现组织内部和组织间的网络通信。设计代理的目的就是为了能够从应用程序中传入消息，并执行一些特别的操作，下面这些是在企业应用中，我们经常需要使用消息代理的场景：

- 将消息路由到一个或多个目的地。

- 消息转化为其他的表现方式。
- 执行消息的聚集、消息的分解，并将结果发送到它们的目的地，然后重新组合响应返回给消息用户。
- 调用 Web 服务来检索数据。
- 响应事件或错误。
- 使用发布-订阅模式来提供内容或基于主题的消息路由。

目前已经有非常多的开源产品可以供大家使用，比如：

- ActiveMQ
- Kafka
- RabbitMQ
- RocketMQ
- ...

当前版本的 Spring Cloud Bus 仅支持两款中间件产品：RabbitMQ 和 Kafka。在下面的章节中，我们将分别介绍如何使用这两款消息中间件与 Spring Cloud Bus 配合实现消息总线。

RabbitMQ 实现消息总线

RabbitMQ 是实现了高级消息队列协议（AMQP）的开源消息代理软件，也称为面向消息的中间件。RabbitMQ 服务器是用高性能、可伸缩而闻名的 Erlang 语言编写而成的，其集群和故障转移是构建在开放电信平台框架上的。

AMQP 是 Advanced Message Queuing Protocol 的简称，它是一个面向消息中间件的开放式标准应用层协议。它定义了以下这些特性：

- 消息方向
- 消息队列
- 消息路由（包括点到点和发布–订阅模式）
- 可靠性
- 安全性

AMQP 要求消息的提供者和客户端接收者的行为要实现对不同供应商可以用相同的方式（比如 SMTP、HTTP、FTP 等）进行互相操作。在以往的中间件标准中，主要还是建立在 API 级别，比如 JMS，集中于通过不同的中间件实现来建立标准化的程序间的互操作性，而不是在多个中间件产品间实现互操作性。

> AMQP 与 JMS 不同，JMS 定义了一个 API 和一组消息收发必须实现的行为，而 AMQP 是一个线路级协议。线路级协议描述的是通过网络发送的数据传输格式。因此，任何符合该数据格式的消息发送和接收工具都能互相兼容和进行操作，这样就能轻易实现跨技术平台的架构方案。

RabbitMQ 以 AMQP 协议实现，所以它可以支持多种操作系统、多种编程语言，几乎可以覆盖所有主流的企业级技术平台。在微服务架构消息中间件的选型中，它是一个非常适合且优秀的选择。因此，在 Spring Cloud Bus 中包含了对 Rabbit 的自动化默认配置，在下面的章节中，我们将先从 RabbitMQ 的基础安装和使用开始，循序渐进地学习如何与 Spring Cloud Bus 进行整合实现消息总线。

基本概念

在开始具体实践之前，我们先介绍一些关于 RabbitMQ 的基本概念，为后续的讲解做一些必要铺垫（如果对于 RabbitMQ 已经很熟悉的读者可以跳过本节，直接从“快速入门”一节开始阅读）。

- Broker：可以理解为消息队列服务器的实体，它是一个中间件应用，负责接收消息生产者的消息，然后将消息发送至消息接收者或者其他的 Broker。
- Exchange：消息交换机，是消息第一个到达的地方，消息通过它指定的路由规则，分发到不同的消息队列中去。
- Queue：消息队列，消息通过发送和路由之后最终到达的地方，到达 Queue 的消息即进入逻辑上等待消费的状态。每个消息都会被发送到一个或多个队列。
- Binding：绑定，它的作用就是把 Exchange 和 Queue 按照路由规则绑定起来，也就是 Exchange 和 Queue 之间的虚拟连接。
- Routing Key：路由关键字，Exchange 根据这个关键字进行消息投递。
- Virtual host：虚拟主机，它是对 Broker 的虚拟划分，将消费者、生产者和它们依赖的 AMQP 相关结构进行隔离，一般都是为了安全考虑。比如，我们可以在一个 Broker 中设置多个虚拟主机，对不同用户进行权限的分离。
- Connection：连接，代表生产者、消费者、Broker 之间进行通信的物理网络。
- Channel：消息通道，用于连接生产者和消费者的逻辑结构。在客户端的每个连接里，可建立多个 Channel，每个 Channel 代表一个会话任务，通过 Channel 可以隔离同一连接中的不同交互内容。
- Producer：消息生产者，制造消息并发送消息的程序。
- Consumer：消息消费者，接收消息并处理消息的程序。

消息投递到队列的整个过程大致如下：

1. 客户端连接到消息队列服务器，打开一个 Channel。
2. 客户端声明一个 Exchange，并设置相关属性。
3. 客户端声明一个 Queue，并设置相关属性。
4. 客户端使用 Routing Key，在 Exchange 和 Queue 之间建立好绑定关系。
5. 客户端投递消息到 Exchange。
6. Exchange 接收到消息后，根据消息的 Key 和已经设置的 Binding，进行消息路由，将消息投递到一个或多个 Queue 里。

Exchange 也有几种类型。

1. Direct 交换机：完全根据 Key 进行投递。比如，绑定时设置了 Routing Key 为 `abc`，那么客户端提交的消息，只有设置了 Key 为 `abc` 的才会被投递到队列。
2. Topic 交换机：对 Key 进行模式匹配后进行投递，可以使用符号#匹配一个或多个词，符号*匹配正好一个词。比如，`abc.#`匹配 `abc.def.ghi`，`abc.*`只匹配 `abc.def`。
3. Fanout 交换机：不需要任何 Key，它采取广播的模式，一个消息进来时，投递到与该交换机绑定的所有队列。

RabbitMQ 支持消息的持久化，也就是将数据写在磁盘上。为了数据安全考虑，大多数情况下都会选择持久化。消息队列持久化包括 3 个部分：

1. Exchange 持久化，在声明时指定 durable => 1。
2. Queue 持久化，在声明时指定 durable => 1。
3. 消息持久化，在投递时指定 delivery_mode => 2（1 是非持久化）。

如果 Exchange 和 Queue 都是持久化的，那么它们之间的 Binding 也是持久化的。如果 Exchange 和 Queue 两者之间有一个是持久化的，一个是非持久化的，就不允许建立绑定。

安装与使用

在 RabbitMQ 官网的下载页面 `https://www.rabbitmq.com/download.html` 中，我们可以获取到针对各种不同操作系统的安装包和说明文档。这里，我们将对几个常用的平台进行一一说明。

下面我们采用的是 Erlang 和 RabbitMQ Server 版本说明：

- Erlang/OTP 19.1
- RabbitMQ Server 3.6.5

在 Windows 系统中安装

1. 安装 Erlang，通过官方下载页面 http://www.erlang.org/downloads 获取 exe 安装包，直接打开并完成安装。
2. 安装 RabbitMQ，通过官方下载页面 https://www.rabbitmq.com/download.html 获取 exe 安装包。
3. 下载完成后，直接运行安装程序。
4. RabbitMQ Server 安装完成之后，会自动注册为服务，并以默认配置进行启动。

名称	描述	状态	启动类型	登录为
RabbitMQ	Multi-protocol open source messaging broker	已启动	自动	本地系统

在 Windows 的安装过程中，有时候会碰到服务启动失败的情况，通常都是由于用户名为中文，导致默认的 db 和 log 目录访问出现问题。要解决该问题，需要先卸载 RabbitMQ Server，然后设置环境变量 RABBITMQ_BASE 为一个不含中文的路径，比如 E:\server\rabbitmq。最后，重新安装 RabbitMQ 即可。

在 Mac OS X 中安装

在 Mac OS X 中使用 brew 工具，可以很容易地安装 RabbitMQ 的服务端，只需按如下命令操作即可：

1. 将 brew 更新到最新版本，执行 brew update 命令。
2. 安装 Erlang，执行 brew install erlang 命令。
3. 安装 RabbitMQ Server，执行 brew install rabbitmq 命令。

通过执行上面的命令，RabbitMQ Server 会被安装到/usr/local/sbin，并不会自动加到用户的环境变量中去，所以我们需要在.bash_profile 或.profile 文件中增加下面的内容：

```
PATH=$PATH:/usr/local/sbin
```

这样，就可以通过 rabbitmq-server 命令来启动 RabbitMQ 的服务端了。

在 Ubuntu 中安装

在 Ubuntu 中，我们可以使用 APT 仓库来进行安装：

1. 安装 Erlang，执行 apt-get install erlang 命令。
2. 执行下面的命令，新增 APT 仓库到/etc/apt/sources.list.d：

```
echo 'deb http://www.rabbitmq.com/debian/ testing main' |
    sudo tee /etc/apt/sources.list.d/rabbitmq.list
```

3. 更新 APT 仓库的 package list，执行 sudo apt-get update 命令。

4. 安装 Rabbit Server，执行 sudo apt-get install rabbitmq-server 命令。

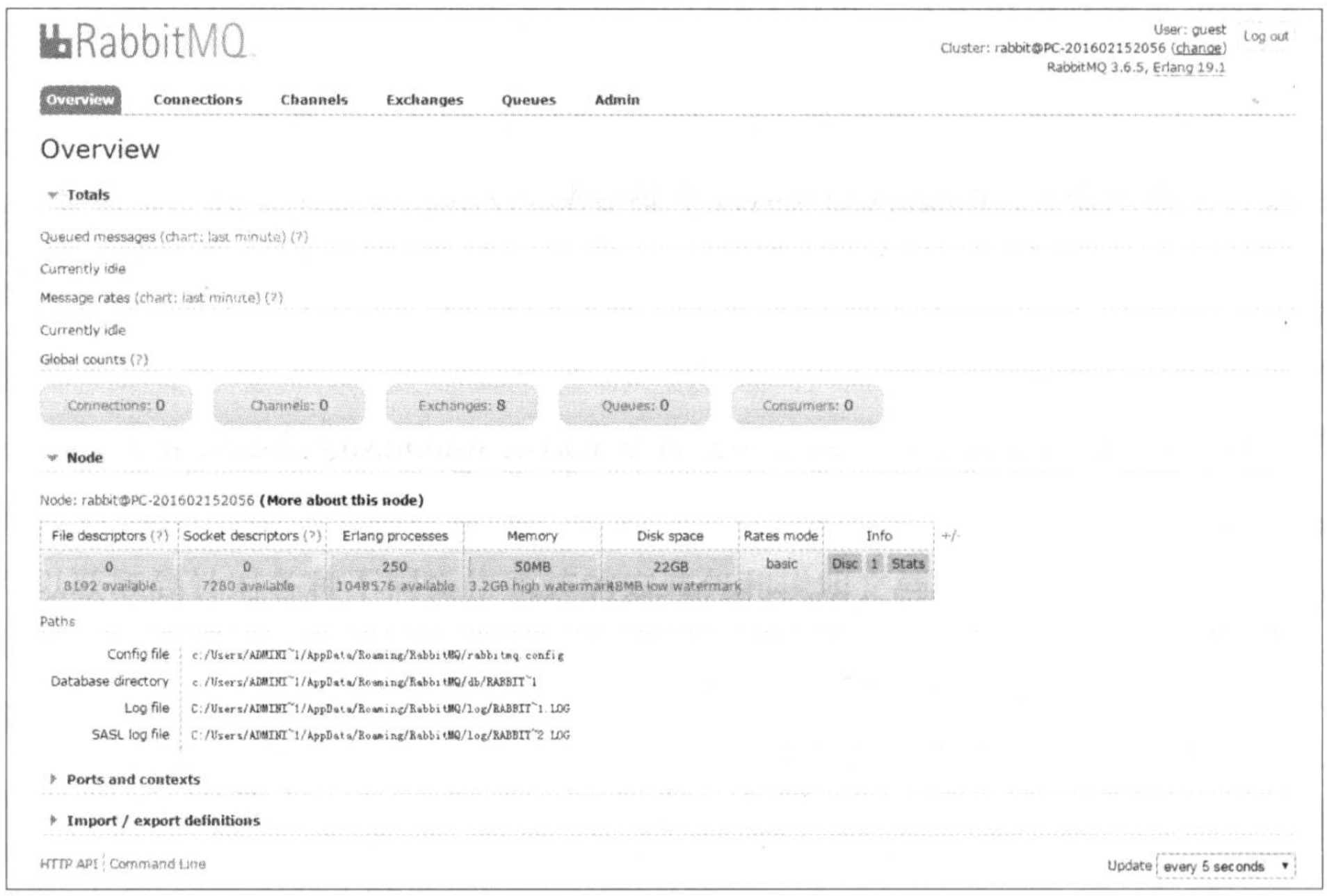
Rabbit 管理

我们可以直接通过访问配置文件进行管理，也可以通过访问 Web 进行管理。下面将介绍如何通过 Web 进行管理。

- 执行 rabbitmq-plugins enable rabbitmq_management 命令，开启 Web 管理插件，这样就可以通过浏览器来进行管理了。

```
> rabbitmq-plugins enable rabbitmq_management
The following plugins have been enabled:
  mochiweb
  webmachine
  rabbitmq_web_dispatch
  amqp_client
  rabbitmq_management_agent
  rabbitmq_management

Applying plugin configuration to rabbit@PC-201602152056... started 6 plugins.
```

- 打开浏览器并访问 http://localhost:15672/，并使用默认用户 guest 登录，密码也为 guest。可以看到如下图所示的管理页面：



从图中我们可以看到之前章节中提到的一些基本概念，比如 Connections、Channels、Exchanges、Queues 等。第一次使用的读者，可以点开各项看看都有些什么内容，熟悉一下 RabbitMQ Server 的服务端。

- 单击 Admin 选项卡，如下图所示，可以尝试创建一个名为 `springcloud` 的用户。

其中，Tags 标签是 RabbitMQ 中的角色分类，共有下面几种。

- none：不能访问 management plugin。
- management：用户可以通过 AMQP 做的任何事外加如下内容。
  - 列出自己可以通过 AMQP 登入的 virtual hosts。
  - 查看自己的 virtual hosts 中的 queues、exchanges 和 bindings。
  - 查看和关闭自己的 channels 和 connections。
  - 查看有关自己的 virtual hosts 的“全局”统计信息，包含其他用户在这些 virtual hosts 中的活动。
- policymaker：management 可以做的任何事外加如下内容。
  - 查看、创建和删除自己的 virtual hosts 所属的 policies 和 parameters。
- monitoring：management 可以做的任何事外加如下内容。
  - 列出所有 virtual hosts，包括它们不能登录的 virtual hosts。
  - 查看其他用户的 connections 和 channels。
  - 查看节点级别的数据，如 clustering 和 memory 的使用情况。

  - 查看真正的关于所有 virtual hosts 的全局的统计信息。
- administrator：policymaker 和 monitoring 可以做的任何事外加如下内容。
  - 创建和删除 virtual hosts。
  - 查看、创建和删除 users。
  - 查看、创建和删除 permissions。
  - 关闭其他用户的 connections。

快速入门

接下来，我们通过在 Spring Boot 应用中整合 RabbitMQ，实现一个简单的发送、接收消息的例子来对 RabbitMQ 有一个直观的感受和理解。

在 Spring Boot 中整合 RabbitMQ 是一件非常容易的事，因为之前我们已经介绍过 Starter POMs，其中的 AMQP 模块就可以很好地支持 RabbitMQ，下面我们就来详细说说整合过程。

- 新建一个 Spring Boot 工程，命名为 `rabbitmq-hello`。
- 在 `pom.xml` 中引入如下依赖内容，其中 `spring-boot-starter-amqp` 用于支持 RabbitMQ。

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-amqp</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

- 在 `application.properties` 中配置关于 RabbitMQ 的连接和用户信息，这里我们使用之前安装时创建的 `springcloud`。若没有自己的用户，可以回到上面的安装内容，在管理页面中创建用户。

```
spring.application.name=rabbitmq-hello

spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=springcloud
spring.rabbitmq.password=123456
```

- 创建消息生产者 Sender。通过注入 AmqpTemplate 接口的实例来实现消息的发送，AmqpTemplate 接口定义了一套针对 AMQP 协议的基础操作。在 Spring Boot 中会根据配置来注入其具体实现。在该生产者中，我们会产生一个字符串，并发送到名为 hello 的队列中。

```
@Component
public class Sender {

    @Autowired
    private AmqpTemplate rabbitTemplate;

    public void send() {
        String context = "hello " + new Date();
        System.out.println("Sender : " + context);
        this.rabbitTemplate.convertAndSend("hello", context);
    }

}
```

- 创建消息消费者 Receiver。通过@RabbitListener 注解定义该类对 hello 队列的监听，并用@RabbitHandler 注解来指定对消息的处理方法。所以，该消费者实现了对 hello 队列的消费，消费操作为输出消息的字符串内容。

```
@Component
@RabbitListener(queues = "hello")
public class Receiver {

    @RabbitHandler
    public void process(String hello) {
        System.out.println("Receiver : " + hello);
    }

}
```

- 创建 RabbitMQ 的配置类 RabbitConfig，用来配置队列、交换器、路由等高级信息。这里我们以入门为主，先以最小化的配置来定义，以完成一个基本的生产和消费过程。

```
@Configuration
public class RabbitConfig {
```

```
    @Bean
    public Queue helloQueue() {
        return new Queue("hello");
    }

}
```

- 创建应用主类。

```
@SpringBootApplication
public class HelloApplication {

    public static void main(String[] args) {
        SpringApplication.run(HelloApplication.class, args);
    }

}
```

- 创建单元测试类，用来调用消息生产。

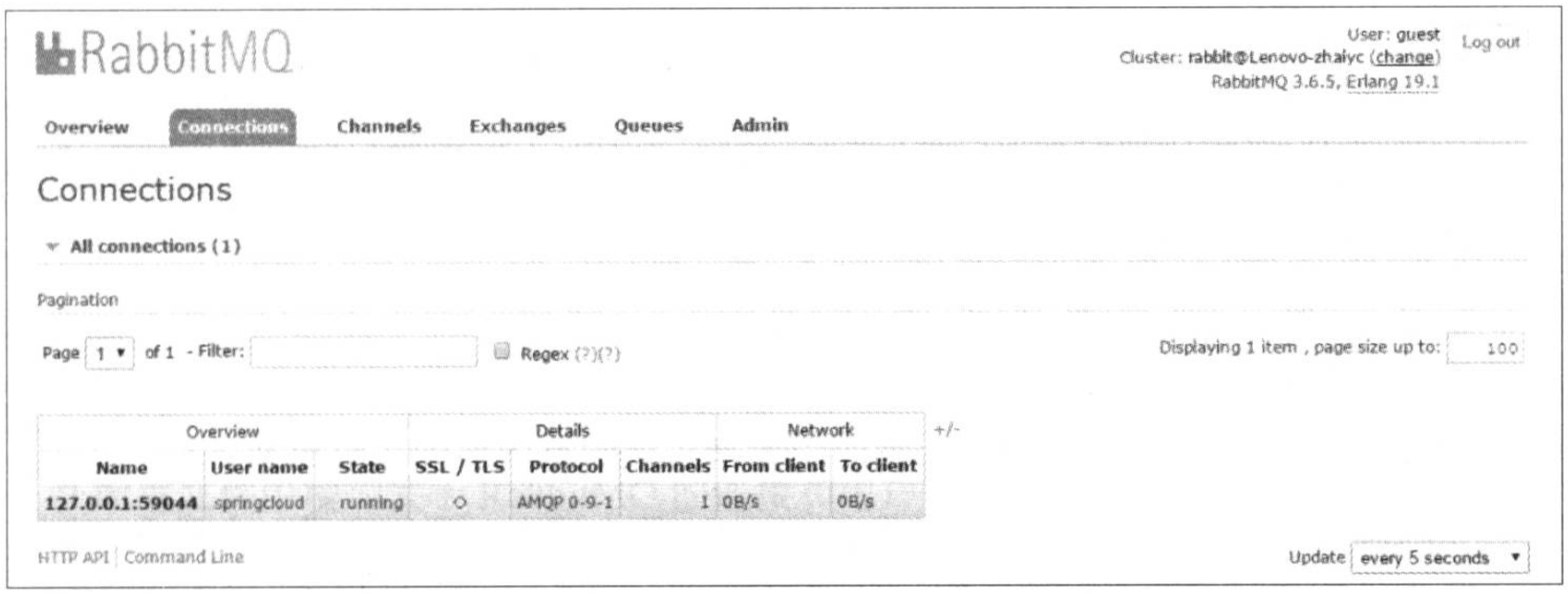
```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = HelloApplication.class)
public class HelloApplicationTests {

    @Autowired
    private Sender sender;

    @Test
    public void hello() throws Exception {
        sender.send();
    }

}
```

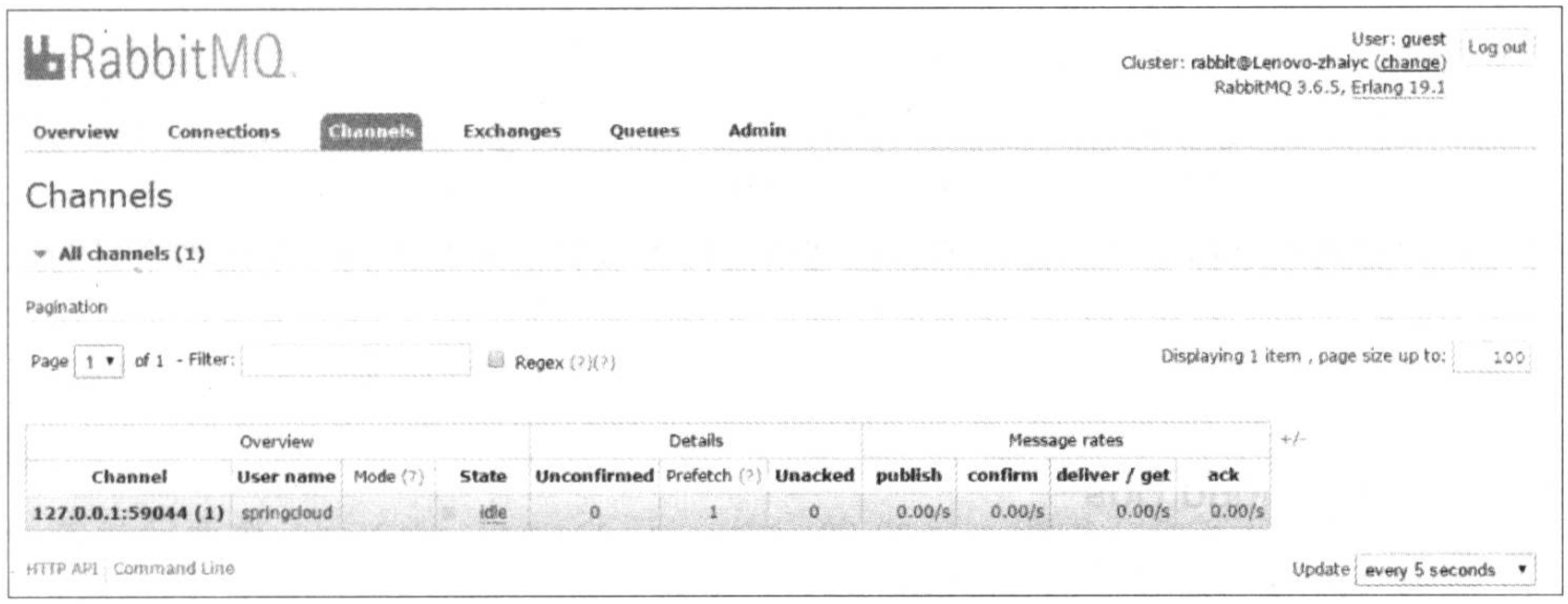
完成程序编写之后，下面开始尝试运行。首先确保 RabbitMQ Server 已经启动，然后进行下面的操作。

- 启动应用主类，从控制台中，我们可看到如下内容，程序创建了一个访问 `127.0.0.1:5672` 中 `springcloud` 的连接。

```
o.s.a.r.c.CachingConnectionFactory : Created new connection: SimpleConnection@29836d32
[delegate=amqp://springcloud@127.0.0.1:5672/]
```

  同时，我们通过 RabbitMQ 的控制面板，可以看到 Connections 和 Channels 中包含当前连接的条目。





- 运行单元测试类，我们可以在控制台中看到下面的输出内容，消息被发送到了 RabbitMQ Server 的 `hello` 队列中。

```
Sender : hello Sun Sep 25 11:06:11 CST 2016
```

- 切换到应用主类的控制台，我们可以看到类似如下的输出，消费者对 `hello` 队列的监听程序执行了，并输出了接收到的消息信息。

```
Receiver : hello Sun Sep 25 11:06:11 CST 2016
```

通过上面的示例，我们在 Spring Boot 应用中引入 `spring-boot-starter-amqp` 模块，进行简单配置就完成了对 RabbitMQ 的消息生产和消费的开发内容。然而在实际应用中，还有很多内容没有演示，比如之前提到的一些概念：交换机、路由关键字、绑定、虚拟主机等，这里不做更多的讲解，读者可以自行查阅 RabbitMQ 的官方教程，其中有更全面的讲解。在这里，我们需要重点理解的是，在整个生产消费过程中，生产和消费是一个异步操作，这也是在分布式系统中要使用消息代理的重要原因，以此我们可以使用通信来解耦业务逻辑。在这个例子中，读者可以进一步做一些测试，比如，不运行消费者，先运

行生产者，此时可以看到在 RabbitMQ Server 管理页面的 Queues 选项卡下多了一些待处理的消息，这时我们再启动消费者，它就会处理这些消息，所以通过生产消费模式的异步操作，系统间调用就没有同步调用需要那么高的实时性要求，同时也更容易控制处理的吞吐量以保证系统的正常运行等。

在上一节中，我们已经介绍了关于消息代理、AMQP 以及 RabbitMQ 的基础知识和使用方法。在下面的内容中，我们开始具体介绍 Spring Cloud Bus 的配置，并以一个 Spring Cloud Bus 与 Spring Cloud Config 结合的例子来实现配置内容的实时更新。

先回顾一下，在上一章 Spring Cloud Config 的介绍中，我们留了一个悬念：如何实现对配置信息的实时更新。虽然我们已经能够通过`/refresh`接口和 Git 仓库的 Web Hook 来实现 Git 仓库中的内容修改触发应用程序的属性更新。但是，若所有触发操作均需要我们手工去维护 Web Hook 中的应用配置的话，随着系统的不断扩展，会变得越来越难以维护，而消息代理中间件是解决该问题最为合适的方案。是否还记得我们在介绍消息代理中的特点时提到过这样一个功能：消息代理中间件可以将消息路由到一个或多个目的地。利用这个功能，我们就能完美地解决该问题，下面来说说 Spring Cloud Bus 中的具体实现方案。

整合 Spring Cloud Bus

因为 Spring Cloud 基于 Spring Boot，在上一节中我们已经体验了 Spring Boot 与 RabbitMQ 的整合，所以在 Spring Cloud Bus 中使用 RabbitMQ 也是非常容易配置的。

下面我们来具体动手尝试整个配置过程。

- 准备工作：这里我们不创建新的应用，但需要用到上一章中已经实现的关于 Spring Cloud Config 的几个工程，若读者对其还不了解，建议先阅读第 8 章的内容。
  - `config-repo`：定义在 Git 仓库中的一个目录，其中存储了应用名为`didispace`的多环境配置文件，配置文件中有一个`from`参数。
  - `config-server-eureka`：配置了 Git 仓库，并注册到了 Eureka 的服务端。
  - `config-client-eureka`：通过 Eureka 发现 Config Server 的客户端，应用名为`didispace`，用来访问配置服务器以获取配置信息。该应用中提供了一个`/from`接口，它会获取`config-repo/didispace-dev.properties`中的`from`属性并返回。

- 扩展 config-client-eureka 应用。
  - 修改pom.xml，增加spring-cloud-starter-bus-amqp模块(注意spring-boot-starter-actuator 模块也是必需的，用来提供刷新端点)。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-amqp</artifactId>
</dependency>
```

- 在配置文件中增加关于 RabbitMQ 的连接和用户信息。

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=springcloud
spring.rabbitmq.password=123456
```

- 启动 config-server-eureka，再启动两个 config-client-eureka（分别在不同的端口上，比如 7002、7003）。我们可以在 config-client-eureka 中的控制台中看到如下内容，在启动时，客户端程序多了一个/bus/refresh 请求。

```
    o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped
"{[/bus/refresh],methods=[POST]}" onto public void
org.springframework.cloud.bus.endpoint.RefreshBusEndpoint.refresh(java.lang.String)
```

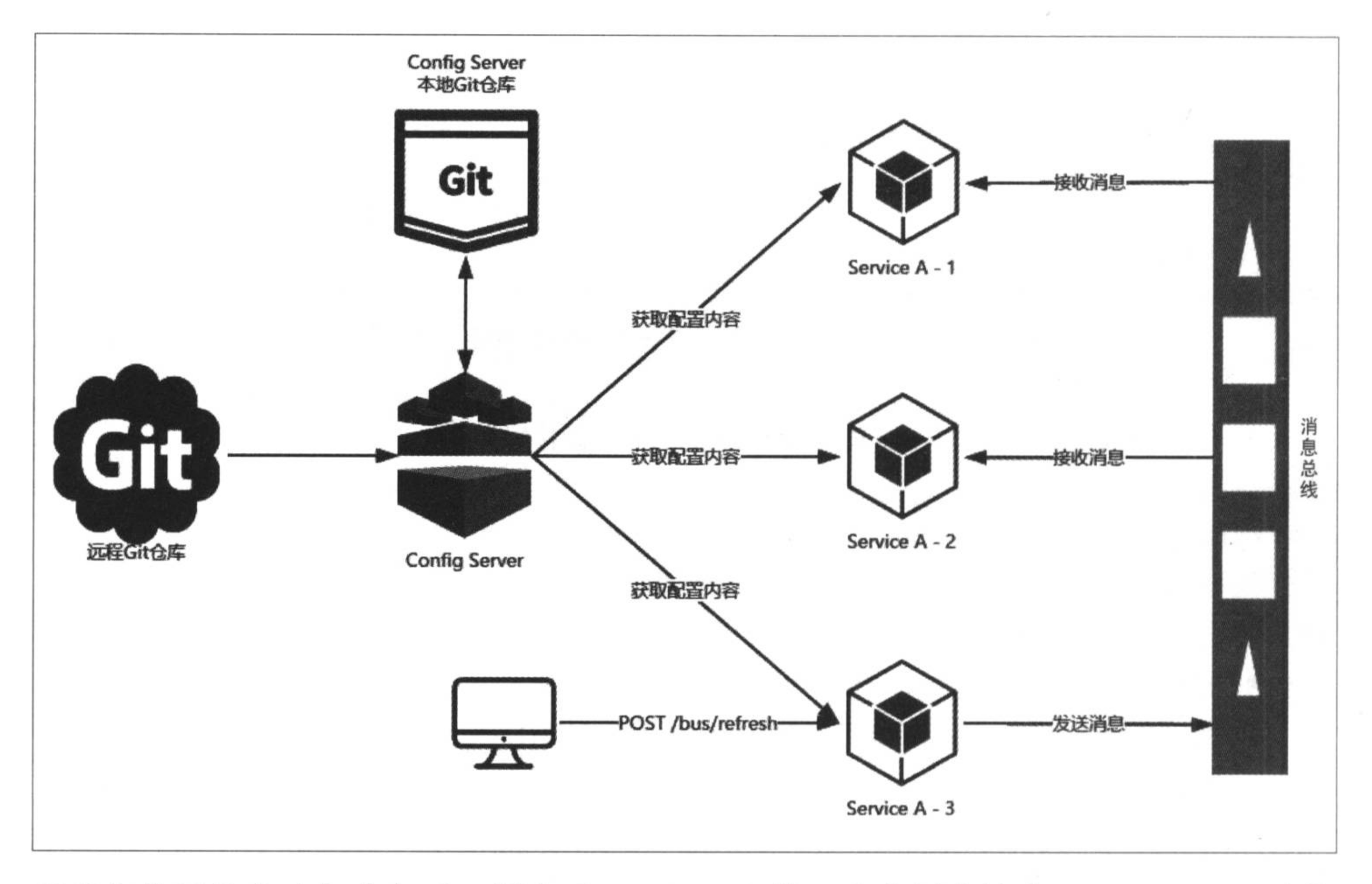
- 先访问两个 config-client-eureka 的/from 请求，会返回当前 config-repo/didispace-dev.properties 中的 from 属性。
- 接着，修改 config-repo/didispace-dev.properties 中的 from 属性值，并发送 POST 请求到其中的一个/bus/refresh。
- 最后，再分别访问启动的两个 config-client-eureka 的/from 请求，此时这两个请求都会返回最新的 config-repo/didispace-dev.properties 中的 from 属性。

到这里，我们已经能够通过 Spring Cloud Bus 来实时更新总线上的属性配置了。

原理分析

上一节中，我们通过使用 Spring Cloud Bus 与 Spring Cloud Config 的整合，并以 RabbitMQ 作为消息代理，实现了应用配置的动态更新。

整个方案的架构如下图所示，其中包含了 Git 仓库、Config Server 以及几个微服务应用的实例，这些微服务应用的实例中都引入了 Spring Cloud Bus，所以它们都连接到了 RabbitMQ 的消息总线上。



当我们将系统启动起来之后，图中“Service A”的三个实例会请求 Config Server 以获取配置信息，Config Server 根据应用配置的规则从 Git 仓库中获取配置信息并返回。

此时，若我们需要修改“Service A”的属性。首先，通过 Git 管理工具去仓库中修改对应的属性值，但是这个修改并不会触发“Service A”实例的属性更新。我们向“Service A”的实例 3 发送 POST 请求，访问`/bus/refresh`接口。此时，“Service A”的实例 3 就会将刷新请求发送到消息总线中，该消息事件会被“Service A”的实例 1 和实例 2 从总线中获取到，并重新从 Config Server 中获取它们的配置信息，从而实现配置信息的动态更新。

而从 Git 仓库中配置的修改到发起`/bus/refresh`的 POST 请求这一步可以通过 Git 仓库的 Web Hook 来自动触发。由于所有连接到消息总线上的应用都会接收到更新请求，所以在 Web Hook 中就不需要维护所有节点内容来进行更新，从而解决了上一章中仅通过 Web Hook 来逐个进行刷新的问题。

指定刷新范围

在上面的例子中，我们通过向服务实例请求 Spring Cloud Bus 的`/bus/refresh`接口，从而触发总线上其他服务实例的`/refresh`。但是在一些特殊场景下，我们希望可以刷新微服务中某个具体实例的配置。

Spring Cloud Bus 对这种场景也有很好的支持，/bus/refresh 接口提供了一个 destination 参数，用来定位具体要刷新的应用程序。比如，我们可以请求 /bus/refresh?destination=customers:9000，此时总线上的各应用实例会根据 destination 属性的值来判断是否为自己的实例名，若符合才进行配置刷新，若不符合就忽略该消息。

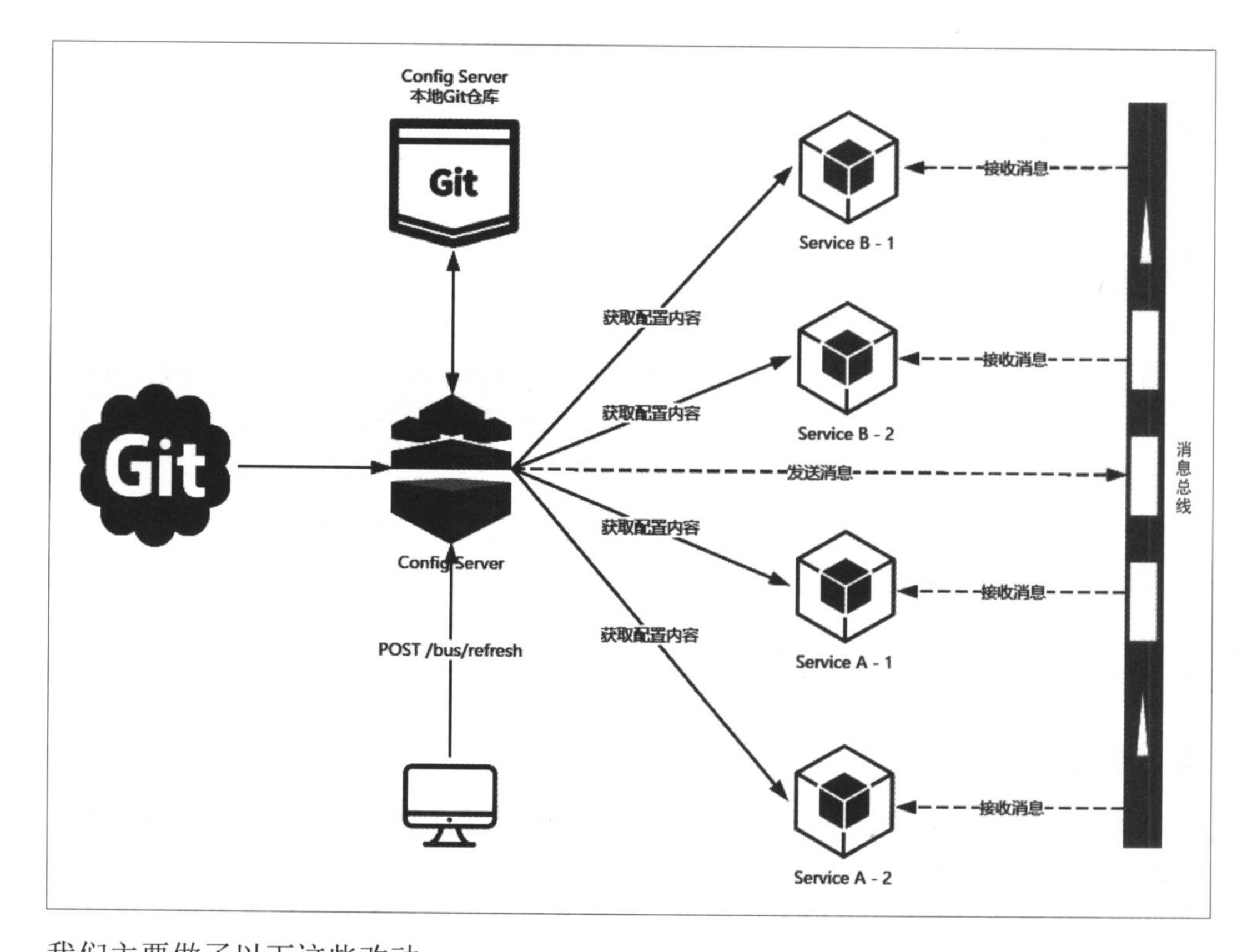
关于应用的实例名，我们在之前介绍 Spring Cloud Netflix 的 Eureka 时有过详细的介绍，它的默认命名按此规则生成：${spring.cloud.client.hostname}:${spring.application.name}:${spring.application.instance_id:${server.port}}，想了解更多内容可查看第 3 章的相关介绍。

destination 参数除了可以定位具体的实例之外，还可以用来定位具体的服务。定位服务的原理是通过使用 Spring 的 PathMatecher（路径匹配）来实现的，比如 /bus/refresh?destination=customers:**，该请求会触发 customers 服务的所有实例进行刷新。

架构优化

既然 Spring Cloud Bus 的/bus/refresh 接口提供了针对服务和实例进行配置更新的参数，那么我们的架构也可以相应做出一些调整。在之前的架构中，服务的配置更新需要通过向具体服务中的某个实例发送请求，再触发对整个服务集群的配置更新。虽然能实现功能，但是这样的结果是，我们指定的应用实例会不同于集群中的其他应用实例，这样会增加集群内部的复杂度，不利于将来的运维工作。比如，需要对服务实例进行迁移，那么我们不得不修改 Web Hook 中的配置等。所以要尽可能地让服务集群中的各个节点是对等的。

因此，我们将之前的架构做了一些调整，如下图所示。



我们主要做了以下这些改动：

1. 在 Config Server 中也引入 Spring Cloud Bus，将配置服务端也加入到消息总线中来。
2. `/bus/refresh` 请求不再发送到具体服务实例上，而是发送给 Config Server，并通过 `destination` 参数来指定需要更新配置的服务或实例。

通过上面的改动，我们的服务实例不需要再承担触发配置更新的职责。同时，对于 Git 的触发等配置都只需要针对 Config Server 即可，从而简化了集群上的一些维护工作。

RabbitMQ 配置

Spring Cloud Bus 中的 RabbitMQ 整合使用了 Spring Boot 的 `ConnectionFactory`，所以在 Spring Cloud Bus 中支持使用以 `spring.rabbit.mq` 为前缀的 Spring Boot 配置属性，具体的配置属性、说明以及默认值如下表所示。

属性名	说明	默认值
spring.rabbitmq.addresses	客户端连接的地址，有多个的时候使用逗号分隔，该地址可以是 IP 与 Port 的结合	
spring.rabbitmq.cache.channel.checkout-timeout	当缓存已满时，获取 Channel 的等待时间，单位为毫秒	
spring.rabbitmq.cache.channel.size	缓存中保持的 Channel 数量	
spring.rabbitmq.cache.connection.mode	连接缓存的模式	CHANNEL
spring.rabbitmq.cache.connection.size	缓存的连接数	
spring.rabbitmq.connection-timeout	连接超时参数，单位为毫秒；设置为“0”代表无穷大	
spring.rabbitmq.dynamic	默认创建一个 AmqpAdmin 的 Bean	true
spring.rabbitmq.host	RabbitMQ 的主机地址	localhost
spring.rabbitmq.listener.acknowledge-mode	容器的 acknowledge 模式	
spring.rabbitmq.listener.auto-startup	启动时自动启动容器	true
spring.rabbitmq.listener.concurrency	消费者的最小数量	
spring.rabbitmq.listener.default-requeue-rejected	投递失败时是否重新排队	true
spring.rabbitmq.listener.max-concurrency	消费者的最大数量	
spring.rabbitmq.listener.prefetch	在单个请求中处理的消息个数，它应该大于等于事务数量	
spring.rabbitmq.listener.retry.enabled	不论是不是重试的发布	false
spring.rabbitmq.listener.retry.initial-interval	第一次与第二次投递尝试的时间间隔	1000
spring.rabbitmq.listener.retry.max-attempts	尝试投递消息的最大数量	3
spring.rabbitmq.listener.retry.max-interval	两次尝试的最大时间间隔	10000
spring.rabbitmq.listener.retry.multiplier	上一次尝试时间间隔的乘数	1.0
spring.rabbitmq.listener.retry.stateless	不论重试是有状态的还是无状态的	true
spring.rabbitmq.listener.transaction-size	在一个事务中处理的消息数量。为了获得最佳效果，该值应设置为小于等于每个请求中处理的消息个数，即 `spring.rabbitmq.listener.prefetch` 的值	
spring.rabbitmq.password	登录到 RabbitMQ 的密码	
spring.rabbitmq.port	RabbitMQ 的端口号	5672
spring.rabbitmq.publisher-confirms	开启 Publisher Confirm 机制	false
spring.rabbitmq.publisher-returns	开启 Publisher Return 机制	false
spring.rabbitmq.requested-heartbeat	请求心跳超时时间，单位为秒	
spring.rabbitmq.ssl.enabled	启用 SSL 支持	false

续表

属性名	说明	默认值
spring.rabbitmq.ssl.key-store	保存 SSL 证书的地址	
spring.rabbitmq.ssl.key-store-password	访问 SSL 证书的地址使用的密码	
spring.rabbitmq.ssl.trust-store	SSL 的可信地址	
spring.rabbitmq.ssl.trust-store-password	访问 SSL 的可信地址的密码	
spring.rabbitmq.ssl.algorithm	SSL 算法，默认使用 Rabbit 的客户端算法库	
spring.rabbitmq.template.mandatory	启用强制消息	false
spring.rabbitmq.template.receive-timeout	`receive()`方法的超时时间	0
spring.rabbitmq.template.reply-timeout	`sendAndReceive()`方法的超时时间	5000
spring.rabbitmq.template.retry.enabled	设置为 true 的时候 `RabbitTemplate` 能够实现重试	false
spring.rabbitmq.template.retry.initial-interval	第一次与第二次发布消息的时间间隔	1000
spring.rabbitmq.template.retry.max-attempts	尝试发布消息的最大数量	3
spring.rabbitmq.template.retry.max-interval	尝试发布消息的最大时间间隔	10000
spring.rabbitmq.template.retry.multiplier	上一次尝试时间间隔的乘数	1.0
spring.rabbitmq.username	登录到 RabbitMQ 的用户名	
spring.rabbitmq.virtual-host	连接到 RabbitMQ 的虚拟主机	

Kafka 实现消息总线

Spring Cloud Bus 除了支持 RabbitMQ 的自动化配置之外，还支持现在被广泛应用的 Kafka。在本节中，我们将搭建一个 Kafka 的本地环境，并通过它来尝试使用 Spring Cloud Bus 对 Kafka 的支持，实现消息总线的功能。

Kafka 简介

Kafka 是一个由 LinkedIn 开发的分布式消息系统，它于 2011 年年初开源，现在由著名的 Apache 基金会维护与开发。Kafka 使用 Scala 实现，被用作 LinkedIn 的活动流和运营数据处理的管道，现在也被诸多互联网企业广泛地用作数据流管道和消息系统。

Kafka 是基于消息发布-订阅模式实现的消息系统，其主要设计目标如下所述。

- 消息持久化：以时间复杂度为 $O(1)$的方式提供消息持久化能力，即使对 TB 级以上的数据也能保证常数时间复杂度的访问性能。
- 高吞吐：在廉价的商用机器上也能支持单机每秒 10 万条以上的吞吐量。

- 分布式：支持消息分区以及分布式消费，并保证分区内的消息顺序。
- 跨平台：支持不同技术平台的客户端（如 Java、PHP、Python 等）。
- 实时性：支持实时数据处理和离线数据处理。
- 伸缩性：支持水平扩展。

Kafka 中涉及的一些基本概念，如下所示。

- Broker：Kafka 集群包含一个或多个服务器，这些服务器被称为 Broker。
- Topic：逻辑上同 RabbitMQ 的 Queue 队列相似，每条发布到 Kafka 集群的消息都必须有一个 Topic。（物理上不同 Topic 的消息分开存储，逻辑上一个 Topic 的消息虽然保存于一个或多个 Broker 上，但用户只需指定消息的 Topic 即可生产或消费数据而不必关心数据存于何处。）
- Partition：Partition 是物理概念上的分区，为了提供系统吞吐率，在物理上每个 Topic 会分成一个或多个 Partition，每个 Partition 对应一个文件夹（存储对应分区的消息内容和索引文件）。
- Producer：消息生产者，负责生产消息并发送到 Kafka Broker。
- Consumer：消息消费者，向 Kafka Broker 读取消息并处理的客户端。
- Consumer Group：每个 Consumer 属于一个特定的组（可为每个 Consumer 指定属于一个组，若不指定则属于默认组），组可以用来实现一条消息被组内多个成员消费等功能。

快速入门

在对 Kafka 有了一些基本了解之后，下面我们来尝试搭建一个 Kafka 服务端，并体验一下基于 Kafka 的消息生产与消费。

环境安装

首先，我们需要从官网上下载安装介质。下载地址为 `http://kafka.apache.org/downloads.html`。本例中采用的版本为 Kafka-0.10.0.1。

在解压 Kafka 的安装包之后，可以看到其目录结构如下所示：

```
kafka
  +-bin
    +-windows
  +-config
  +-libs
  +-logs
  +-site-docs
```

由于 Kafka 的设计中依赖了 ZooKeeper，所以我们在 bin 和 config 目录中除了看到 Kafka 相关的内容之外，还有 ZooKeeper 相关的内容。其中 bin 目录中存放了 Kafka 和 ZooKeeper 的命令行工具，bin 根目录下存放的是适用于 Linux/UNIX 的 shell，而 bin/windows 下存放的则是适用于 Windows 下的 bat。我们可以根据实际的系统来设置环境变量，以方便后续的使用和操作。而 config 目录，则用来存放关于 Kafka 与 ZooKeeper 的配置信息。

启动测试

下面我们来尝试启动 ZooKeeper 和 Kafka 来进行消息的生产和消费。示例中所有的命令均以配置了 Kafka 的环境变量为例。

- 启动 ZooKeeper，执行命令 zookeeper-server-start config/zookeeper.properties，该命令需要指定 ZooKeeper 的配置文件位置才能正确启动，Kafka 的压缩包中包含了其默认配置，开发与测试环境基本不需要修改，所以这里不做详细介绍，对于线上的调优需求，请读者自行查看官方文档进行操作。

```
[2016-09-28 08:05:34,849] INFO Reading configuration from: config\zookeeper.
properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2016-09-28 08:05:34,850] INFO autopurge.snapRetainCount set to 3 (org.apache.
zookeeper.server.DatadirCleanupManager)
[2016-09-28 08:05:34,851] INFO autopurge.purgeInterval set to 0 (org.apache.
zookeeper.server.DatadirCleanupManager)
[2016-09-28 08:05:34,851] INFO Purge task is not scheduled. (org.apache.zookeeper.
server.DatadirCleanupManager)
[2016-09-28 08:05:34,852] WARN Either no config or no quorum defined in config,
running in standalone mode (org.apache.zookeeper.server.quorum.QuorumPeerMain)
[2016-09-28 08:05:34,868] INFO Reading configuration from: config\zookeeper.
properties (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
[2016-09-28 08:05:34,869] INFO Starting server (org.apache.zookeeper.server.
ZooKeeperServerMain)
...
[2016-09-28 08:05:34,940] INFO binding to port 0.0.0.0/0.0.0.0:2181 (org.apache.
zookeeper.server.NIOServerCnxnFactory)
```

从控制台信息中我们可以看到，ZooKeeper 从指定的 config/zookeeper.properties 配置文件中读取信息并绑定 2181 端口启动服务。有时候启动失败，可查看一下端口是否被占用，可以杀掉占用进程或通过修改 config/zookeeper.properties 配置文件中的 clientPort 内容以绑定其他端口号来启动 ZooKeeper。

- 启动 Kafka，执行命令 kafka-server-start config/server.properties，该命令也需要指定 Kafka 配置文件的正确位置，如上命令中指向了解压目录包含的默认配置。若在测试时，使用外部集中环境的 ZooKeeper 的话，我们可以在该配置

文件中通过 zookeeper.connect 参数来设置 ZooKeeper 的地址和端口，它默认会连接本地 2181 端口的 ZooKeeper；如果需要设置多个 ZooKeeper 节点，可以为这个参数配置多个 ZooKeeper 地址，并用逗号分隔。比如 zookeeper.connect=127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002。除此之外，该配置文件中还提供了关于服务端连接、日志等配置参数，具体的线上配置可根据实际情况进行调整。

- 创建 Topic，执行命令 kafka-topics --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test。通过该命令，创建了一个名为 test 的 Topic，该 Topic 包含一个分区和一个 Replica。在创建完成后，可以使用 kafka-topics --list --zookeeper localhost:2181 命令来查看当前的 Topic。

另外，如果不使用 kafka-topics 命令来手工创建，直接使用下面的内容进行消息创建时也会自动创建 Topics。

- 创建消息生产者，执行命令 kafka-console-producer --broker-list localhost:9092 --topic test。kafka-console-producer 命令可以启动 Kafka 基于命令行的消息生产客户端，启动后可以直接在控制台中输入消息来发送，控制台中的每一行数据都会被视为一条消息来发送。我们可以尝试输入几行消息，由于此时并没有消费者，所以这些输入的消息都会被阻塞在名为 test 的 Topics 中，直到有消费者将其消费掉。
- 创建消息消费者，执行命令 kafka-console-consumer --zookeeper localhost:2181 --topic test --from-beginning。kafka-console-consumer 命令启动的是 Kafka 基于命令行的消息消费客户端，启动之后，马上可以在控制台中看到输出了之前我们在消息生产客户端中发送的消息。我们可以再次打开之前的消息生产客户端来发送消息，并观察消费者这边对消息的输出来体验 Kafka 对消息的基础处理。

整合 Spring Cloud Bus

在介绍 Kafka 之前，我们已经通过引入 spring-cloud-starter-bus-amqp 模块，完成了使用 RabbitMQ 来实现消息总线。若我们要使用 Kafka 来实现消息总线时，只需把 spring-cloud-starter-bus-amqp 替换成 spring-cloud-starter-bus- kafka 模块，在 pom.xml 的 dependency 节点中进行修改，具体如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bus-kafka</artifactId>
</dependency>
```

如果在启动 Kafka 时均采用了默认配置，那么我们不需要再做任何其他配置就能在本地实现从 RabbitMQ 到 Kafka 的切换。可以尝试把刚刚搭建的 ZooKeeper、Kafka 启动起来，并将修改为 spring-cloud-starter-bus-kafka 模块的 config-server 和 config-client 启动起来。

在 config-server 启动时，我们可以在控制台中看到如下输出：

```
    2016-09-28 22:11:29.627  INFO 15144 --- [          main]
o.s.c.s.b.k.KafkaMessageChannelBinder    : Using kafka topic for outbound:
springCloudBus
    2016-09-28 22:11:29.642  INFO 15144 --- [-localhost:2181]
org.I0Itec.zkclient.ZkEventThread        : Starting ZkClient event thread.
    ...
    016-09-28 22:11:30.290  INFO 15144 --- [          main]
o.s.i.kafka.support.ProducerFactoryBean  : Using producer properties =>
{bootstrap.servers=localhost:9092, linger.ms=0, acks=1, compression.type=none,
batch.size=16384}
    2016-09-28 22:11:30.298  INFO 15144 --- [          main]
o.a.k.clients.producer.ProducerConfig    : ProducerConfig values:
    ...
    2016-09-28 22:11:30.322  INFO 15144 --- [          main]
o.s.c.s.b.k.KafkaMessageChannelBinder$1  : Adding
{message-handler:outbound.springCloudBus} as a subscriber to the 'springCloudBusOutput'
channel
    2016-09-28 22:11:30.322  INFO 15144 --- [          main]
o.s.integration.channel.DirectChannel    : Channel
'config-server:7001.springCloudBusOutput' has 1 subscriber(s).
    2016-09-28 22:11:30.322  INFO 15144 --- [          main]
o.s.c.s.b.k.KafkaMessageChannelBinder$1  : started outbound.springCloudBus
    ...
    2016-09-28 22:11:31.465  INFO 15144 --- [          main]
s.i.k.i.KafkaMessageDrivenChannelAdapter : started
org.springframework.integration.kafka.inbound.KafkaMessageDrivenChannelAdapter@4178
cb34
    2016-09-28 22:11:31.467  INFO 15144 --- [          main]
o.s.c.s.b.k.KafkaMessageChannelBinder$7  : Adding
{message-handler:inbound.springCloudBus.anonymous.8b9e6c7b-6a50-48c5-b981-8282a0d5a
30b} as a subscriber to the 'bridge.springCloudBus' channel
    2016-09-28 22:11:31.467  INFO 15144 --- [          main]
o.s.c.s.b.k.KafkaMessageChannelBinder$7  : started
inbound.springCloudBus.anonymous.8b9e6c7b-6a50-48c5-b981-8282a0d5a30b
```

从控制台的输出内容我们可以看到，config-server 连接到了 Kafka 中，并使用了名为 springCloudBus 的 Topic。

此时，我们可以使用 kafka-topics --list --zookeeper localhost:2181 命令来查看当前 Kafka 中的 Topic。若已成功启动了 config-server 并配置正确，可以在 Kafka 中看到已经多了一个名为 springCloudBus 的 Topic。

我们再启动配置了 spring-cloud-starter-bus-kafka 模块的 config-client，可以看到控制台中输出了如下内容：

```
    2016-09-28 22:43:55.067  INFO 6136 --- [          main]
o.s.c.s.b.k.KafkaMessageChannelBinder   : Using kafka topic for outbound:
springCloudBus
    2016-09-28 22:43:55.078  INFO 6136 --- [-localhost:2181]
org.I0Itec.zkclient.ZkEventThread       : Starting ZkClient event thread.
    ...
    2016-09-28 22:50:38.584  INFO 828 --- [          main]
o.s.i.kafka.support.ProducerFactoryBean  : Using producer properties =>
{bootstrap.servers=localhost:9092, linger.ms=0, acks=1, compression.type=none,
batch.size=16384}
    2016-09-28 22:50:38.592  INFO 828 --- [          main]
o.a.k.clients.producer.ProducerConfig   : ProducerConfig values:
    ...
    2016-09-28 22:50:38.615  INFO 828 --- [          main]
o.s.c.s.b.k.KafkaMessageChannelBinder$1  : Adding
{message-handler:outbound.springCloudBus} as a subscriber to the 'springCloudBusOutput'
channel
    2016-09-28 22:50:38.616  INFO 828 --- [          main]
o.s.integration.channel.DirectChannel   : Channel
'didispace:7002.springCloudBusOutput' has 1 subscriber(s).
    2016-09-28 22:50:38.616  INFO 828 --- [          main]
o.s.c.s.b.k.KafkaMessageChannelBinder$1  : started outbound.springCloudBus
    ...
    2016-09-28 22:50:39.162  INFO 828 --- [          main]
s.i.k.i.KafkaMessageDrivenChannelAdapter : started
org.springframework.integration.kafka.inbound.KafkaMessageDrivenChannelAdapter@60cf
855e
    2016-09-28 22:50:39.162  INFO 828 --- [          main]
o.s.c.s.b.k.KafkaMessageChannelBinder$7  : Adding
{message-handler:inbound.springCloudBus.anonymous.f8fc9c0c-ccd3-46dd-9537-07198f4ee
216} as a subscriber to the 'bridge.springCloudBus' channel
    2016-09-28 22:50:39.163  INFO 828 --- [          main]
o.s.c.s.b.k.KafkaMessageChannelBinder$7  : started
inbound.springCloudBus.anonymous.f8fc9c0c-ccd3-46dd-9537-07198f4ee216
```

可以看到，config-client 启动时输出了类似的内容，它们都订阅了名为 springCloudBus 的 Topic。从这里我们也可以知道，在消息总线上的节点，从结构上来说，不论是 config-server 还是 config-client，它们都是对等的。

在启动了 config-server 和 config-client 之后，为了更明显地观察消息总线刷新配置的效果，我们可以在本地启动多个不同端口的 config-client。此时，我们的 config-server 以及多个 config-client 都已经连接到了由 Kafka 实现的消息总线上。我们可以先访问各个 config-client 上的/from 请求，查看它获取到的配置内容。然后，修改 Git 中对应的参数内容，再访问各个 config-client 上的/from 请求，可以

看到配置内容并没有改变。最后，我们向 config-server 发送 POST 请求：/bus/refresh，此时再去访问各个 config-client 上的/from 请求，就能获得最新的配置信息，各客户端上的配置都已经加载为最新的 Git 配置内容。

从 config-client 的控制台中，我们可以看到如下内容：

```
    2016-09-29 08:20:34.361  INFO 21256 --- [ kafka-binder-1]
o.s.cloud.bus.event.RefreshListener     : Received remote refresh request. Keys
refreshed [from]
```

RefreshListener 监听类记录了收到远程刷新请求，并刷新了 from 属性的日志，在下一节中，我们将根据消息内容与日志输出信息作为线索来探索 Spring Cloud Bus 的工作机制。

Kafka 配置

在上面的例子中，由于 Kafka、ZooKeeper 均运行于本地，在自动化配置的支持下，我们没有在测试程序中通过配置信息来指定 Kafka 和 ZooKeeper 的配置信息，就完成了本地消息总线的试验。但是在实际应用中，Kafka 和 ZooKeeper 一般都会独立部署，所以在应用中需要为 Kafka 和 ZooKeeper 配置一些连接信息等。Kafka 的整合与 RabbitMQ 不同，在 Spring Boot 1.3.7 中并没有直接提供 Starter 模块，而是采用了 Spring Cloud Stream 的 Kafka 模块，所以对于 Kafka 的配置均采用了 spring.cloud.stream.kafka 前缀，具体的配置内容我们可以参考第 10 章的“绑定器配置”一节中关于 Kafka 配置的内容。

深入理解

在整合 Kafka 实现了消息总线之后，我们不妨继续使用 Kafka 提供的控制台消费者来看看，当执行/bus/refresh 时，消息消费者都获得了什么。通过前文我们从控制台中获得的信息可以知道，Spring Cloud Bus 使用了名为 springCloudBus 的 Topic，所以我们可以使用命令 kafka-console-consumer --zookeeper localhost:2181 --topic springCloudBus，启动对 springCloudBus 的消费者控制台来进行观察。

启动消费者控制台之后，我们向 config-server 发送 POST 请求：/bus/refresh，此时在控制台中可以看到类似如下的内容：

```
    contentType  "application/json"
    {
        "type": "RefreshRemoteApplicationEvent",
        "timestamp": 1475073160814,
        "originService": "config-server:7001",
        "destinationService": "*:**",
        "id": "bbfbf495-39d8-4ff9-93d6-174873ff7299"
```

```
}

contentType  "application/json"
{
    "type": "AckRemoteApplicationEvent",
    "timestamp": 1475073160821,
    "originService": "config-server:7001",
    "destinationService": "*:**",
    "id": "1f794774-10d6-4140-a80d-470983c6c0ff",
    "ackId": "bbfbf495-39d8-4ff9-93d6-174873ff7299",
    "ackDestinationService": "*:**",
    "event": "org.springframework.cloud.bus.event.RefreshRemoteApplicationEvent"
}

contentType  "application/json"
{
    "type": "AckRemoteApplicationEvent",
    "timestamp": 1475075467554,
    "originService": "didispace:7002",
    "destinationService": "*:**",
    "id": "7560151e-f60c-49cd-8167-b691e846ad08",
    "ackId": "21502725-28f5-4d19-a98a-f8114fa4f1dc",
    "ackDestinationService": "*:**",
    "event": "org.springframework.cloud.bus.event.RefreshRemoteApplicationEvent"
}
...
```

下面，我们来详细理解消息中的信息内容。

- type：消息的事件类型。在上面的例子中，包含了 RefreshRemoteApplicationEvent 和 AckRemoteApplicationEvent。其中，RefreshRemoteApplicationEvent 事件就是我们用来刷新配置的事件，而 AckRemoteApplicationEvent 是响应消息已经正确接收的告知消息事件。
- timestamp：消息的时间戳。
- originService：消息的来源服务实例。
- destinationService：消息的目标服务实例。上面示例中的*:**代表了总线上的所有服务实例。如果想要指定服务或是实例，在之前介绍 RabbitMQ 实现消息总线时已经提过，只需要通过使用 destination 参数来定位具体要刷新的应用实例即可，比如发起/bus/refresh?destination=didispace 请求，就可以得到如下的刷新事件消息，其中 destinationService 为 didispace:**，表示总线上所有 didispace 服务的实例。

```
contentType  "application/json"
{
    "type": "RefreshRemoteApplicationEvent",
    "timestamp": 1475131215007,
    "originService": "config-server:7001",
    "destinationService": "didispace:**",
    "id": "667fe948-e9b2-447f-be22-3c8acf647ead"
}
```

- id：消息的唯一标识。

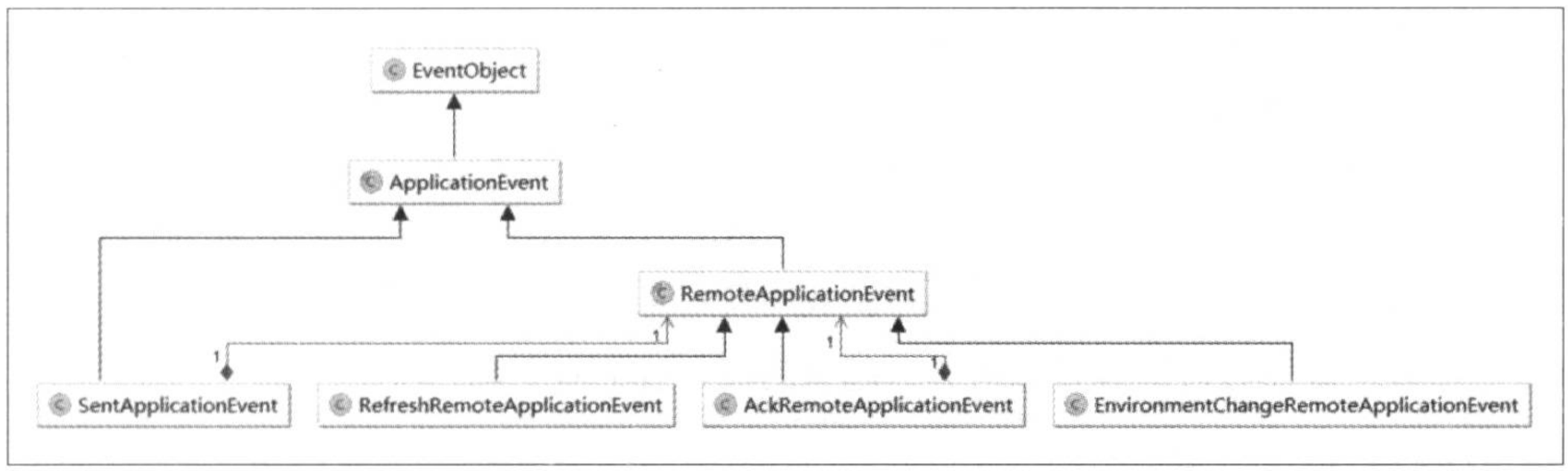
上面的消息内容是 RefreshRemoteApplicationEvent 和 AckRemoteApplicationEvent 类型共有的，下面几个属性是 AckRemoteApplicationEvent 所特有的，分别表示如下含义。

- ackId：Ack 消息对应的消息来源。我们可以看到第一条 AckRemoteApplicationEvent 的 ackId 对应了 RefreshRemoteApplicationEvent 的 id，说明这条 Ack 是告知该 RefreshRemoteApplicationEvent 事件的消息已经被收到。
- ackDestinationService：Ack 消息的目标服务实例。可以看到这里使用的是 *:**，所以消息总线上所有的实例都会收到该 Ack 消息。
- event：Ack 消息的来源事件。可以看到上例中的两个 Ack 均来源于刷新配置的 RefreshRemoteApplicationEvent 事件，我们在测试的时候由于启动了两个 config-client，所以有两个实例接收到了配置刷新事件，同时它们都会返回一个 Ack 消息。由于 ackDestinationService 为*:**，所以两个 config-client 都会收到对 RefreshRemoteApplicationEvent 事件的 Ack 消息。

源码分析

通过上面的分析，我们已经得到了两个非常重要的线索 RefreshRemoteApplicationEvent 和 AckRemoteApplicationEvent。我们不妨顺着这两个事件类来详细看看 Spring Cloud Bus 的源码，以帮助我们理解它的运行机制。

顺着 RefreshRemoteApplicationEvent 和 AckRemoteApplicationEvent，我们可以整理出如下的事件关系类图。

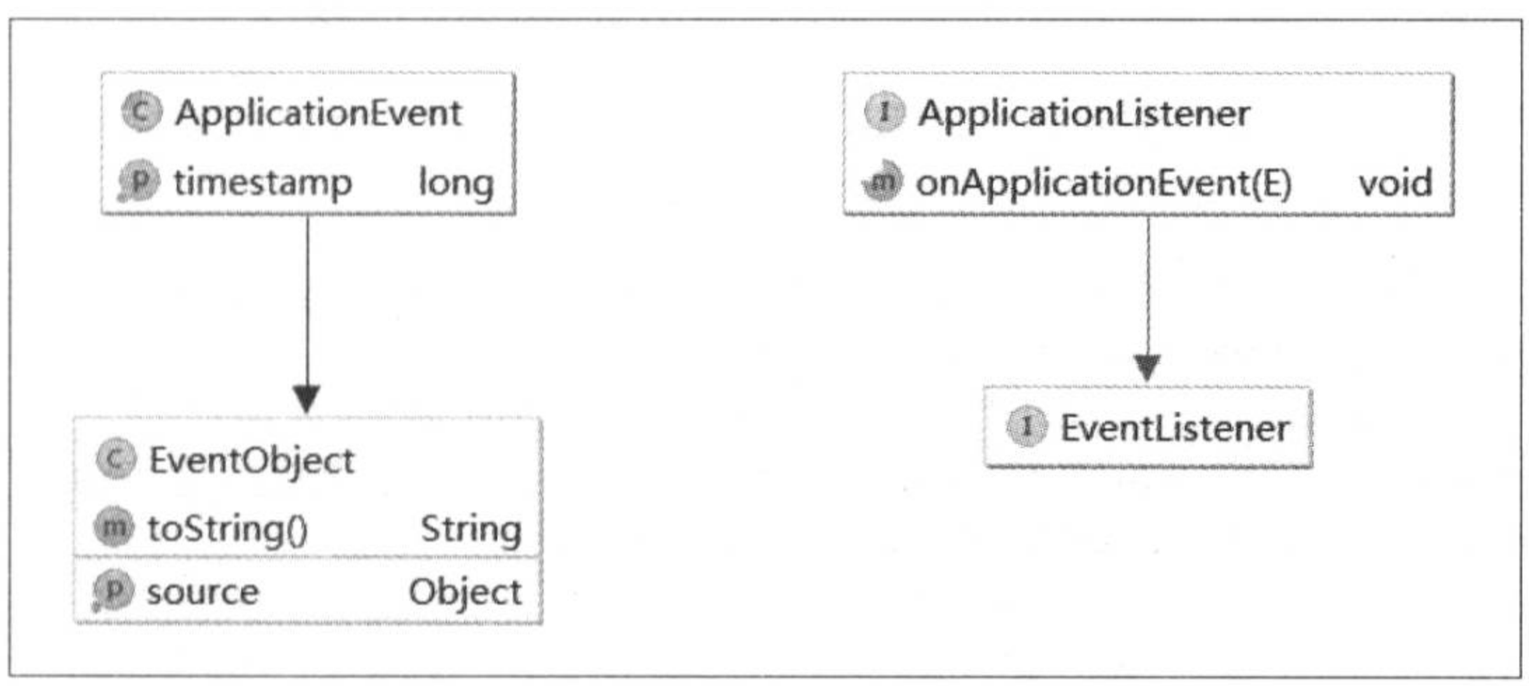


可以看到，其中 RefreshRemoteApplicationEvent 和 AckRemoteApplication-Event 这些我们已经接触过的事件都继承了 RemoteApplicationEvent 抽象类，而 RemoteApplicationEvent 继承自 Spring Framework 的 ApplicationEvent，可以断定，Spring Cloud Bus 也采用了 Spring 的事件驱动模型。

事件驱动模型

如果读者对 Spring 的事件驱动模型已经非常了解，那么可以跳过这一小节，直接看后面的分析。如果你还不清楚它的原理，建议先通过本小节的内容来理解其基本原理，以帮助阅读和理解后续的源码分析内容。

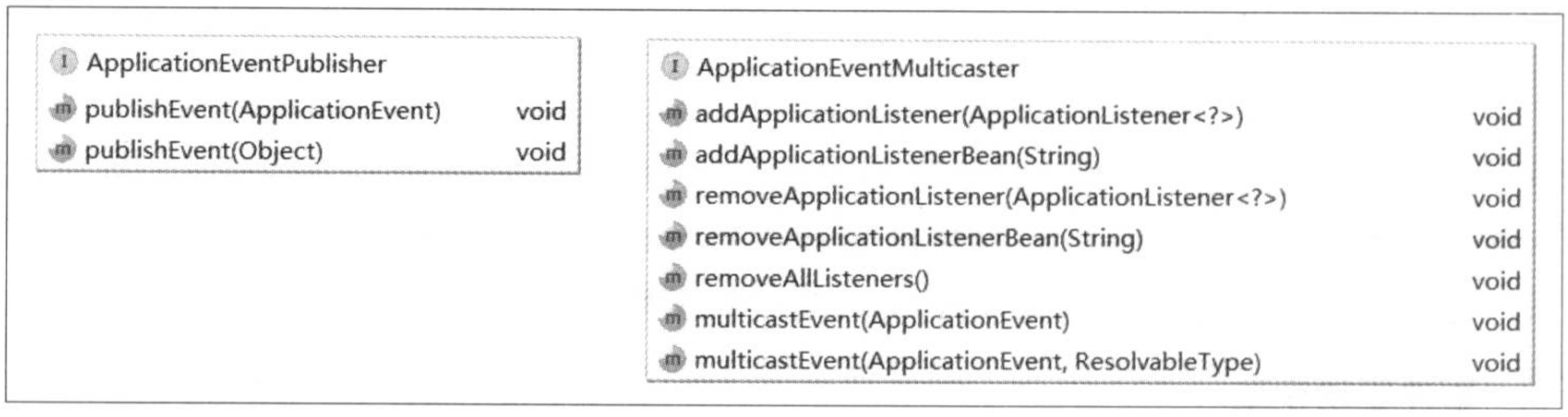
Spring 的事件驱动模型中包含了三个基本概念：事件、事件监听者和事件发布者，如下图所示。



- **事件**：Spring 中定义了事件的抽象类 ApplicationEvent，它继承自 JDK 的 EventObject 类。从图中我们可以看到，事件包含了两个成员变量：timestamp，该字段用于存储事件发生的时间戳，以及父类中的 source，该字段表示源事件对象。当我们需要自定义事件的时候，只需要继承 ApplicationEvent，比如 RemoteApplicationEvent、RefreshRemoteApplicationEvent 等，可以在自定义的 Event 中增加一些事件的属性来给事件监听者处理。

- **事件监听者**：Spring 中定义了事件监听者的接口 ApplicationListener，它继承自 JDK 的 EventListener 接口，同时 ApplicationListener 接口限定了 ApplicationEvent 子类作为该接口中 onApplicationEvent(E event);函数的参数。所以，每一个 ApplicationListener 都是针对某个 ApplicationEvent 子类的监听和处理者。

那么，事件与监听者是如何关联起来的呢？我们看下图：



- **事件发布者**：Spring 中定义了 ApplicationEventPublisher 和 ApplicationEventMulticaster 两个接口用来发布事件。其中 ApplicationEventPublisher 接口定义了发布事件的函数 publishEvent(ApplicationEvent event) 和 publishEvent(Object event)；而 ApplicationEventMulticaster 接口中定义了对 ApplicationListener 的维护操作（比如新增、移除等）以及将 ApplicationEvent 多播给可用 ApplicationListener 的操作。

ApplicationEventPublisher 的 publishEvent 实现在 AbstractApplicationContext 中，具体如下：

```
protected void publishEvent(Object event, ResolvableType eventType) {
    Assert.notNull(event, "Event must not be null");
    ...
    if (this.earlyApplicationEvents != null) {
      this.earlyApplicationEvents.add(applicationEvent);
    }
    else {
      getApplicationEventMulticaster().multicastEvent(applicationEvent,
eventType);
    }
    ...
  }
```

可以看到，它最终会调用 ApplicationEventMulticaster 的 multicastEvent 来具体实现发布事件给监听者的操作。而 ApplicationEventMulticaster 在 Spring 的默认实现位于 SimpleApplicationEventMulticaster 中，具体如下：

```
    public void multicastEvent(final ApplicationEvent event, ResolvableType eventType) {
      ResolvableType type = (eventType != null ? eventType :
resolveDefaultEventType(event));
      for (final ApplicationListener<?> listener : getApplicationListeners(event, type)) {
        Executor executor = getTaskExecutor();
        if (executor != null) {
          executor.execute(new Runnable() {
            @Override
            public void run() {
              invokeListener(listener, event);
            }
          });
        }
        else {
          invokeListener(listener, event);
        }
      }
    }

    protected void invokeListener(ApplicationListener listener, ApplicationEvent event) {
      ErrorHandler errorHandler = getErrorHandler();
      if (errorHandler != null) {
        try {
          listener.onApplicationEvent(event);
        }
        catch (Throwable err) {
          errorHandler.handleError(err);
        }
      }
      else {
        try {
          listener.onApplicationEvent(event);
        }
        catch (ClassCastException ex) {
          LogFactory.getLog(getClass()).debug("Non-matching event type for listener: "
+ listener, ex);
        }
      }
    }
```

SimpleApplicationEventMulticaster 通过遍历维护的 ApplicationListener 集合来找到对应 ApplicationEvent 的监听器，然后调用监听器的 onApplicationEvent 函数来对具体事件做出处理操作。

事件定义

在对 Spring 的事件模型有了一定的理解之后，下面我们来详细介绍 Spring Cloud Bus 中的事件定义。首先，从 RemoteApplicationEvent 抽象类开始：

```
@SuppressWarnings("serial")
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property = "type")
@JsonIgnoreProperties("source")
public abstract class RemoteApplicationEvent extends ApplicationEvent {
    private static final Object TRANSIENT_SOURCE = new Object();
    private final String originService;
    private final String destinationService;
    private final String id;

    protected RemoteApplicationEvent() {
        // for serialization libs like jackson
        this(TRANSIENT_SOURCE, null, null);
    }

    protected RemoteApplicationEvent(Object source, String originService,
            String destinationService) {
        super(source);
        this.originService = originService;
        if (destinationService == null) {
            destinationService = "*";
        }
        if (!destinationService.contains(":")) {
            // All instances of the destination unless specifically requested
            destinationService = destinationService + ":**";
        }
        this.destinationService = destinationService;
        this.id = UUID.randomUUID().toString();
    }

    protected RemoteApplicationEvent(Object source, String originService) {
        this(source, originService, null);
    }

    ...
}
```

先来看看 RemoteApplicationEvent 类上修饰的注解。

- @JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property = "type"): Jackson 对多态类型的处理注解，当进行序列化时，会使用子类的名称作为 type 属性的值，比如之前示例中的"type": "RefreshRemoteApplicationEvent"。

- @JsonIgnoreProperties("source")：序列化的时候忽略 source 属性，source 是 ApplicationEvent 的父类 EventObject 的属性，用来定义事件的发生源。

再来看看它的属性：originService、destinationService、id，这些内容都可以在 RemoteApplicationEvent 的子类事件消息中找到，比如：

```
{
    "type": "RefreshRemoteApplicationEvent",
    "timestamp": 1475073160814,
    "originService": "config-server:7001",
    "destinationService": "*:**",
    "id": "bbfbf495-39d8-4ff9-93d6-174873ff7299"
}

{
    "type": "AckRemoteApplicationEvent",
    "timestamp": 1475075467554,
    "originService": "didispace:7002",
    "destinationService": "*:**",
    "id": "7560151e-f60c-49cd-8167-b691e846ad08",
    "ackId": "21502725-28f5-4d19-a98a-f8114fa4f1dc",
    "ackDestinationService": "*:**",
    "event": "org.springframework.cloud.bus.event.RefreshRemoteApplicationEvent"
}
```

下面，我们再来分别看看 RemoteApplicationEvent 的几个具体实现的事件类。

- RefreshRemoteApplicationEvent 事件类，该事件用于远程刷新应用的配置信息。它的实现非常简单，只是继承了 RemoteApplicationEvent，并没有增加其他内容。从之前的示例中我们也能看到，消息中的内容与 RemoteApplication-Event 中包含的属性完全一致。

```
@SuppressWarnings("serial")
public class RefreshRemoteApplicationEvent extends RemoteApplicationEvent {

    @SuppressWarnings("unused")
    private RefreshRemoteApplicationEvent() {
        // for serializers
    }

    public RefreshRemoteApplicationEvent(Object source, String originService,
            String destinationService) {
```

```
        super(source, originService, destinationService);
    }
}
```

- AckRemoteApplicationEvent 事件类，该事件用于告知某个事件消息已经被接收，通过该消息我们可以监控各个事件消息的响应。从其成员属性中，我们可以找到之前示例中所总结的，比 RefreshRemoteApplicationEvent 事件的消息多出的几个属性：ackId、ackDestinationService 以及 event。其中 event 成员变量通过泛型限定了必须为 RemoteApplicationEvent 的子类对象，该定义符合这样的逻辑：Ack 消息肯定有一个事件源头，而每一个事件都必须继承 RemoteApplicationEvent 抽象类，所以 AckRemoteApplicationEvent 的事件源头肯定是一个 RemoteApplicationEvent 的子类，比如示例中的 Ack 消息源头就是RemoteApplicationEvent 的子类事件:RefreshRemoteApplicationEvent。

```
@SuppressWarnings("serial")
public class AckRemoteApplicationEvent extends RemoteApplicationEvent {

    private final String ackId;
    private final String ackDestinationService;
    private final Class<? extends RemoteApplicationEvent> event;

    @SuppressWarnings("unused")
    private AckRemoteApplicationEvent() {
        super();
        this.ackDestinationService = null;
        this.ackId = null;
        this.event = null;
    }

    public AckRemoteApplicationEvent(Object source, String originService,
            String destinationService, String ackDestinationService, String ackId,
            Class<? extends RemoteApplicationEvent> type) {
        super(source, originService, destinationService);
        this.ackDestinationService = ackDestinationService;
        this.ackId = ackId;
        this.event = type;
    }

    ...
}
```

- EnvironmentChangeRemoteApplicationEvent 事件类，该事件用于动态更新消息总线上每个节点的 Spring 环境属性。可以看到，该类中定义了一个 Map 类型

的成员变量，而接收消息的节点就是根据该 Map 对象中的属性来覆盖本地的 Spring 环境属性。

```
    @SuppressWarnings("serial")
    public class EnvironmentChangeRemoteApplicationEvent extends
RemoteApplicationEvent {

        private final Map<String, String> values;

    @SuppressWarnings("unused")
    private EnvironmentChangeRemoteApplicationEvent() {
        // for serializers
        values = null;
    }

    public EnvironmentChangeRemoteApplicationEvent(Object source, String originService,
            String destinationService, Map<String, String> values) {
        super(source, originService, destinationService);
        this.values = values;
    }
    ...
}
```

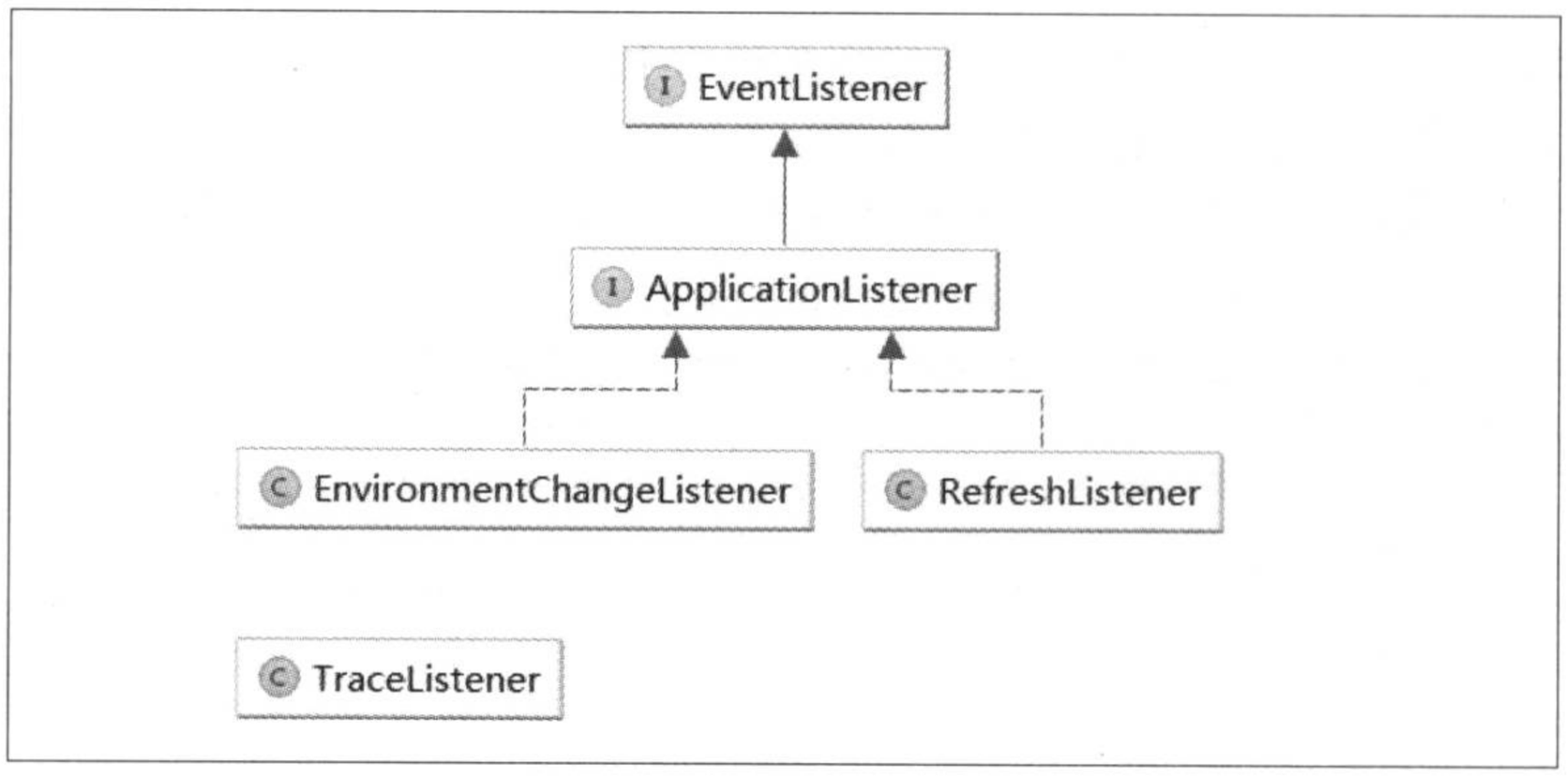
- SentApplicationEvent 事件类，细心的读者可能已经发现，该类的结构和内容与 RemoteApplicationEvent 非常相似，不同的是：该类不是抽象类，并且多一个成员 Class<? extends RemoteApplicationEvent> type。SentApplicationEvent 事件较为特殊，它主要用于发送信号来表示一个远程的事件已经在系统中被发送到某些地方了，从它的继承关系中，我们可以知道它本身并不是一个远程的事件（不是继承自 RemoteApplicationEvent），所以它不会被发送到消息总线上去，而是在本地产生（通常是由于响应了某个远程的事件）。由于该事件的 id 属性能够匹配消费者 AckRemoteApplicationEvent 消息中的 ackId，所以应用程序可以通过监听这个事件来监控远程事件消息的消费情况。

```
    @SuppressWarnings("serial")
    @JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property = "type")
    @JsonIgnoreProperties("source")
    public class SentApplicationEvent extends ApplicationEvent {

        private static final Object TRANSIENT_SOURCE = new Object();
        private final String originService;
        private final String destinationService;
        private final String id;
        private Class<? extends RemoteApplicationEvent> type;
```

```
    protected SentApplicationEvent() {
        // for serialization libs like jackson
        this(TRANSIENT_SOURCE, null, null, null, RemoteApplicationEvent.class);
    }

    public SentApplicationEvent(Object source, String originService,
            String destinationService, String id,
            Class<? extends RemoteApplicationEvent> type) {
        super(source);
        this.originService = originService;
        this.type = type;
        if (destinationService == null) {
            destinationService = "*";
        }
        if (!destinationService.contains(":")) {
            // All instances of the destination unless specifically requested
            destinationService = destinationService + ":**";
        }
        this.destinationService = destinationService;
        this.id = id;
    }

    ...
}
```

事件监听器

在了解了 Spring Cloud Bus 中的事件类之后，我们来看看另外一个重要元素：事件监听器。通过整理源码，可以得到下面的类图关系。



其中，`RefreshListener` 和 `EnvironmentChangeListener` 都继承了 Spring 事

件模型中的监听器接口 ApplicationListener。我们先来看看 RefreshListener：

```
public class RefreshListener
      implements ApplicationListener<RefreshRemoteApplicationEvent> {

   private static Log log = LogFactory.getLog(RefreshListener.class);

   private ContextRefresher contextRefresher;

   public RefreshListener(ContextRefresher contextRefresher) {
      this.contextRefresher = contextRefresher;
   }

   @Override
   public void onApplicationEvent(RefreshRemoteApplicationEvent event) {
      Set<String> keys = contextRefresher.refresh();
      log.info("Received remote refresh request. Keys refreshed " + keys);
   }
}
```

从泛型中我们可以看到该监听器就是针对我们之前所介绍的 RefreshRemoteApplicationEvent 事件的，其中 onApplicationEvent 函数中调用了 ContextRefresher 中的 refresh() 函数进行配置属性的刷新。

```
public class ContextRefresher {
   ...
   private ConfigurableApplicationContext context;
   ...
   public synchronized Set<String> refresh() {
      Map<String, Object> before = extract(
            this.context.getEnvironment().getPropertySources());
      addConfigFilesToEnvironment();
      Set<String> keys = changes(before,

extract(this.context.getEnvironment().getPropertySources())).keySet();
      this.context.publishEvent(new EnvironmentChangeEvent(keys));
      this.scope.refreshAll();
      return keys;
   }
   ...
}
```

再来看看 EnvironmentChangeListener 监听器。

```
public class EnvironmentChangeListener
      implements ApplicationListener<EnvironmentChangeRemoteApplicationEvent> {
```

```
    private static Log log = LogFactory.getLog(EnvironmentChangeListener.class);

    @Autowired
    private EnvironmentManager env;

    @Override
    public void onApplicationEvent(EnvironmentChangeRemoteApplicationEvent event) {
        Map<String, String> values = event.getValues();
        log.info("Received remote environment change request. Keys/values to update "
                + values);
        for (Map.Entry<String, String> entry : values.entrySet()) {
            env.setProperty(entry.getKey(), entry.getValue());
        }
    }
}
```

它是针对 EnvironmentChangeRemoteApplicationEvent 事件的监听类，在处理类中，可以看到它从 EnvironmentChangeRemoteApplicationEvent 中获取了之前提到的事件中定义的 Map 对象，然后通过遍历来更新 EnvironmentManager 中的属性内容。

事件跟踪

除了上面介绍的 RefreshListener 和 EnvironmentChangeListener 监听器外，还有一个与它们都有点不同的 TraceListener 监听器。

```
public class TraceListener {

    private static Log log = LogFactory.getLog(TraceListener.class);

    private TraceRepository repository;

    public TraceListener(TraceRepository repository) {
        this.repository = repository;
    }

    @EventListener
    public void onAck(AckRemoteApplicationEvent event) {
        this.repository.add(getReceivedTrace(event));
    }

    @EventListener
    public void onSend(SentApplicationEvent event) {
        this.repository.add(getSentTrace(event));
    }
```

```
    protected Map<String, Object> getSentTrace(SentApplicationEvent event) {
        ...
    }

    protected Map<String, Object> getReceivedTrace(AckRemoteApplicationEvent event) {
        ...
    }
}
```

从之前整理的类图和源码中，我们都可以看到该监听器并没有实现 ApplicationListener 接口，但可以看到这里使用了@EventListener 注解。该注解是从 Spring 4.2 开始提供的新功能，通过它可以自动地将函数注册为一个 ApplicationListener 的实现。所以在该类中，实际上等价于实现了两个监听器，一个监听 AckRemoteApplicationEvent 事件，一个监听 SentApplicationEvent 事件。

在这两个监听处理函数中调用了类似的方法：this.repository.add(getReceivedTrace(event));，其中 TraceRepository 是对 Trace 跟踪信息的操作接口，而它的默认实现是 spring-boot-actuator 模块的 InMemoryTraceRepository，具体如下：

```
public class InMemoryTraceRepository implements TraceRepository {

    private int capacity = 100;
    private boolean reverse = true;
    private final List<Trace> traces = new LinkedList<Trace>();

    public void setReverse(boolean reverse) {
        synchronized (this.traces) {
            this.reverse = reverse;
        }
    }

    public void setCapacity(int capacity) {
        synchronized (this.traces) {
            this.capacity = capacity;
        }
    }

    @Override
    public List<Trace> findAll() {
        synchronized (this.traces) {
            return Collections.unmodifiableList(new ArrayList<Trace>(this.traces));
        }
    }
```

```
    @Override
    public void add(Map<String, Object> map) {
        Trace trace = new Trace(new Date(), map);
        synchronized (this.traces) {
            while (this.traces.size() >= this.capacity) {
                this.traces.remove(this.reverse ? this.capacity - 1 : 0);
            }
            if (this.reverse) {
                this.traces.add(0, trace);
            }
            else {
                this.traces.add(trace);
            }
        }
    }

}
```

可以看到，默认的 Trace 跟踪信息存储并没有用到特别的数据库或消息系统，而是采用了内存存储的方式。如上代码所示，通过 LinkedList<Trace>集合和 capacity 属性的定义，在 add(Map<String, Object> map)函数中进行循环存储，所以默认的 Trace 跟踪实现只能存储和查询最近的 100 条跟踪信息。

那么跟踪事件都记录了哪些内容呢？我们继续看 TraceListener 中 getSentTrace 和 getReceivedTrace 的具体实现：

```
public class TraceListener {
    ...
    protected Map<String, Object> getSentTrace(SentApplicationEvent event) {
        Map<String, Object> map = new LinkedHashMap<String, Object>();
        map.put("signal", "spring.cloud.bus.sent");
        map.put("type", event.getType().getSimpleName());
        map.put("id", event.getId());
        map.put("origin", event.getOriginService());
        map.put("destination", event.getDestinationService());
        if (log.isDebugEnabled()) {
            log.debug(map);
        }
        return map;
    }

    protected Map<String, Object> getReceivedTrace(AckRemoteApplicationEvent event) {
        Map<String, Object> map = new LinkedHashMap<String, Object>();
        map.put("signal", "spring.cloud.bus.ack");
        map.put("event", event.getEvent().getSimpleName());
```

```
        map.put("id", event.getAckId());
        map.put("origin", event.getOriginService());
        map.put("destination", event.getAckDestinationService());
        if (log.isDebugEnabled()) {
            log.debug(map);
        }
        return map;
    }
}
```

可以看到，这两个函数会收集关于发送和接收到的 Ack 事件信息，并且两个函数获得的内容就是事件定义相关的一些属性，看到这里大家是否感觉似曾相识？是的，这些信息与之前我们通过 Kafka 的控制台工具获取的消息内容非常类似。既然 Spring Cloud Bus 已经提供了 Trace 跟踪信息的监听和记录，我们不妨尝试使用一下。要开启该功能非常简单，只需在配置文件中将下面的属性设置为 true 即可：

```
spring.cloud.bus.trace.enabled=true
```

通过请求配置主机的/trace 接口，比如 http://localhost:7002/trace，可以获得如下信息，

```
[
  {
    "timestamp": 1475129670494,
    "info": {
      "signal": "spring.cloud.bus.ack",
      "event": "RefreshRemoteApplicationEvent",
      "id": "84ecdf83-a904-41bc-a34d-62680ccf35d7",
      "origin": "config-server:7001",
      "destination": "*:**"
    }
  },
  {
    "timestamp": 1475129670475,
    "info": {
      "signal": "spring.cloud.bus.sent",
      "type": "RefreshRemoteApplicationEvent",
      "id": "84ecdf83-a904-41bc-a34d-62680ccf35d7",
      "origin": "config-server:7001",
      "destination": "*:**"
    }
  },
  {
    "timestamp": 1475129670473,
    "info": {
      "signal": "spring.cloud.bus.ack",
```

```
        "event": "RefreshRemoteApplicationEvent",
        "id": "84ecdf83-a904-41bc-a34d-62680ccf35d7",
        "origin": "didispace:7002",
        "destination": "*:**"
      }
    }
  ]
```

与我们分析的内容一样，该请求返回了最近的 Send 和 Ack 消息内容。

如果希望针对 `AckRemoteApplicationEvent` 或是 `SentApplicationEvent` 做一些特殊处理，我们也可以通过`@EventListener` 注解在应用程序中编写自己的处理逻辑，或者重写 `TraceRepository` 来改造跟踪的存储等。

原则上每一个消息总线上的应用都可以用来跟踪 Ack 消息，但是大多数情况下我们把这个任务交给更核心的服务（比如特定的监控服务），这样在该服务中我们就能在 Ack 消息中实现更复杂的逻辑进行预警和善后工作。

事件发布

通过上面的分析，我们已经了解了 Spring Cloud Bus 中事件以及监听器的定义，下面来看看这些事件是如何发布给监听器进行处理的。

在 `org.springframework.cloud.bus` 包下，我们可以找到关于 Spring Cloud Bus 启动时加载的一些基础类和接口，包括自动化配置类 `BusAutoConfiguration`、属性定义类 `BusProperties` 等。我们可以从 Spring Cloud Bus 的自动化配置类中看看它在启动的时候都加载了什么内容：

```
@Configuration
@ConditionalOnBusEnabled
@EnableBinding(SpringCloudBusClient.class)
@EnableConfigurationProperties(BusProperties.class)
public class BusAutoConfiguration implements ApplicationEventPublisherAware {

    public static final String BUS_PATH_MATCHER_NAME = "busPathMatcher";

    @Autowired
    @Output(SpringCloudBusClient.OUTPUT)
    private MessageChannel cloudBusOutboundChannel;

    @Autowired
    private ServiceMatcher serviceMatcher;

    @Autowired
    private ChannelBindingServiceProperties bindings;
```

```
    @Autowired
    private BusProperties bus;

    private ApplicationEventPublisher applicationEventPublisher;
    ...
}
```

我们先来看看在该自动化配置类中，都定义了哪些成员。

- MessageChannel cloudBusOutboundChannel：该接口定义了发送消息的抽象方法。
- ServiceMatcher serviceMatcher：该对象中提供了下面两个重要函数，用来判断事件的来源服务是否为自己，以及判断目标是否为自己，以此作为依据是否要响应消息进行事件的处理。

```
public boolean isFromSelf(RemoteApplicationEvent event) {
    String originService = event.getOriginService();
    String serviceId = getServiceId();
    return this.matcher.match(originService, serviceId);
}

public boolean isForSelf(RemoteApplicationEvent event) {
    String destinationService = event.getDestinationService();
    return (destinationService == null || destinationService.trim().isEmpty() ||
this.matcher.match(destinationService, getServiceId()));
}
```

- ChannelBindingServiceProperties bindings：定义了消息服务的绑定属性。
- BusProperties bus：该对象定义了 Spring Cloud Bus 的属性，具体如下所示。

```
@ConfigurationProperties("spring.cloud.bus")
public class BusProperties {

    private Env env = new Env();
    private Refresh refresh = new Refresh();
    private Ack ack = new Ack();
    private Trace trace = new Trace();
    private String destination = "springCloudBus";
    private boolean enabled = true;
    ...
}
```

从中可以看到，Spring Cloud Bus 的属性前缀使用了 spring.cloud.bus。

destination 和 enabled 属性分别定义了默认的队列（Queue）或主题（Topic）是否连接到消息总线，所以我们可以通过 spring.cloud.bus.destination 来修改消息总线使用的队列或主题名称，以及使用 spring.cloud.bus.enabled 属性来设置应用是否要连接到消息总线上。

另外，在该配置类中为 Env、Refresh、Ack、Trace 4 种已经实现的事件分别创建了配置对象，这些配置类都是 BusProperties 的内部类。从下面的源码中，我们可以看到对于这 4 种事件，Env、Refresh、Ack 均是默认开启的，只有 Trace 事件需要通过修改配置来开启，就如之前我们介绍“事件跟踪”的时候配置 spring.cloud.bus.trace.enabled=true 属性那样。

```
public static class Env {

    private boolean enabled = true;
    ...
}

public static class Refresh {

    private boolean enabled = true;
    ...
}

public static class Ack {

    private boolean enabled = true;
    private String destinationService;
    ...
}

public static class Trace {

    private boolean enabled = false;
    ...
}
```

- ApplicationEventPublisher：Spring 事件模型中用来发布事件的接口，也就是我们之前介绍的事件以及监听的桥梁。

除了定义的这些成员变量之外，还能看到这里定义了两个监听方法 acceptLocal 和 acceptRemote。

其中，acceptLocal 方法如下所示，它通过@EventListener(classes =

RemoteApplicationEvent.class)注解修饰。之前已经介绍过该注解，可以将该函数理解为对 RemoteApplicationEvent 事件的监听器，但是在其实现中并非所有的 RemoteApplicationEvent 事件都会处理。根据 if 中的条件，可以看到在该监听处理中，只对事件来源是自己并且事件类型不是 AckRemoteApplicationEvent 的内容进行后续的处理，而后续的处理就是通过消息管道将该事件发送出去。所以，该监听器的功能就是监听本地事件来进行消息的发送。

```
    @EventListener(classes = RemoteApplicationEvent.class)
    public void acceptLocal(RemoteApplicationEvent event) {
        if (this.serviceMatcher.isFromSelf(event)
                && !(event instanceof AckRemoteApplicationEvent)) {
            this.cloudBusOutboundChannel.send(MessageBuilder.withPayload(event).
build());
        }
    }
```

再来看看 acceptRemote 方法。该方法中使用了@StreamListener 注解修饰，该注解的作用是将该函数注册为消息代理上数据流的事件监听器，注解中的属性值 SpringCloudBusClient.INPUT 指定了监听的通道名。同时，回头看该函数所在类的定义，使用了@EnableBinding 注解，该注解用来实现与消息代理的连接，注解中的属性值 SpringCloudBusClient.class 声明了输入和输出通道的定义（这部分内容源自 Spring Cloud Stream，在下一章中，我们会对这些内容做详细介绍，这里我们只需理解它用来绑定消息代理的输入与输出，以实现向消息总线上发送和接收消息即可）。

```
    @StreamListener(SpringCloudBusClient.INPUT)
    public void acceptRemote(RemoteApplicationEvent event) {
        if (event instanceof AckRemoteApplicationEvent) {
            if (this.bus.getTrace().isEnabled() && !this.serviceMatcher.isFromSelf(event)
                    && this.applicationEventPublisher != null) {
                this.applicationEventPublisher.publishEvent(event);
            }
            // If it's an ACK we are finished processing at this point
            return;
        }
        if (this.serviceMatcher.isForSelf(event)
                && this.applicationEventPublisher != null) {
            if (!this.serviceMatcher.isFromSelf(event)) {
                this.applicationEventPublisher.publishEvent(event);
            }
            if (this.bus.getAck().isEnabled()) {
                AckRemoteApplicationEvent ack = new AckRemoteApplicationEvent(this,
                        this.serviceMatcher.getServiceId(),
                        this.bus.getAck().getDestinationService(),
```

```
                event.getDestinationService(), event.getId(), event.getClass());
            this.cloudBusOutboundChannel
                    .send(MessageBuilder.withPayload(ack).build());
            this.applicationEventPublisher.publishEvent(ack);
        }
    }
    if (this.bus.getTrace().isEnabled() && this.applicationEventPublisher != null) {
        // We are set to register sent events so publish it for local consumption,
        // irrespective of the origin
        this.applicationEventPublisher.publishEvent(new SentApplicationEvent(this,
                event.getOriginService(), event.getDestinationService(),
                event.getId(), event.getClass()));
    }
}
```

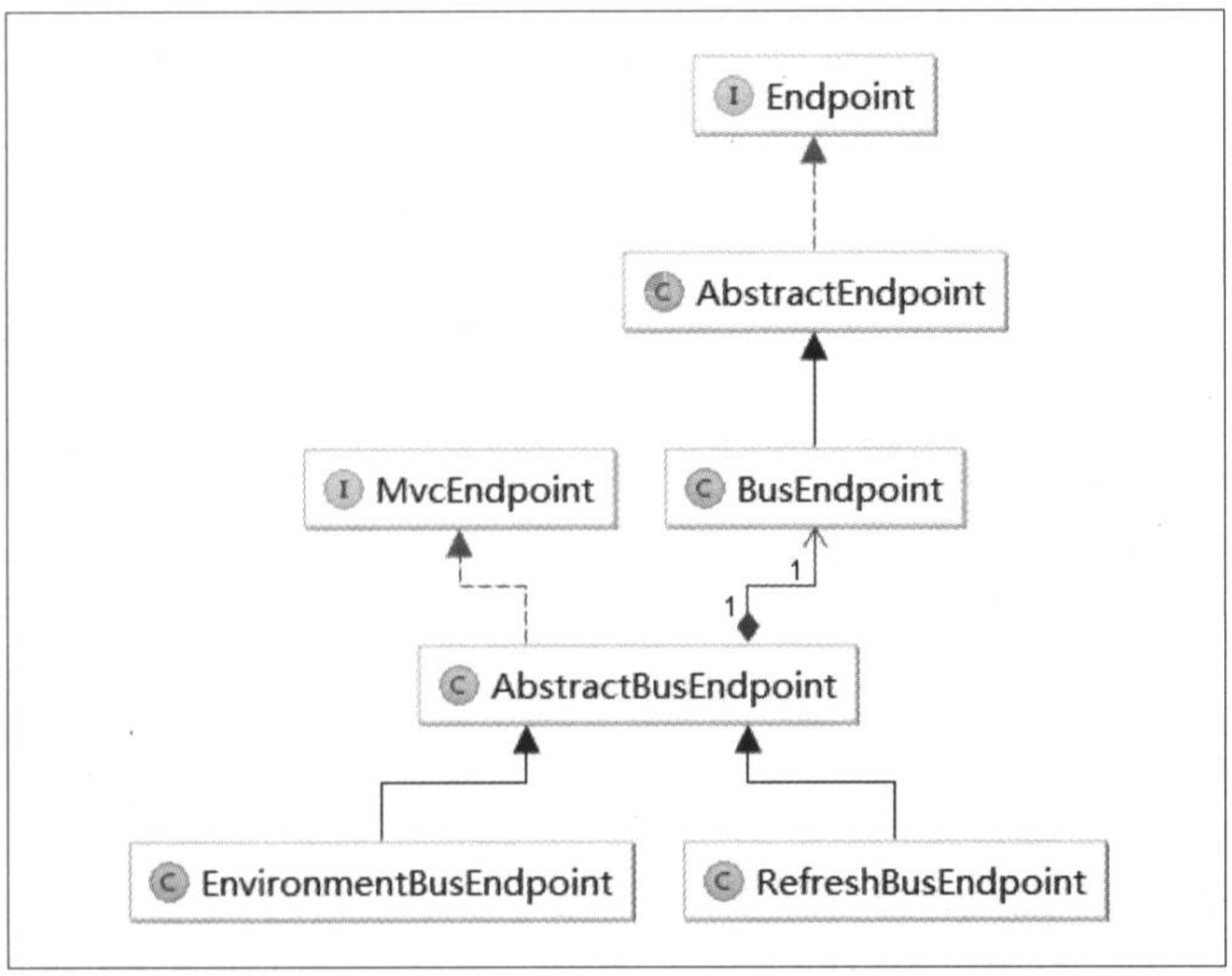
通过上面的分析，我们已经可以知道 Spring Cloud Bus 通过 acceptRemote 方法来监听消息代理的输入通道，并根据事件类型和配置内容来确定是否要发布事件给我们之前分析的几个事件监听器来对事件做具体的处理；而 acceptLocal 方法用来监听本地的事件，针对事件来源是自己，并且事件类型不是 AckRemoteApplicationEvent 的内容通过消息代理的输出通道发送到总线上去。

控制端点

在介绍了 Spring Cloud Bus 中实现的事件模型之后，我们已经知道每个节点是如何响应消息总线上的事件了。那么这些发送到消息总线上用来触发各个节点的事件处理的动作是如何实现的呢？回想一下之前在实现配置属性刷新时，我们在修改了 Git 仓库上的配置信息之后，往总线上的某个节点发送了一个请求/bus/refresh 来触发总线上的所有节点进行配置刷新；我们在连接到消息总线的应用启动时，也能在控制台中看到类似下面的输出：

```
    2016-09-30 11:05:13.037  INFO 18720 --- [          main]
o.s.b.a.e.mvc.EndpointHandlerMapping    : Mapped "{[/bus/refresh],methods=[POST]}"
onto public void
org.springframework.cloud.bus.endpoint.RefreshBusEndpoint.refresh(java.lang.String)
    2016-09-30 11:05:13.045  INFO 18720 --- [          main]
o.s.b.a.e.mvc.EndpointHandlerMapping    : Mapped "{[/bus/env],methods=[POST]}" onto
public void
org.springframework.cloud.bus.endpoint.EnvironmentBusEndpoint.env(java.util.Map<jav
a.lang.String, java.lang.String>,java.lang.String)
```

从上面的日志信息中可以看到，在 org.springframework.cloud.bus. endpoint 包下的 RefreshBusEndpoint 和 EnvironmentBusEndpoint 分别创建了两个控制端点：/bus/refresh 和/bus/env。通过整理 org.springframework.cloud.bus. endpoint 包下的内容，我们可以得到如下类图：



从图中可以发现，Spring Cloud Bus 中的 Endpoint 也是通过 spring-boot-actuator 模块来实现的。下面，简单介绍一下 spring-boot-actuator 模块中的几个重要元素。

- Endpoint：该接口中定义了监控端点需要暴露的一些有用信息，比如，id、是否开启标识、是否开启敏感信息标识等。
- AbstractEndPoint：该抽象类是对 Endpoint 的基础实现，在该抽象类中引入了 Environment 接口对象，从而对接口暴露信息的控制可以通过配置文件的方式来控制。
- MvcEndpoint 接口：该接口定义了 Endpoint 接口在 MVC 层的策略。在这里可以通过使用 Spring MVC 的@RequestMapping 注解来定义端点暴露的接口地址。

下面我们来看看 Spring Cloud Bus 是如何扩展 Endpoint 的。

- BusEndpoint：该类继承自 AbstractEndPoint。从类上的注解@Configuration-Properties 配置可以知道，Spring Cloud Bus 实现的端点配置属性需要以 endpoints.bus 开头，通过该类的构造函数（配合 AbstractEndpoint 中的构造函数），我们可以知道默认 id 为 bus，并且端点默认敏感标识为 true：

```
@ConfigurationProperties(prefix = "endpoints.bus", ignoreUnknownFields = false)
public class BusEndpoint extends AbstractEndpoint<Collection<String>> {

    public BusEndpoint() {
        super("bus");
```

```
    }

    @Override
    public Collection<String> invoke() {
        return Collections.emptyList();
    }
}
public abstract class AbstractEndpoint<T> implements Endpoint<T>, EnvironmentAware {
    ...
    public AbstractEndpoint(String id) {
        this(id, true);
    }
    public AbstractEndpoint(String id, boolean sensitive) {
        this.id = id;
        this.sensitiveDefault = sensitive;
    }
    ...
}
```

- AbstractBusEndpoint 类是实现 Spring Cloud Bus 中端点的重要基类，它实现了 MvcEndpoint 接口来暴露 MVC 层的接口，同时关联了 BusEndpoint 对象。通过下面的源码，我们可以看到，getPath、isSensitive 和 getEndpointType 都是委托给 BusEndpoint 来获取的，从而实现通过 Environment 配置接口。

```
public class AbstractBusEndpoint implements MvcEndpoint {

    private ApplicationEventPublisher context;
    private BusEndpoint delegate;
    private String appId;

    public AbstractBusEndpoint(ApplicationEventPublisher context, String appId,
            BusEndpoint busEndpoint) {
        this.context = context;
        this.appId = appId;
        this.delegate = busEndpoint;
    }

    protected String getInstanceId() {
        return this.appId;
    }
```

```
    protected void publish(ApplicationEvent event) {
        context.publishEvent(event);
    }

    @Override
    public String getPath() {
        return "/" + this.delegate.getId();
    }

    @Override
    public boolean isSensitive() {
        return this.delegate.isSensitive();
    }

    @Override
    @SuppressWarnings("rawtypes")
    public Class<? extends Endpoint> getEndpointType() {
        return this.delegate.getClass();
    }
}
```

默认实现的几个端点都继承自 AbstractBusEndpoint 类来实现 MVC 层接口的暴露和配置，下面我们来看看具体的两个实现端点。

- 实现配置刷新的端点 RefreshBusEndpoint 类。通过下面的源码，我们可以看到，在该类中定义了 refresh 的 POST 请求，由于在 BusEndpoint 默认构造时 id 为 bus，而 AbstractBusEndpoint 中 getPath 函数通过 BusEndpoint 中的 id 拼接而成，所以对于 RefreshBusEndpoint 中 refresh 请求的完整路径为 /bus/refresh。同时，该请求通过@RequestParam 注解还定义了一个可选的参数 destination，正如在之前的示例中介绍的，该参数用于指定刷新的服务实例。在请求处理部分直接调用了父类中的 publish 函数将 RefreshRemoteApplicationEvent 事件发布出来，实现在总线上发布消息的功能。

```
public class RefreshBusEndpoint extends AbstractBusEndpoint {

    public RefreshBusEndpoint(ApplicationEventPublisher context, String id,
            BusEndpoint delegate) {
        super(context, id, delegate);
    }
```

```
        @RequestMapping(value = "refresh", method = RequestMethod.POST)
        @ResponseBody
        public void refresh(
                @RequestParam(value = "destination", required = false) String destination) {
            publish(new RefreshRemoteApplicationEvent(this, getInstanceId(),
destination));
        }
    }
```

- EnvironmentBusEndpoint 的实现与 RefreshBusEndpoint 类似，通过暴露 /bus/env 的 POST 请求接口，并提供了 Map 类型的 params 参数设定需要更新的配置信息，以及同 refresh 接口一样的 destination 参数指定需要更新的服务实例，来触发环境参数更新的消息总线控制。

```
public class EnvironmentBusEndpoint extends AbstractBusEndpoint {

    public EnvironmentBusEndpoint(ApplicationEventPublisher context, String id,
            BusEndpoint delegate) {
        super(context, id, delegate);
    }

    @RequestMapping(value = "env", method = RequestMethod.POST)
    @ResponseBody
    public void env(@RequestParam Map<String, String> params,
            @RequestParam(value = "destination", required = false) String destination) {
        publish(new EnvironmentChangeRemoteApplicationEvent(this, getInstanceId(),
                destination, params));
    }
}
```

其他消息代理的支持

由于目前版本的 Spring Cloud Bus 只实现了 RabbitMQ 和 Kafka 的封装，虽然大部分情况下，这两个产品的特性已经涵盖我们大部分的业务场景，但是由于一些特殊需求或是遗留系统等其他因素，有些团队不得不使用其他的消息代理，这个时候我们就需要扩展消息代理的支持。实际上，通过之前对源码的分析，我们可以看到，Spring Cloud Bus 在绑定具体消息代理的输入与输出通道时均使用了抽象接口的方式，所以真正的实现来自于 spring-cloud-starter-bus-amqp 和 spring-cloud-starter-bus-kafka 的依赖。

我们可以查看 spring-cloud-starter-bus-amqp 和 spring-cloud-starter-

bus-kafka 的依赖，可以看到它们分别依赖了 spring-cloud-starter-stream-rabbit 和 spring-cloud-starter-stream-kafka，真正实现与这些消息代理进行交互操作的是 Spring Cloud Stream。所以，我们在本章中使用的所有 Spring Cloud Bus 的消息通信基础实际上都是由 Spring Cloud Stream 所提供的。一定程度上，我们可以将 Spring Cloud Bus 理解为是一个使用了 Spring Cloud Stream 构建的上层应用。由于 Spring Cloud Stream 为了让开发者屏蔽各个消息代理之间的差异，将来能够方便地切换不同的消息代理而不影响业务程序，所以在业务程序与消息代理之间定义了一层抽象，称为绑定器（Binder）。我们在整合 RabbitMQ 和 Kafka 的时候就是分别引入了它们各自的绑定器实现，可以回想一下之前的实现内容，不论使用 RabbitMQ 还是 Kafka 实现，在程序上其实没有任何变化，变化的只是对绑定器的配置。所以，当我们要在其他消息代理上使用 Spring Cloud Bus 消息总线时，只需要去实现一套指定消息代理的绑定器即可。

第 10 章

消息驱动的微服务：Spring Cloud Stream

Spring Cloud Stream 是一个用来为微服务应用构建消息驱动能力的框架。它可以基于 Spring Boot 来创建独立的、可用于生产的 Spring 应用程序。它通过使用 Spring Integration 来连接消息代理中间件以实现消息事件驱动。Spring Cloud Stream 为一些供应商的消息中间件产品提供了个性化的自动化配置实现，并且引入了发布-订阅、消费组以及分区这三个核心概念。简单地说，Spring Cloud Stream 本质上就是整合了 Spring Boot 和 Spring Integration，实现了一套轻量级的消息驱动的微服务框架。通过使用 Spring Cloud Stream，可以有效简化开发人员对消息中间件的使用复杂度，让系统开发人员可以有更多的精力关注于核心业务逻辑的处理。由于 Spring Cloud Stream 基于 Spring Boot 实现，所以它秉承了 Spring Boot 的优点，自动化配置的功能可帮助我们快速上手使用，但是到目前为止，Spring Cloud Stream 只支持下面两个著名的消息中间件的自动化配置：

- RabbitMQ
- Kafka

对于这两个消息中间件的介绍，我们在上一章消息总线的内容中已有过一些基础的介绍，更多关于这两个消息中间件的高级使用和性能配置等内容不在本书的关注范围之内，读者在实际应用中可以查看它们的官方文档或是其他专业书籍进行进一步学习和实践。

快速入门

下面我们通过构建一个简单的示例来对 Spring Cloud Stream 有一个初步认识。该示例的主要目标是构建一个基于 Spring Boot 的微服务应用，这个微服务应用将通过使用消息中

间件 RabbitMQ 来接收消息并将消息打印到日志中。所以，在进行下面的步骤之前请先确认已经在本地安装了 RabbitMQ，具体安装步骤可根据在上一章中的介绍来操作。

- 创建一个基础的 Spring Boot 工程，命名为 `stream-hello`。
- 编辑 `pom.xml` 中的依赖关系，引入 Spring Cloud Stream 对 RabbitMQ 的支持，具体如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
    </dependency>
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

- 创建用于接收来自 RabbitMQ 消息的消费者 SinkReceiver，具体如下：

```
@EnableBinding(Sink.class)
public class SinkReceiver {

    private static Logger logger = LoggerFactory.getLogger(HelloApplication.class);

    @StreamListener(Sink.INPUT)
    public void receive(Object payload) {
        logger.info("Received: " + payload);
    }

}
```

- 创建应用主类，这里同其他 Spring Boot 一样，没有什么特别之处，具体如下：

```
@SpringBootApplication
public class HelloApplication {

    public static void main(String[] args) {
        SpringApplication.run(HelloApplication.class, args);
    }

}
```

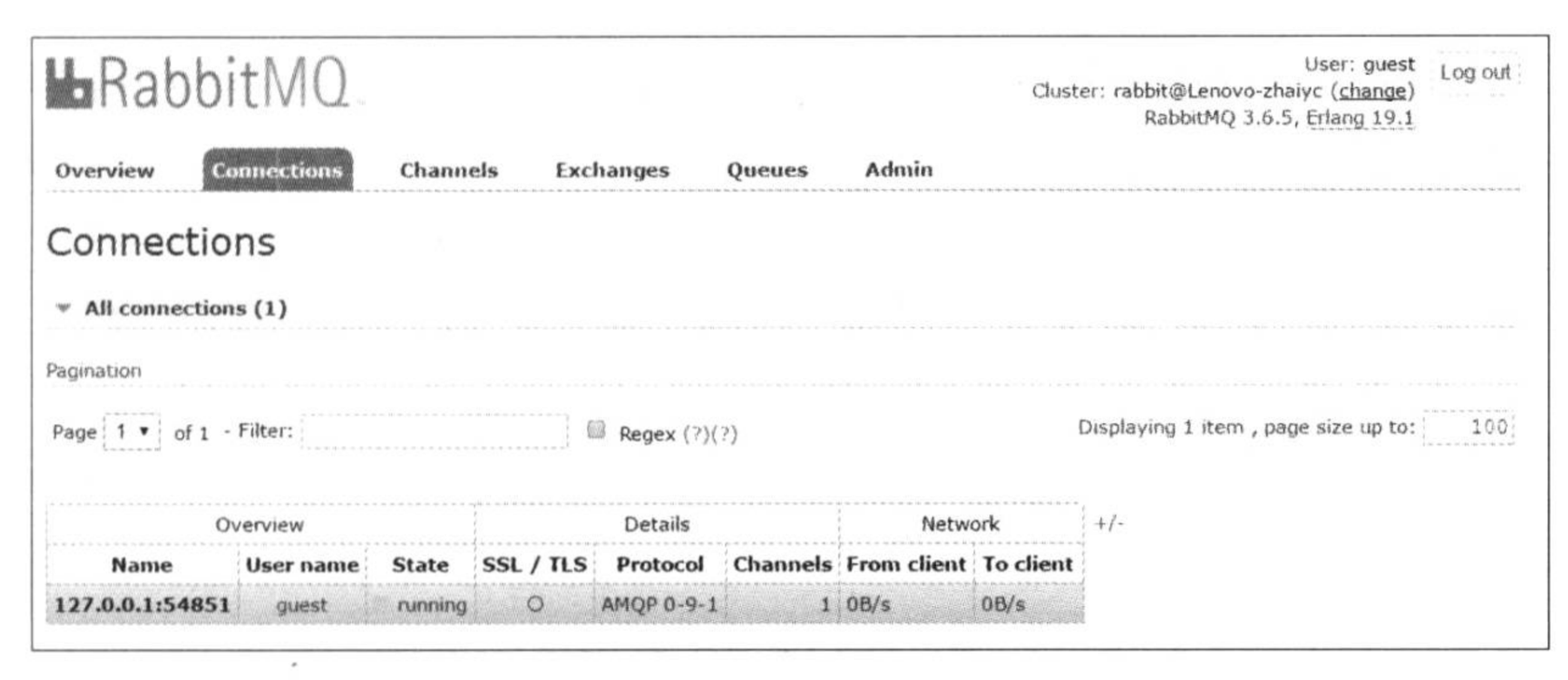
到这里，快速入门示例的编码任务已经完成了。下面我们分别启动 RabbitMQ 以及该 Spring Boot 应用，然后做下面的测试，看看它们是如何运作的。

- 先来看一下 Spring Boot 应用的启动日志。

```
...
    INFO 16272 --- [main] o.s.c.s.b.r.RabbitMessageChannelBinder   : declaring queue for
inbound: input.anonymous.Y8VsFILmSC27eS5StsXp6A, bound to: input
    INFO 16272 --- [main] o.s.a.r.c.CachingConnectionFactory       : Created new
connection: SimpleConnection@3c78e551 [delegate=amqp://guest@127.0.0.1:5672/]
    INFO 16272 --- [main] o.s.integration.channel.DirectChannel    : Channel
'input.anonymous.Y8VsFILmSC27eS5StsXp6A.bridge' has 1 subscriber(s).
    INFO 16272 --- [main] o.s.i.a.i.AmqpInboundChannelAdapter      : started
inbound.input.anonymous.Y8VsFILmSC27eS5StsXp6A
...
```

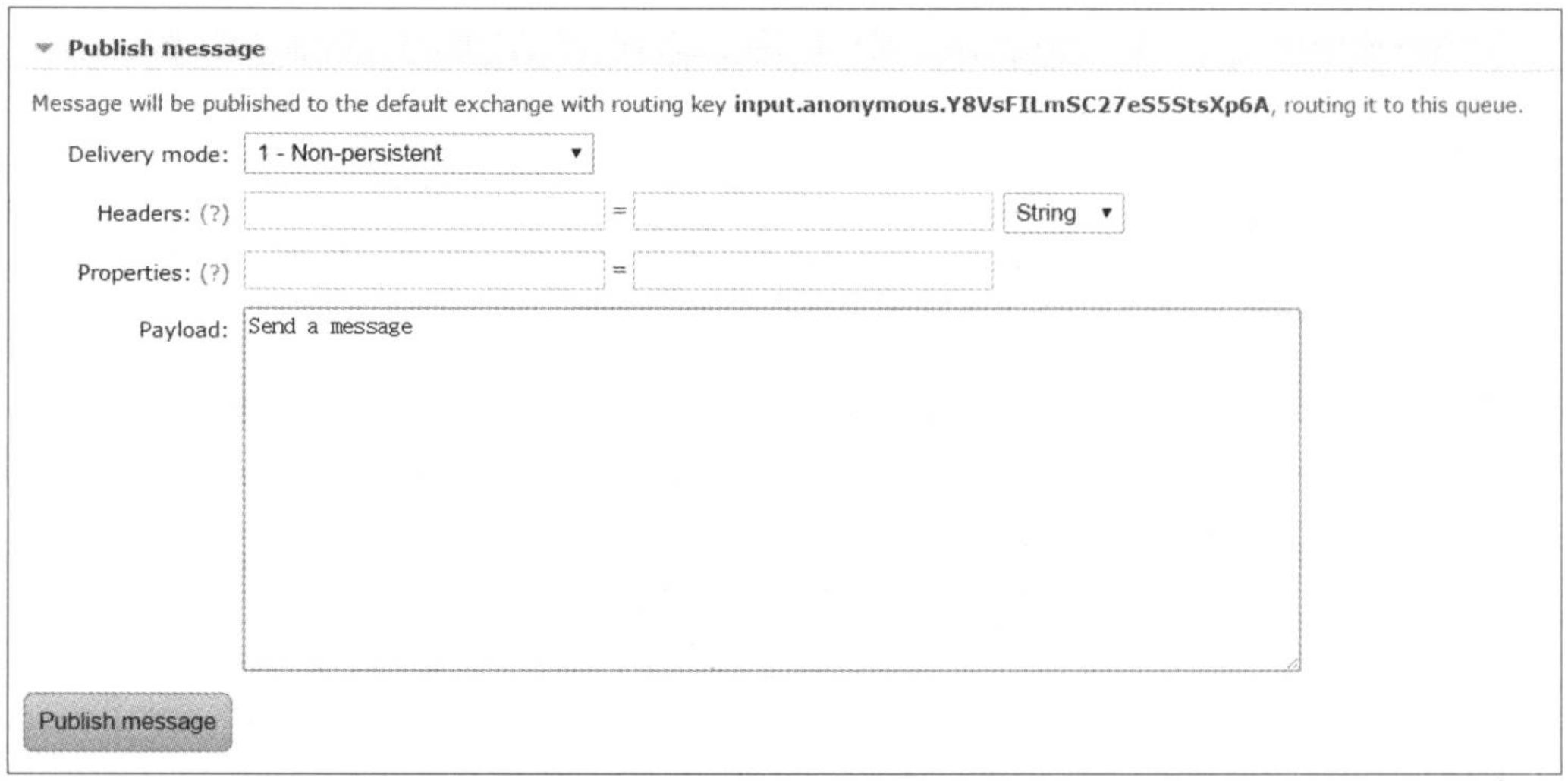
从上面的日志内容中，可以获得以下信息：

- 使用 guest 用户创建了一个指向 127.0.0.1:5672 位置的 RabbitMQ 连接，在 RabbitMQ 的控制台中我们也可以发现它。



- 声明了一个名为 `input.anonymous.Y8VsFILmSC27eS5StsXp6A` 的队列，并通过 `RabbitMessageChannelBinder` 将自己绑定为它的消费者。这些信息我们也能在 RabbitMQ 的控制台中发现它们。

下面我们可以在 RabbitMQ 的控制台中进入 input.anonymous.Y8VsFILmSC-27eS5StsXp6A 队列的管理页面，通过 Publish message 功能来发送一条消息到该队列中。



此时，我们可以在当前启动的 Spring Boot 应用程序的控制台中看到下面的内容：

```
    INFO 16272 --- [C27eS5StsXp6A-1] com.didispace.HelloApplication      : Received:
[B@7cba610e
```

可以发现在应用控制台中输出的内容就是 SinkReceiver 中的 receive 方法定义的，而输出的具体内容则来自消息队列中获取的对象。这里由于我们没有对消息进行序列化，所以输出的只是该对象的引用，在后面的章节中我们会详细介绍接收消息后的处理。

在顺利完成上面快速入门的示例后，我们简单解释一下上面的步骤是如何将我们的 Spring Boot 应用连接上 RabbitMQ 来消费消息以实现消息驱动业务逻辑的。

首先，我们对 Spring Boot 应用做的就是引入 spring-cloud-starter-stream-rabbit 依赖，该依赖包是 Spring Cloud Stream 对 RabbitMQ 支持的封装，其中包含了对 RabbitMQ 的自动化配置等内容。从下面它定义的依赖关系中，我们还可以知道它等价于 spring-cloud-stream-binder-rabbit 依赖。

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
    </dependency>
</dependencies>
```

接着，我们再来看看这里用到的几个 Spring Cloud Stream 的核心注解，它们都被定义

在 SinkReceiver 中。

- @EnableBinding，该注解用来指定一个或多个定义了@Input 或@Output 注解的接口，以此实现对消息通道（Channel）的绑定。在上面的例子中，我们通过@EnableBinding(Sink.class)绑定了 Sink 接口，该接口是 Spring Cloud Stream 中默认实现的对输入消息通道绑定的定义，它的源码如下：

```
public interface Sink {

    String INPUT = "input";

    @Input(Sink.INPUT)
    SubscribableChannel input();

}
```

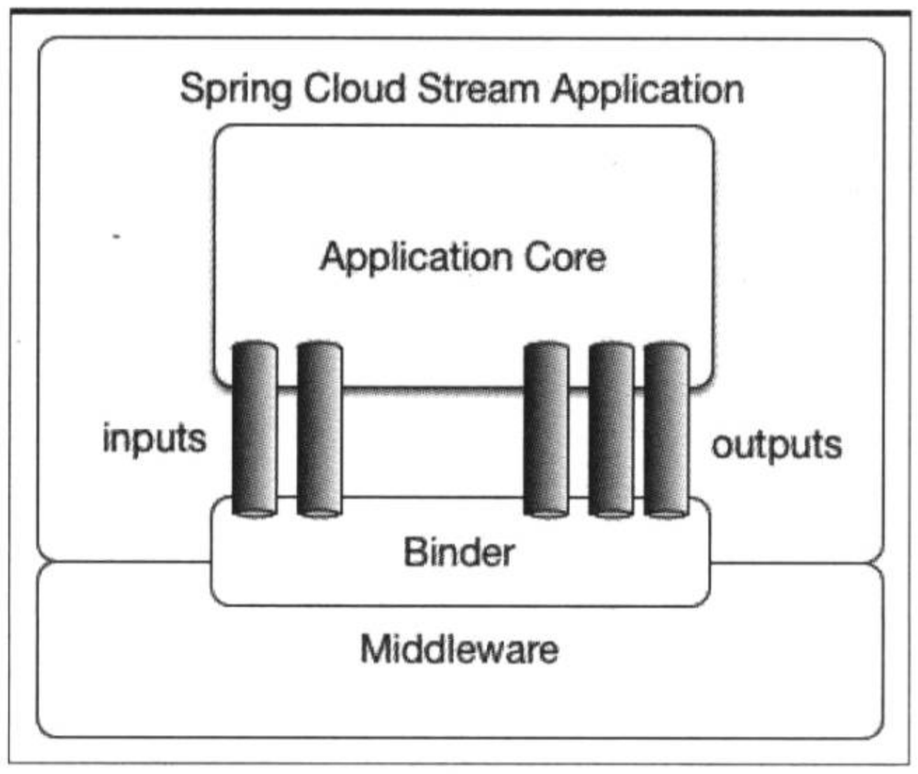
  它通过@Input 注解绑定了一个名为 input 的通道。除了 Sink 之外，Spring Cloud Stream 还默认实现了绑定 output 通道的 Source 接口，还有结合了 Sink 和 Source 的 Processor 接口，实际使用时我们也可以自己通过@Input 和@Output 注解来定义绑定消息通道的接口。当需要为@EnableBinding 指定多个接口来绑定消息通道的时候，可以这样定义：@EnableBinding(value = {Sink.class, Source.class})。
- @StreamListener：如果已经读过上一章 Spring Cloud Bus 的源码分析，相信对该注解不会感到陌生。它主要定义在方法上，作用是将被修饰的方法注册为消息中间件上数据流的事件监听器，注解中的属性值对应了监听的消息通道名。在上面的例子中，我们通过@StreamListener(Sink.INPUT)注解将 receive 方法注册为 input 消息通道的监听处理器，所以当我们在 RabbitMQ 的控制页面中发布消息的时候，receive 方法会做出对应的响应动作。

核心概念

通过上一节介绍的快速入门示例，相信大家对 Spring Cloud Stream 的工作模式已经有了一些基础概念，比如输入、输出通道的绑定，通道消息事件的监听等。在本节中，我们将详细介绍在 Spring Cloud Stream 中是如何通过定义一些基础概念来对各种不同的消息中间件做抽象的。

下图是官方文档中 Spring Cloud Stream 应用模型的结构图。从中我们可以看到，Spring Cloud Stream 构建的应用程序与消息中间件之间是通过绑定器 Binder 相关联的，绑定器对于应用程序而言起到了隔离作用，它使得不同消息中间件的实现细节对应用程序来说是

透明的。所以对于每一个 Spring Cloud Stream 的应用程序来说，它不需要知晓消息中间件的通信细节，它只需知道 Binder 对应程序提供的抽象概念来使用消息中间件来实现业务逻辑即可，而这个抽象概念就是在快速入门中我们提到的消息通道：Channel。如下图所示，在应用程序和 Binder 之间定义了两条输入通道和三条输出通道来传递消息，而绑定器则是作为这些通道和消息中间件之间的桥梁进行通信。



绑定器

Binder 绑定器是 Spring Cloud Stream 中一个非常重要的概念。在没有绑定器这个概念的情况下，Spring Boot 应用要直接与消息中间件进行信息交互的时候，由于各消息中间件构建的初衷不同，所以它们在实现细节上会有较大的差异，这使得我们实现的消息交互逻辑就会非常笨重，因为对具体的中间件实现细节有太重的依赖，当中间件有较大的变动升级或是更换中间件的时候，我们就需要付出非常大的代价来实施。

通过定义绑定器作为中间层，完美地实现了应用程序与消息中间件细节之间的隔离。通过向应用程序暴露统一的 Channel 通道，使得应用程序不需要再考虑各种不同的消息中间件的实现。当需要升级消息中间件，或是更换其他消息中间件产品时，我们要做的就是更换它们对应的 Binder 绑定器而不需要修改任何 Spring Boot 的应用逻辑。这一点在上一章实现消息总线时，从 RabbitMQ 切换到 Kafka 的过程中，已经能够让我们体验到这一好处。

目前版本的 Spring Cloud Stream 为主流的消息中间件产品 RabbitMQ 和 Kafka 提供了默认的 Binder 实现，在快速入门的例子中，我们就使用了 RabbitMQ 的 Binder。另外，Spring Cloud Stream 还实现了一个专门用于测试的 TestSupportBinder，开发者可以直接使用它来对通道的接收内容进行可靠的测试断言。如果要使用除了 RabbitMQ 和 Kafka 以外的消息中间件的话，我们也可以通过使用它所提供的扩展 API 来实现其他中间件的 Binder。

仔细的读者可能已经发现，我们在快速入门示例中，并没有使用 application.

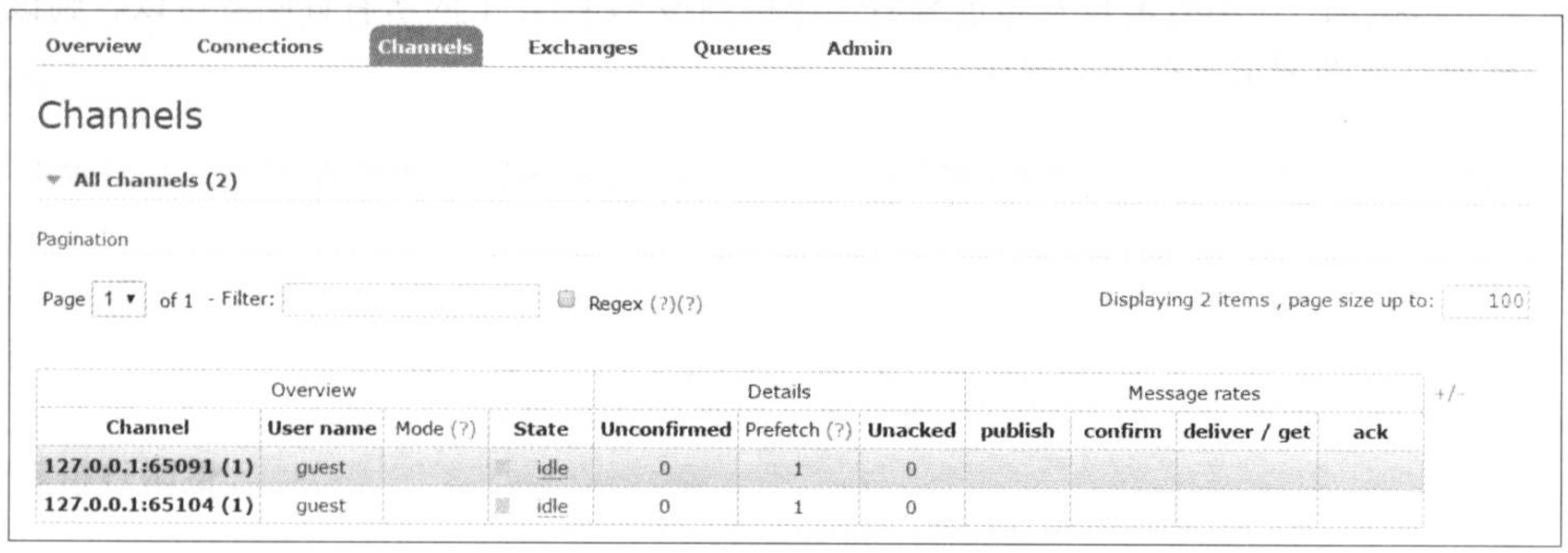
properties 或是 application.yml 来做任何属性设置。那是因为它也秉承了 Spring Boot 的设计理念，提供了对 RabbitMQ 默认的自动化配置。当然，我们也可以通过 Spring Boot 应用支持的任何方式来修改这些配置，比如，通过应用程序参数、环境变量、application.properties 或是 application.yml 配置文件等。比如，下面就是通过配置文件来对 RabbitMQ 的连接信息以及 input 通道的主题进行配置的示例：

```
spring.cloud.stream.bindings.input.destination=raw-sensor-data

spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=springcloud
spring.rabbitmq.password=123456
```

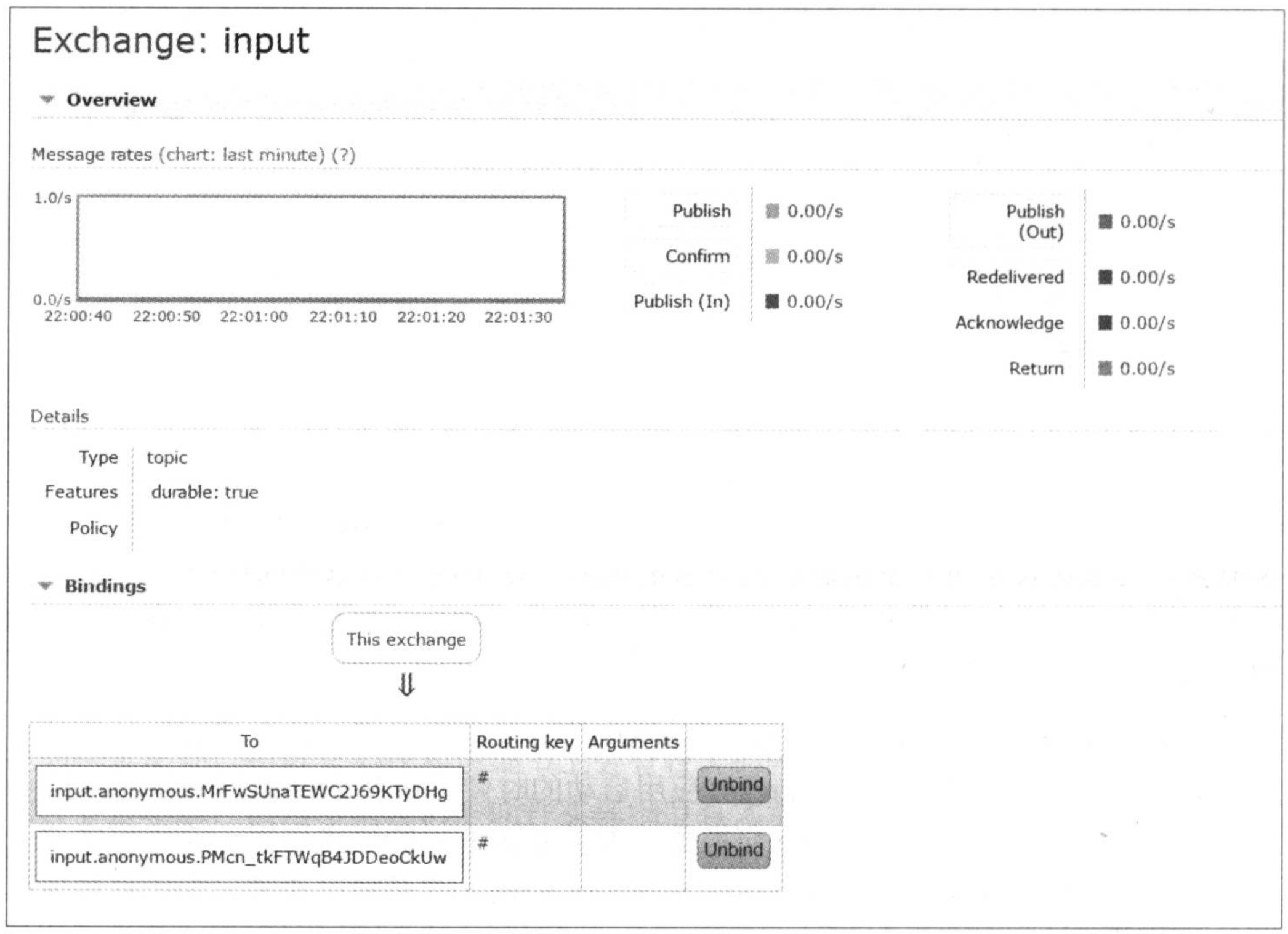
发布-订阅模式

Spring Cloud Stream 中的消息通信方式遵循了发布-订阅模式，当一条消息被投递到消息中间件之后，它会通过共享的 Topic 主题进行广播，消息消费者在订阅的主题中收到它并触发自身的业务逻辑处理。这里所提到的 Topic 主题是 Spring Cloud Stream 中的一个抽象概念，用来代表发布共享消息给消费者的地方。在不同的消息中间件中，Topic 可能对应不同的概念，比如，在 RabbitMQ 中，它对应 Exchange，而在 Kakfa 中则对应 Kafka 中的 Topic。

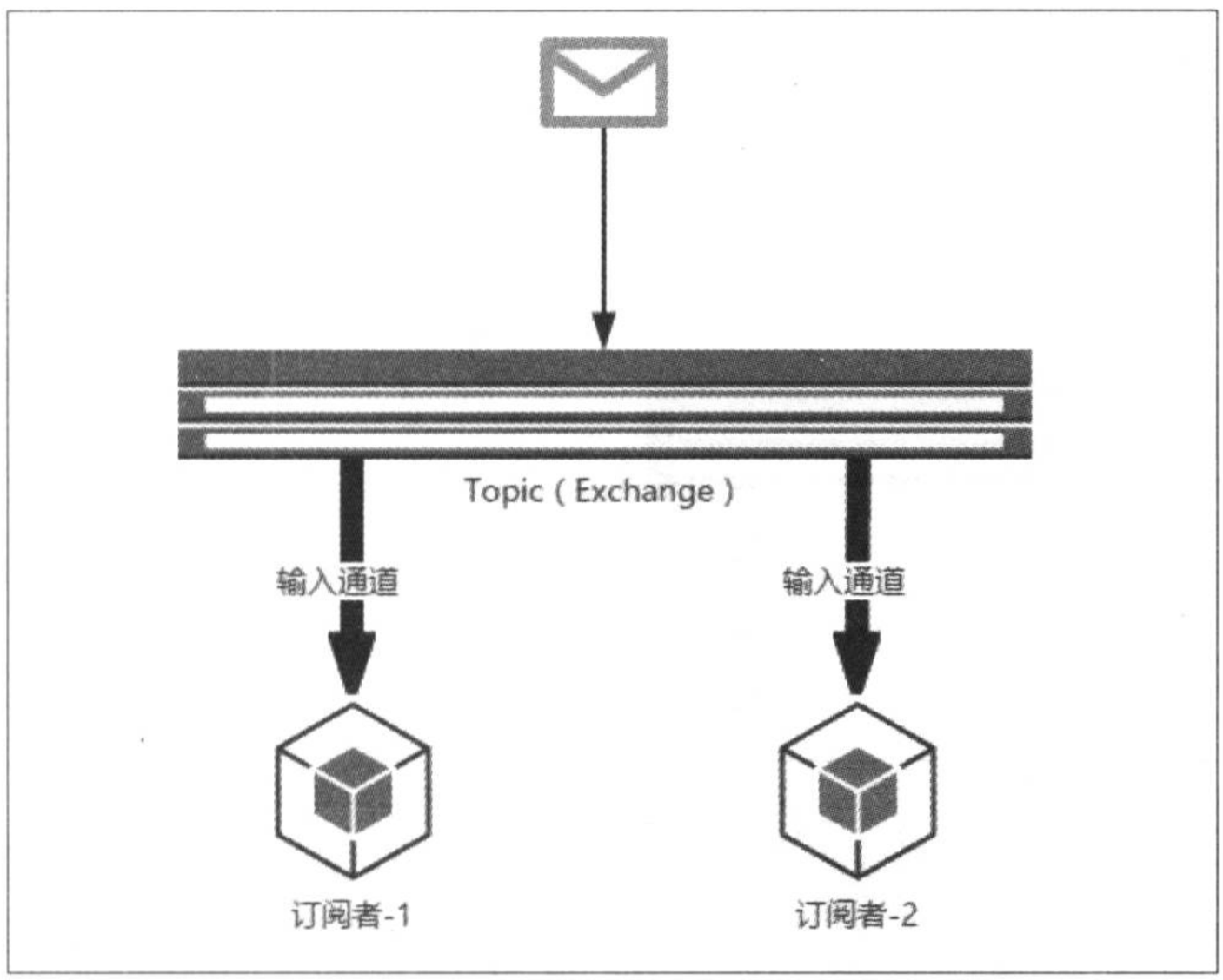
在快速入门的示例中，我们通过 RabbitMQ 的 Channel 发布消息给我们编写的应用程序消费，而实际上 Spring Cloud Stream 应用启动的时候，在 RabbitMQ 的 Exchange 中也创建了一个名为 input 的 Exchange 交换器，由于 Binder 的隔离作用，应用程序并无法感知它的存在，应用程序只知道自己指向 Binder 的输入或是输出通道。为了直观地感受在发布-订阅模式中，消息是如何被分发到多个订阅者的，我们可以使用快速入门的例子，通过命令行的方式启动两个不同端口的进程。此时，我们在 RabbitMQ 控制页面的 Channels 选项卡中看到如下图所示的两个消息通道，它们分别绑定了启动的两个应用程序。



而在 Exchanges 选项卡中，我们还能找到名为 input 的交换器，单击进入可以看到如下图所示的详情页面。Bindings 栏中的内容就是两个应用程序绑定通道中的消息队列，我们可以通过 Exchange 页面的 Publish Message 来发布消息，此时可以发现两个启动的应用程序都输出了消息内容。



下图总结了我们上面所做尝试的基础结构。启动的两个应用程序分别是“订阅者-1”和“订阅者-2”，它们都建立了一条输入通道绑定到同一个 Topic（RabbitMQ 的 Exchange）上。当该 Topic 中有消息发布进来后，连接到该 Topic 上的所有订阅者可以收到该消息并根据自身的需求进行消费操作。

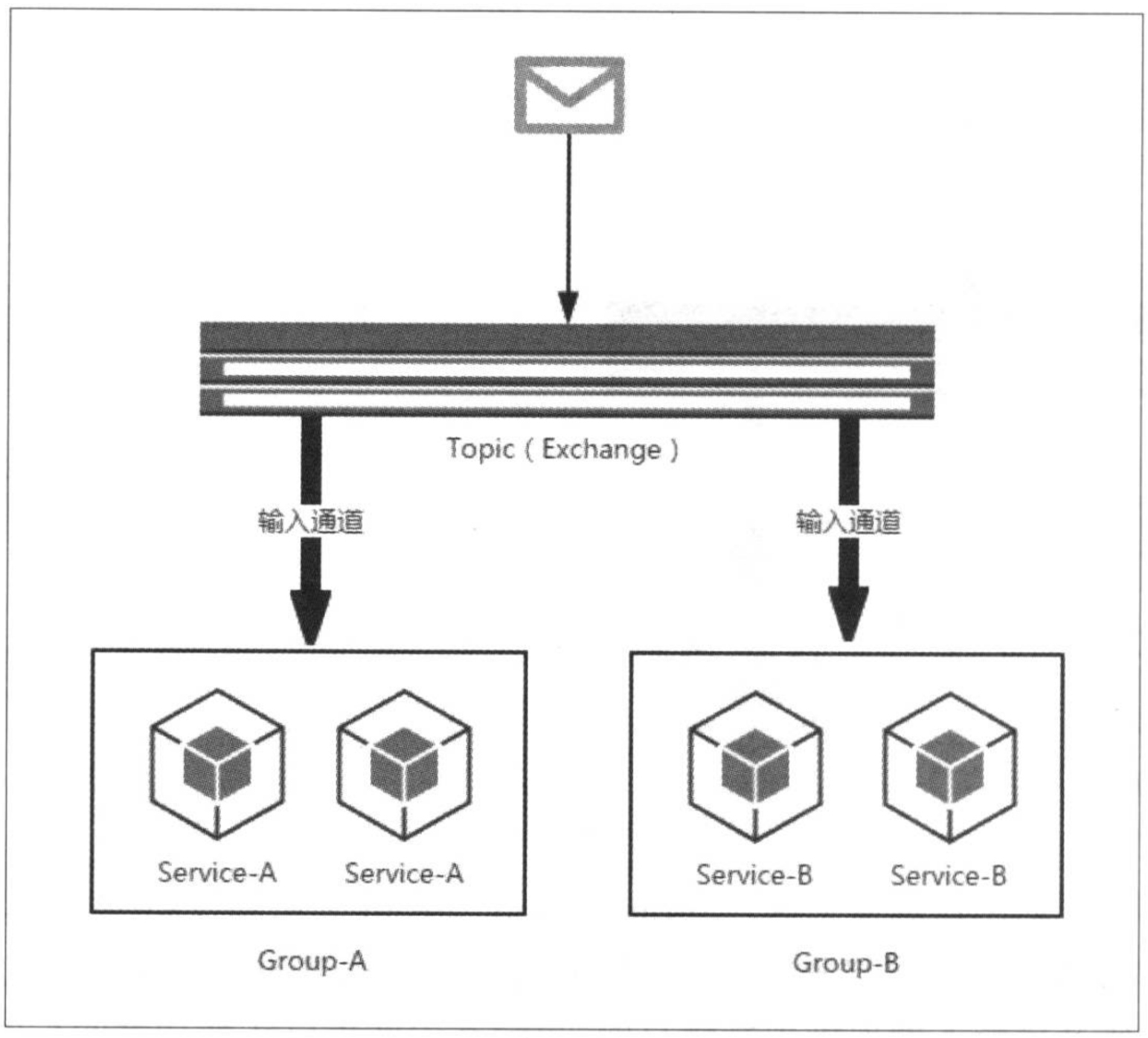


相对于点对点队列实现的消息通信来说，Spring Cloud Stream 采用的发布-订阅模式可以有效降低消息生产者与消费者之间的耦合。当需要对同一类消息增加一种处理方式时，只需要增加一个应用程序并将输入通道绑定到既有的 `Topic` 中就可以实现功能的扩展，而不需要改变原来已经实现的任何内容。

消费组

虽然 Spring Cloud Stream 通过发布-订阅模式将消息生产者与消费者做了很好的解耦，基于相同主题的消费者可以轻松地进行扩展，但是这些扩展都是针对不同的应用实例而言的。在现实的微服务架构中，我们的每一个微服务应用为了实现高可用和负载均衡，实际上都会部署多个实例。在很多情况下，消息生产者发送消息给某个具体微服务时，只希望被消费一次，按照上面我们启动两个应用的例子，虽然它们同属一个应用，但是这个消息出现了被重复消费两次的情况。为了解决这个问题，在 Spring Cloud Stream 中提供了消费组的概念。

如果在同一个主题上的应用需要启动多个实例的时候，我们可以通过 `spring.cloud.stream.bindings.input.group` 属性为应用指定一个组名，这样这个应用的多个实例在接收到消息的时候，只会有一个成员真正收到消息并进行处理。如下图所示，我们为 Service-A 和 Service-B 分别启动了两个实例，并且根据服务名进行了分组，这样当消息进入主题之后，Group-A 和 Group-B 都会收到消息的副本，但是在两个组中都只会有一个实例对其进行消费。



默认情况下，当没有为应用指定消费组的时候，Spring Cloud Stream 会为其分配一个独立的匿名消费组。所以，如果同一主题下的所有应用都没有被指定消费组的时候，当有消息发布之后，所有的应用都会对其进行消费，因为它们各自都属于一个独立的组。大部分情况下，我们在创建 Spring Cloud Stream 应用的时候，建议最好为其指定一个消费组，以防止对消息的重复处理，除非该行为需要这样做（比如刷新所有实例的配置等）。

消息分区

通过引入消费组的概念，我们已经能够在多实例的情况下，保障每个消息只被组内的一个实例消费。通过上面对消费组参数设置后的实验，我们可以观察到，消费组无法控制消息具体被哪个实例消费。也就是说，对于同一条消息，它多次到达之后可能是由不同的实例进行消费的。但是对于一些业务场景，需要对一些具有相同特征的消息设置每次都被同一个消费实例处理，比如，一些用于监控服务，为了统计某段时间内消息生产者发送的报告内容，监控服务需要在自身聚合这些数据，那么消息生产者可以为消息增加一个固有的特征 ID 来进行分区，使得拥有这些 ID 的消息每次都能被发送到一个特定的实例上实现累计统计的效果，否则这些数据就会分散到各个不同的节点导致监控结果不一致的情况。而分区概念的引入就是为了解决这样的问题：当生产者将消息数据发送给多个消费者实例时，保证拥有共同特征的消息数据始终是由同一个消费者实例接收和处理。

Spring Cloud Stream 为分区提供了通用的抽象实现，用来在消息中间件的上层实现分区处理，所以它对于消息中间件自身是否实现了消息分区并不关心，这使得 Spring Cloud Stream 为不具备分区功能的消息中间件也增加了分区功能扩展。

使用详解

在介绍了 Spring Cloud Steam 的基础结构和核心概念之后，下面我们来详细地学习一下它所提供的一些核心注解的具体使用方法。

开启绑定功能

在 Spring Cloud Stream 中，我们需要通过@EnableBinding 注解来为应用启动消息驱动的功能，该注解我们在快速入门中已经有了基本的介绍，下面来详细看看它的定义：

```
@Target({ElementType.TYPE, ElementType.ANNOTATION_TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Configuration
@Import({
  ChannelBindingServiceConfiguration.class,
  BindingBeansRegistrar.class,
  BinderFactoryConfiguration.class,
  SpelExpressionConverterConfiguration.class})
@EnableIntegration
public @interface EnableBinding {

    Class<?>[] value() default {};

}
```

从该注解的定义中我们可以看到，它自身包含了@Configuration 注解，所以用它注解的类也会成为 Spring 的基本配置类。另外该注解还通过@Import 加载了 Spring Cloud Stream 运行需要的几个基础配置类。

- ChannelBindingServiceConfiguration：该配置会加载消息通道绑定必要的一些实例，比如，用于处理消息通道绑定的 ChannelBindingService 实例、消息类型转换器 MessageConverterConfigurer、消息通道工厂 BindableChannel-Factory 等重要实例，有兴趣的读者可以自行查看这些默认配置，以对其有更深入的理解。

- BindingBeansRegistrar：该类是 ImportBeanDefinitionRegistrar 接口的实现，主要是在 Spring 加载 Bean 的时候被调用，用来实现加载更多的 Bean。由于 BindingBeansRegistrar 被@EnableBinding 注解的@Import 所引用，所以在其他配置加载完后，它的实现会被回调来创建其他的 Bean，而这些 Bean 则从@EnableBinding 注解的 value 属性定义的类中获取。就如我们入门实例中定义的@EnableBinding(Sink.class)，它在加载用于消息驱动的基础 Bean 之后，会继续加载 Sink 中定义的具体消息通道绑定。
- BinderFactoryConfiguration：Binder 工厂的配置，主要用来加载与消息中间件相关的配置信息，比如，它会从应用工程的 META-INF/spring.binders 中加载针对具体消息中间件相关的配置文件等。
- SpelExpressionConverterConfiguration：SpEL 表达式转换器配置。

@EnableBinding 注解只有一个唯一的属性：value。上面已经介绍过，由于该注解@Import 了 BindingBeansRegistrar 实现，所以在加载了基础配置内容之后，它会回调来读取 value 中的类，以创建消息通道的绑定。另外，由于 value 是一个 Class 类型的数组，所以我们可以通过 value 属性一次性指定多个关于消息通道的配置。

绑定消息通道

在 Spring Cloud Steam 中，我们可以在接口中通过@Input 和@Output 注解来定义消息通道，而用于定义绑定消息通道的接口则可以被@EnableBinding 注解的 value 参数来指定，从而在应用启动的时候实现对定义消息通道的绑定。

在快速入门的示例中，我们演示了使用 Sink 接口绑定的消息通道。Sink 接口是 Spring Cloud Steam 提供的一个默认实现，除此之外还有 Source 和 Processor，可从它们的源码中学习它们的定义方式：

```
public interface Sink {

    String INPUT = "input";

    @Input(Sink.INPUT)
    SubscribableChannel input();

}

public interface Source {

    String OUTPUT = "output";

    @Output(Source.OUTPUT)
```

```
    MessageChannel output();

}

public interface Processor extends Source, Sink {

}
```

从上面的源码中，我们可以看到，Sink 和 Source 中分别通过@Input 和@Output 注解定义了输入通道和输出通道，而 Processor 通过继承 Source 和 Sink 的方式同时定义了一个输入通道和一个输出通道。

另外，@Input 和@Output 注解都还有一个 value 属性，该属性可以用来设置消息通道的名称，这里 Sink 和 Source 中指定的消息通道名称分别为 input 和 output。如果我们直接使用这两个注解而没有指定具体的 value 值，将默认使用方法名作为消息通道的名称。

最后，需要注意一点，当我们定义输出通道的时候，需要返回 MessageChannel 接口对象，该接口定义了向消息通道发送消息的方法；而定义输入通道时，需要返回 SubscribableChannel 接口对象，该接口继承自 MessageChannel 接口，它定义了维护消息通道订阅者的方法。

注入绑定接口

在完成了消息通道绑定的定义之后，Spring Cloud Stream 会为其创建具体的实例，而开发者只需要通过注入的方式来获取这些实例并直接使用即可。举个简单的例子，我们在快速入门示例中已经为 Sink 接口绑定的 input 消息通道实现了具体的消息消费者，下面可以通过注入的方式实现一个消息生成者，向 input 消息通道发送数据。

- 创建一个将 Input 消息通道作为输出通道的接口，具体如下：

```
public interface SinkSender {

    @Output(Sink.INPUT)
    MessageChannel output();

}
```

- 对快速入门中定义的 SinkReceiver 做一些修改：在@EnableBinding 注解中增加对 SinkSender 接口的指定，使 Spring Cloud Stream 能创建出对应的实例。

```
@EnableBinding(value = {Sink.class, SinkSender.class})
public class SinkReceiver {
```

```
    private static Logger logger = LoggerFactory.getLogger(SinkReceiver.class);

    @StreamListener(Sink.INPUT)
    public void receive(Object payload) {
        logger.info("Received: " + payload);
    }

}
```

- 创建一个单元测试类，通过@Autowired 注解注入 SinkSender 的实例，并在测试用例中调用它的发送消息方法。

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = HelloApplication.class)
@WebAppConfiguration
public class HelloApplicationTests {

    @Autowired
    private SinkSender sinkSender;

    @Test
    public void contextLoads() {
        sinkSender.output().send(MessageBuilder.withPayload("From
SinkSender").build());
    }

}
```

- 运行该单元测试用例，如果可以在控制台中找到如下输出内容，表明我们的试验已经成功了，消息被正确地发送到了 input 通道中，并被相对应的消息消费者输出。

```
...
INFO 10656 --- [          main] com.didispace.HelloApplication          : Received:
From SinkSender
...
```

注入消息通道

由于 Spring Cloud Stream 会根据绑定接口中的@Input 和@Output 注解来创建消息通道实例，所以我们也可以通过直接注入的方式来使用消息通道对象。比如，我们可以通过下面的示例，注入上面例子中 SinkSender 接口中定义的名为 input 的消息输入通道。

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = HelloApplication.class)
@WebAppConfiguration
public class HelloApplicationTests {
```

```
    @Autowired
    private MessageChannel input;

    @Test
    public void contextLoads() {
      input.send(MessageBuilder.withPayload("From MessageChannel").build());
    }

}
```

上面定义的内容，完成了与之前通过注入绑定接口 SinkSender 方式实现的测试用例相同的操作。因为在通过注入绑定接口实现时，sinkSender.output()方法实际获得的就是 SinkSender 接口中定义的 MessageChannel 实例，只是在这里我们直接通过注入的方式来实现了而已。这种用法虽然很直接，但是也容易犯错，很多时候我们在一个微服务应用中可能会创建多个不同名的 MessageChannel 实例，这样通过@Autowired 注入时，要注意参数命名需要与通道同名才能被正确注入，或者也可以使用@Qualifier 注解来特别指定具体实例的名称，该名称需要与定义 MessageChannel 的@Output 中的 value 参数一致，这样才能被正确注入。比如下面的例子，在一个接口中定义了两个输出通道，分别命名为 Output-1 和 Output-2，当要使用 Output-1 的时候，可以通过 @Qualifier("Output-1")来指定这个具体的实例来注入使用。

```
public interface MySource {

    @Output("Output-1")
    MessageChannel output1();

    @Output("Output-2")
    MessageChannel output2();

}

@Component
public class OutputSender {

    @Autowired @Qualifier("Output-1")
    private MessageChannel output;
    ...

}
```

消息生产与消费

由于 Spring Cloud Stream 是基于 Spring Integration 构建起来的，所以在使用 Spring Cloud Stream 构建消息驱动服务的时候，完全可以使用 Spring Integration 的原生注解来实现各种业务需求。同时，为了简化面向消息的编程模型，Spring Cloud Stream 还提供了 `@StreamListener` 注解对输入通道的处理做了进一步优化。下面我们分别从这两方面来学习一下对消息的处理。

Spring Integration 原生支持

通过之前的内容，我们已经能够通过注入绑定接口和消息通道的方式实现向名为 `input` 的消息通道发送信息。接下来，我们通过 Spring Integration 原生的`@ServiceActivator`和`@InboundChannelAdapter`注解来尝试实现相同的功能，具体实现如下：

```
@EnableBinding(value = {Sink.class})
public class SinkReceiver {

    private static Logger logger = LoggerFactory.getLogger(SinkReceiver.class);

    @ServiceActivator(inputChannel=Sink.INPUT)
    public void receive(Object payload) {
        logger.info("Received: " + payload);
    }

}

@EnableBinding(value = {SinkSender.SinkOutput.class})
public class SinkSender {

    private static Logger logger = LoggerFactory.getLogger(SinkSender.class);

    @Bean
    @InboundChannelAdapter(value = SinkOutput.OUTPUT, poller = @Poller(fixedDelay
= "2000"))
    public MessageSource<Date> timerMessageSource() {
        return () -> new GenericMessage<>(new Date());
    }

    public interface SinkOutput {

        String OUTPUT = "input";

        @Output(SinkOutput.OUTPUT)
```

```
        MessageChannel output();
    }

}
```

上面展示的两段代码分别属于两个不同的应用。

- SinkReceiver 类属于消息消费者实现，与之前实现的类似，只是做了一些修改：使用原生的@ServiceActivator 注解替换了@StreamListener，实现对 Sink.INPUT 通道的监听处理，而该通道绑定了名为 input 的主题。
- SinkSender 类属于消息生产者实现，它在内部定义了 SinkOutput 接口来将输出通道绑定到名为 input 的主题中。由于 SinkSender 和 SinkReceiver 共用一个主题，所以它们构成了一组生产者与消费者。另外，在 SinkSender 中还创建了用于生产消息的 timerMessageSource 方法，该方法会将当前时间作为消息返回。而@InboundChannelAdapter 注解定义了该方法是对 SinkOutput.OUTPUT 通道的输出绑定，同时使用 poller 参数将该方法设置为轮询执行，这里我们定义为 2000 毫秒，所以它会以 2 秒的频率向 SinkOutput.OUTPUT 通道输出当前时间。

执行上面定义的程序，可以得到类似下面的输出：

```
    INFO 248 --- [ask-scheduler-2] com.didispace.HelloApplication          : Received:
Sun Nov 13 11:22:39 CST 2016
    INFO 248 --- [ask-scheduler-3] com.didispace.HelloApplication          : Received:
Sun Nov 13 11:22:41 CST 2016
    INFO 248 --- [ask-scheduler-2] com.didispace.HelloApplication          : Received:
Sun Nov 13 11:22:43 CST 2016
    INFO 248 --- [ask-scheduler-1] com.didispace.HelloApplication          : Received:
Sun Nov 13 11:22:45 CST 2016
    INFO 248 --- [ask-scheduler-5] com.didispace.HelloApplication          : Received:
Sun Nov 13 11:22:47 CST 2016
```

另外，还可以通过@Transformer 注解对指定通道的消息进行转换。比如，我们可以在上面的 SinkSender 类中增加下面的内容：

```
    @Transformer(inputChannel = Sink.INPUT, outputChannel = Sink.INPUT)
    public Object transform(Date message) {
        return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(message);
    }
```

再次执行程序，可以得到下面的输出内容，消息内容被成功转换为 yyyy-MM-dd HH:mm:ss 格式了。

```
    INFO 22092 --- [ask-scheduler-2] com.didispace.HelloApplication          : Received:
2016-11-13 12:12:21
    INFO 22092 --- [ask-scheduler-3] com.didispace.HelloApplication          : Received:
2016-11-13 12:12:23
```

```
    INFO 22092 --- [ask-scheduler-2] com.didispace.HelloApplication          : Received:
2016-11-13 12:12:25
    INFO 22092 --- [ask-scheduler-1] com.didispace.HelloApplication          : Received:
2016-11-13 12:12:27
    INFO 22092 --- [ask-scheduler-5] com.didispace.HelloApplication          : Received:
2016-11-13 12:12:29
```

更多关于 Spring Integration 的使用细节请查阅官方文档。

@StreamListener 详解

通过入门示例，对于@StreamListener 注解，大家应该都已经有了一些基本的认识，通过该注解修饰的方法，Spring Cloud Steam 会将其注册为输入消息通道的监听器。当输入消息通道中有消息到达的时候，会立即触发该注解修饰方法的处理逻辑对消息进行消费。

消息转换

@SteamListener 和@ServiceActivator 注解都实现了对输入消息通道的监听，但是@SteamListener 相比@ServiceActivator 更为强大，因为它还内置了一系列的消息转换功能，这使得基于@SteamListener 注解实现的消息处理模型更为简单。

大部分情况下，我们通过消息来对接服务或系统时，消息生产者都会以结构化的字符串形式来发送，比如 JSON 或 XML。当消息到达的时候，输入通道的监听器需要对该字符串做一定的转化，将 JSON 或 XML 转换成具体的对象，然后再做后续的处理。

假设，我们需要传输一个 User 对象，该对象有 name 和 age 两个字段，这时，如果使用@ServiceActivator 注解的话，可以通过下面的代码实现：

```
@EnableBinding(value = {Sink.class})
public class SinkReceiver {

    private static Logger logger = LoggerFactory.getLogger(SinkReceiver.class);

    @ServiceActivator(inputChannel=Sink.INPUT)
    public void receive(User user) {
        logger.info("Received: " + user);
    }

    @Transformer(inputChannel = Sink.INPUT, outputChannel = Sink.INPUT)
    public User transform(String message) throws Exception {
        ObjectMapper objectMapper = new ObjectMapper();
        User user = objectMapper.readValue(message, User.class);
        return user;
    }

}
```

```
@EnableBinding(value = {SinkSender.SinkOutput.class})
public class SinkSender {

    private static Logger logger = LoggerFactory.getLogger(SinkSender.class);

    @Bean
    @InboundChannelAdapter(value = Sink.INPUT, poller = @Poller(fixedDelay = "2000"))
    public MessageSource<String> timerMessageSource() {
        return () -> new GenericMessage<>("{\"name\":\"didi\", \"age\":30}");
    }

    public interface SinkOutput {

        String OUTPUT = "input";

        @Output(SinkOutput.OUTPUT)
        MessageChannel output();

    }
}
```

由于@ServiceActivator 本身不具备对消息的转换能力，所以当代表 User 对象的 JSON 字符串到达后，它自身无法将其转换成 User 对象。所以，这里需要通过 @Transformer 注解帮助将字符串类型的消息转换成 User 对象，并将转换结果传递给 @ServiceActivator 的处理方法做后续的消费。

如果我们使用@SteamListener 注解的话，就可以把上面的实现简化为下面的代码：

```
@EnableBinding(value = {Sink.class})
public class SinkReceiver {

    private static Logger logger = LoggerFactory.getLogger(SinkReceiver.class);

    @StreamListener(Sink.INPUT)
    public void receive(User user) {
        logger.info("Received: " + user);
    }

}
```

从上面的实现中可以看到，我们去掉了@Transformer 注解的方法，同时用 @StreamListener 替代了@ServiceActivator 注解，而 SinkSender 类不需要做任何修改，只需在配置文件中增加 spring.cloud.stream.bindings.input.content-type=application/json 属性设置，这样我们可以得到与之前一样的结果。

@StreamListener 注解能够通过配置属性实现 JSON 字符串到对象的转换，这是因为在 Spring Cloud Steam 中实现了一套可扩展的消息转换机制。在消息消费逻辑执行之前，消息转换机制会根据消息头信息中声明的消息类型（即上面对 input 通道配置的 content-type 参数），找到对应的消息转换器并实现对消息的自动转换。

消息反馈

很多时候在处理完输入消息之后，需要反馈一个消息给对方，这时候可以通过 @SendTo 注解来指定返回内容的输出通道。

比如，假设我们有这样的两个应用，App1 和 App2。

- App1 中实现了对 input 输入通道的监听，并且在接收到消息之后，对消息做了一些简单加工，然后通过@SendTo 把处理方法返回的内容以消息的方式发送到 output 通道中，具体如下：

```
@EnableBinding(value = {Processor.class})
public class App1 {

    private static Logger logger = LoggerFactory.getLogger(App1.class);

    @StreamListener(Processor.INPUT)
    @SendTo(Processor.OUTPUT)
    public Object receiveFromInput(Object payload) {
        logger.info("Received: " + payload);
        return "From Input Channel Return - " + payload;
    }

}
```

- App2 是 App1 应用中 input 通道的生产者以及 output 通道的消费者。为了继续使用 Processor 绑定接口的定义，我们可以在配置文件中将该应用的 input 和 output 绑定反向地做一些配置。因为对于 App2 来说，它的 input 绑定通道实际上是对 output 主题的消费者，而 output 绑定通道实际上是对 input 主题的生产者，所以我们可以做如下具体配置。指定通道的具体主题来实现与 App1 应用的消息交互：

```
spring.cloud.stream.bindings.input.destination=output
spring.cloud.stream.bindings.output.destination=input
```

- App2 中程序的逻辑实现如下所示。通过 timerMessageSource 方法，向 output 绑定的输出通道中发送内容，也就是向名为 input 的主题中传递消息，这样 App1 中的 input 输入通道就可以收到这个消息。同时，这里还创建了对 input 输入消

息通道的绑定。通过上面的配置，它会监听来自 output 主题的消息，通过 receiveFromOutput 方法，会将消息内容输出。

```
@EnableBinding(value = {Processor.class})
public class App2 {

    private static Logger logger = LoggerFactory.getLogger(App2.class);

    @Bean
    @InboundChannelAdapter(value = Processor.OUTPUT, poller = @Poller(fixedDelay =
"2000"))
    public MessageSource<Date> timerMessageSource() {
        return () -> new GenericMessage<>(new Date());
    }

    @StreamListener(Processor.INPUT)
    public void receiveFromOutput(Object payload) {
        logger.info("Received: " + payload);
    }

}
```

将 App1 和 App2 同时运行起来，我们分别可以从它们的控制台中看到如下内容。

- App1 应用的控制台输出如下所示，它输出了来自 App2 应用中定义的向 input 主题中轮询发送的时间。

```
    INFO 28180 --- [mWYODqFPjxMmg-1] com.didispace.HelloApplication      : Received:
Wed Nov 16 08:19:16 CST 2016
    INFO 28180 --- [mWYODqFPjxMmg-1] com.didispace.HelloApplication      : Received:
Wed Nov 16 08:19:18 CST 2016
    INFO 28180 --- [mWYODqFPjxMmg-1] com.didispace.HelloApplication      : Received:
Wed Nov 16 08:19:20 CST 2016
```

- App2 应用的控制台输出如下所示，由于 App1 应用在打印了 input 主题中的消息之后做了一些简单的字符串拼接，然后将拼接后的字符串输出到 output 主题，而下面的输出内容正是 App2 应用从 output 主题中获取的消息内容。

```
    INFO 31916 --- [xmrMN07yhd8Ag-1] com.didispace.HelloApplication      : Received:
From Input Channel Return - Wed Nov 16 08:19:16 CST 2016
    INFO 31916 --- [xmrMN07yhd8Ag-1] com.didispace.HelloApplication      : Received:
From Input Channel Return - Wed Nov 16 08:19:18 CST 2016
    INFO 31916 --- [xmrMN07yhd8Ag-1] com.didispace.HelloApplication      : Received:
From Input Channel Return - Wed Nov 16 08:19:20 CST 2016
```

在 Spring Cloud Stream 中除了可以使用@SendTo 注解将方法返回结果输出到消息通道中，还可以使用原生注解@ServiceActivator 的 outputChannel 属性配置输出通道

把返回结果发送给消息中间件。根据上面的例子，我们可以将 App1 应用修改成下面这样，它们实现的效果是一样的。

```
@EnableScheduling
@EnableBinding(value = {Processor.class})
public class App1 {

    private static Logger logger = LoggerFactory.getLogger(App1.class);

    @ServiceActivator(inputChannel= Processor.INPUT, outputChannel =
Processor.OUTPUT)
    public Object receiveFromInput(Object payload) {
        logger.info("Received: " + payload);
        return "From Input Channel Return - " + payload;
    }

}
```

响应式编程

在 Spring Cloud Stream 中还支持使用基于 RxJava 的响应式编程来处理消息的输入和输出。与 RxJava 的整合使用同样很容易，下面我们详细看看如何使用 RxJava 实现上面消息反馈中试验的场景：App1 应用接收来自 App2 应用发送到 input 主题的消息，并返回一条消息到 output 主题供 App2 应用消息输出。

- 在 pom.xml 中引入 spring-cloud-stream-rxjava 依赖，具体如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-rxjava</artifactId>
    <version>1.0.2.RELEASE</version>
</dependency>
```

- 改造 App1 的实现，具体如下：

```
@EnableRxJavaProcessor
public class App1 {

    private static Logger logger = LoggerFactory.getLogger(App1.class);

    @Bean
    public RxJavaProcessor<String,String> processor() {
        return inputStream -> inputStream.map(data -> {
            logger.info("Received: " + data);
            return data;
```

```
        }).map(data -> String.valueOf("From Input Channel Return - " + data));
    }

}
```

通过上面的改造，我们再次运行 App1 和 App2 就可以得到与上一节消息反馈中的试验相同的结果。下面对前面的实现内容做一些说明。

- 在 App1 的类名上，我们使用 `@EnableRxJavaProcessor` 替代了原来的 `@EnableBinding(value = {Processor.class})`，该注解用来标识当前类中应该提供一个 `RxJavaProcessor` 实现的 Bean。另外从其源码中还可以看到它自身就实现了对 `Processor` 定义通道的绑定：`@EnableBinding({Processor.class})`。

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@EnableBinding({Processor.class})
@Import({RxJavaProcessorConfiguration.class})
public @interface EnableRxJavaProcessor {
}
```

- 在 App1 中还定义了一个 `RxJavaProcessor` 的实现 Bean。在 `RxJavaProcessor` 接口中定义了一个用来处理输入通道和返回内容给输出通道的 `process` 方法，由于输入输出都采用了 `Observable`，所以该方法只会在应用启动的时候调用一次，用来设置数据流。当有消息到达输入通道时，会采用 RxJava 实现的观察者模式来消费和输出内容。

```
public interface RxJavaProcessor<I, O> {

    Observable<O> process(Observable<I> input);

}
```

- 除了实现上面的场景之外，通过利用 RxJava 的支持，我们还能轻易地实现消息的缓存聚合。比如，我们希望 App1 在接收到 5 条消息之后才将处理结果返回给输出通道，那么只需通过下面的改进即可轻松实现这样的场景：

```
@EnableRxJavaProcessor
public class App1 {

    private static Logger logger = LoggerFactory.getLogger(App1.class);

    @Bean
    public RxJavaProcessor<String,String> processor() {
```

```
            return inputStream -> inputStream.map(data -> {
                logger.info("Received: " + data);
                return data;
            }).buffer(5).map(data -> String.valueOf("From Input Channel Return - " +
data));
        }

    }
```

再次执行 App1 和 App2 应用，此时我们在 App1 应用中会得到一样的输出，但是在 App2 应用中会得到下面的输出：

```
    INFO 8796 --- [-maOhv-th7Raw-1] com.didispace.HelloApplication          : Received:
From Input Channel Return - [1479390916408, 1479390918409, 1479390920412, 1479390922414,
1479390924415]
    INFO 8796 --- [-maOhv-th7Raw-1] com.didispace.HelloApplication          : Received:
From Input Channel Return - [1479390926416, 1479390928418, 1479390930419, 1479390932421,
1479390934423]
```

消费组与消息分区

在“核心概念”一节中，我们对消费组和消息分区已经进行了基本的介绍，在这里来详细介绍一下这两个概念的使用方法。

消费组

通常每个服务都不会以单节点的方式运行在生产环境中，当同一个服务启动多个实例的时候，这些实例会绑定到同一个消息通道的目标主题上。默认情况下，当生产者发出一条消息到绑定通道上，这条消息会产生多个副本被每个消费者实例接收和处理。但是在有些业务场景之下，我们希望生产者产生的消息只被其中一个实例消费，这个时候就需要为这些消费者设置消费组来实现这样的功能。实现的方式非常简单，只需在服务消费者端设置 `spring.cloud.stream.bindings.input.group` 属性即可，比如可以像下面这样实现。

- 可以先实现一个消费者应用 `SinkReceiver`，实现 `greetings` 主题上的输入通道绑定，它的实现如下：

```
@EnableBinding(value = {Sink.class})
public class SinkReceiver {

    private static Logger logger = LoggerFactory.getLogger(SinkReceiver.class);

    @StreamListener(Sink.INPUT)
    public void receive(User user) {
```

```
        logger.info("Received: " + user);
    }
}
```

- 为了将 SinkReceiver 的输入通道目标设置为 greetings 主题，以及将该服务的实例设置为同一个消费组，可做如下设置：

```
spring.cloud.stream.bindings.input.group=Service-A
spring.cloud.stream.bindings.input.destination=greetings
```

通过 spring.cloud.stream.bindings.input.group 属性指定了该应用实例都属于 Service-A 消费组，而 spring.cloud.stream.bindings.input.destination 属性则指定了输入通道对应的主题名。

- 完成了消息消费者应用之后，我们再来实现一个消息生产者应用 SinkSender，具体如下：

```
@EnableBinding(value = {Source.class})
public class SinkSender {

    private static Logger logger = LoggerFactory.getLogger(SinkSender.class);

    @Bean
    @InboundChannelAdapter(value = Source.OUTPUT, poller = @Poller(fixedDelay =
"2000"))
    public MessageSource<String> timerMessageSource() {
        return () -> new GenericMessage<>("{\"name\":\"didi\", \"age\":30}");
    }

}
```

- 为消息生产者 SinkSender 做一些设置，让它的输出通道绑定目标也指向 greetings 主题，具体如下：

```
spring.cloud.stream.bindings.output.destination=greetings
```

到这里，对于消费分组的示例就完成了。分别运行上面实现的生产者与消费者，其中消费者我们启动多个实例。通过控制台，可以发现，每个生产者发出的消息会被启动的消费者以轮询的方式进行接收和输出。

消息分区

通过消费组的设置，虽然我们已经能够在多实例环境下，保证同一消息只被一个消费者实例进行接收和处理，但是，对于一些特殊场景，除了要保证单一实例消费之外，还希望那些具备相同特征的消息都能够被同一个实例进行消费。这时候我们就需要对消息进行分区处理。

在 Spring Cloud Stream 中实现消息分区非常简单，我们对消费组示例做一些配置修改就能实现，具体如下所示。

- 在消费者应用 SinkReceiver 中，对配置文件做一些修改，具体如下：

```
spring.cloud.stream.bindings.input.group=Service-A

spring.cloud.stream.bindings.input.destination=greetings
spring.cloud.stream.bindings.input.consumer.partitioned=true
spring.cloud.stream.instanceCount=2
spring.cloud.stream.instanceIndex=0
```

从上面的配置中，我们可以看到增加了下面这三个参数。

1. spring.cloud.stream.bindings.input.consumer.partitioned：通过该参数开启消费者分区功能。
2. spring.cloud.stream.instanceCount：该参数指定了当前消费者的总实例数量。
3. spring.cloud.stream.instanceIndex：该参数设置当前实例的索引号，从 0 开始，最大值为 spring.cloud.stream.instanceCount 参数-1。试验的时候需要启动多个实例，可以通过运行参数来为不同实例设置不同的索引值。

- 在生产者应用 SinkSender 中，对配置文件也做一些修改，具体如下所示。

```
spring.cloud.stream.bindings.output.destination=greetings
spring.cloud.stream.bindings.output.producer.partitionKeyExpression=payload
spring.cloud.stream.bindings.output.producer.partitionCount=2
```

从上面的配置中，我们可以看到增加了下面这两个参数。

1. spring.cloud.stream.bindings.output.producer.partitionKey-Expression：通过该参数指定了分区键的表达式规则，我们可以根据实际的输出消息规则配置 SpEL 来生成合适的分区键。
2. spring.cloud.stream.bindings.output.producer.partitionCount：该参数指定了消息分区的数量。

到这里消息分区配置就完成了，我们可以再次启动这两个应用，同时启动多个消费者。但需要注意的是，要为消费者指定不同的实例索引号，这样当同一个消息被发送给消费组时，可以发现只有一个消费实例在接收和处理这些相同的消息。

消息类型

Spring Cloud Stream 为了让开发者能够在消息中声明它的内容类型，在输出消息中定

义了一个默认的头信息：contentType。对于那些不直接支持头信息的消息中间件，Spring Cloud Stream 提供了自己的实现机制，它会在消息发出前自动将消息包装进它自定义的消息封装格式中，并加入头信息。而对于那些自身就支持头信息的消息中间件，Spring Cloud Stream 构建的服务可以接收并处理来自非 Spring Cloud Stream 构建但包含符合规范头信息的应用程序发出的消息。

Spring Cloud Stream 允许使用 spring.cloud.stream.bindngs.<channelName>.content-type 属性以声明式的配置方式为绑定的输入和输出通道设置消息内容的类型。此外，原生的消息类型转换器依然可以轻松地用于我们的应用程序。目前，Spring Cloud Stream 中自带支持了以下几种常用的消息类型转换。

- JSON 与 POJO 的互相转换。
- JSON 与 org.springframework.tuple.Tuple 的互相转换。
- Object 与 byte[]的互相转换。为了实现远程传输序列化的原始字节，应用程序需要发送 byte 类型的数据，或是通过实现 Java 的序列化接口来转换为字节（Object 对象必须可序列化）。
- String 与 byte[]的互相转换。
- Object 向纯文本的转换：Object 需要实现 toString()方法。

上面所指的 JSON 类型可以表现为一个 byte 类型的数组，也可以是一个包含有效 JSON 内容的字符串。另外，Object 对象可以由 JSON、byte 数组或者字符串转换而来，但是在转换为 JSON 的时候总是以字符串的形式返回。

MIME 类型

在 Spring Cloud Stream 中定义的 content-type 属性采用了 Media Type，即 Internet Media Type（互联网媒体类型），也被称为 MIME 类型，常见的有 application/json、text/plain;charset=UTF-8，相信接触过 HTTP 的工程师们对这些类型都不会感到陌生。MIME 类型对于标示如何转换为 String 或 byte[]非常有用。并且，我们还可以使用 MIME 类型格式来表示 Java 类型，只需要使用带有类型参数的一般类型：application/x-java-object。比如，我们可以使用 application/x-java-object;type=java.util.Map 来表示传输的是一个 java.util.Map 对象，或是使用 application/x-java-object;type=com.didispace.User 来表示传输的是一个 com.didispace.User 对象；除此之外，更重要的是，它还提供了自定义的 MIME 类型，比如通过 application/x-spring-tuple 来指定 Spring 的 Tuple 类型。

在 Spring Cloud Stream 中默认提供了一些可以开箱即用的类型转换器，具体如下表所示。

源内容类型	目标内容类型	content-type 头	content-type	注释
POJO	JSON String	ignored	application/json	
Tuple	JSON String	ignored	application/json	JSON is tailored for Tuple
POJO	String (toString())	ignored	text/plain, java.lang.String	
POJO	byte[] (java.io serialized)	ignored	application/x-java-serialized-object	
JSON byte[]或 String	POJO	application/json (or none)	application/x-java-object	
byte[]或 String	Serializable	application/x-java-serialized-object	application/x-java-object	
JSON byte[]或 String	Tuple	application/json (or none)	application/x-spring-tuple	
byte[]	String	any	text/plain, java.lang.String	will apply any Charset specified in the content-type header
String	byte[]	any	application/octet-stream	will apply any Charset specified in the content-type header

消息类型的转换行为只会在需要进行转换时才被执行，比如，当服务模块产生了一个头信息为 application/json 的 XML 字符串消息，Spring Cloud Stream 是不会将该 XML 字符串转换为 JSON 的，这是因为该模块的输出内容已经是一个字符串类型了，所以它并不会将其做进一步的转换。

另外需要注意的是，Spring Cloud Stream 虽然同时支持输入通道和输出通道的消息类型转换，但还是推荐开发者尽量在输出通道中做消息转换。因为对于输入通道的消费者来说，当目标是一个 POJO 的时候，使用@StreamListener 注解是能够支持自动对其进行转换的。

Spring Cloud Stream 除了提供上面这些开箱即用的转换器之外，还支持开发者自定义的消息转换器。这使得我们可以使用任意格式（包括二进制）的数据进行发送和接收，并且将这些数据与特定的 contentType 相关联。在应用启用的时候，Spring Cloud Stream 会将所有 org.springframework.messaging.converter.MessageConverter 接口实现的自定义转换器以及默认实现的那些转换器都加载到消息转换工厂中，以提供给消息处理时使用。

绑定器详解

在“核心概念”一节中，我们已经简单介绍过 Binder 绑定器的基本概念和作用：它是定义在应用程序与消息中间件之间的抽象层，用来屏蔽消息中间件对应用的复杂性，并提供简单而统一的操作接口给应用程序使用。在本节中，我们将详细介绍绑定器背后的细节和行为。

绑定器 SPI

绑定器 SPI 涵盖了一套可插拔的用于连接外部中间件的实现机制，其中包含了许多接口、开箱即用的实现类以及发现策略等内容。其中，最为关键的就是 Binder 接口，它是用来将输入和输出连接到外部中间件的抽象：

```
    public interface Binder<T, C extends ConsumerProperties, P extends
ProducerProperties> {

        Binding<T> bindConsumer(String name, String group, T inboundBindTarget, C
consumerProperties);

        Binding<T> bindProducer(String name, T outboundBindTarget, P
producerProperties);

    }
```

当应用程序对输入和输出通道进行绑定的时候，实际上就是通过该接口的实现来完成的。

- 向消息通道发送数据的生产者调用 bindProducer 方法来绑定输出通道时，第一个参数代表了发往消息中间件的目标名称，第二个参数代表了发送消息的本地通道实例，第三个参数是用来创建通道时使用的属性配置（比如分区键的表达式等）。
- 从消息通道接收数据的消费者调用 bindConsumer 方法来绑定输入通道时，第一个参数代表了接收消息中间件的目标名称，第二个参数代表了消费组的名称（如果多个消费者实例使用相同的组名，则消息将对这些消费者实例实现负载均衡，每个生产者发出的消息只会被组内一个消费者实例接收和处理），第三个参数代表了接收消息的本地通道实例，第四个参数是用来创建通道时使用的属性配置。

另外，从 Binder 的定义中，我们还可以知道 Binder 是一个参数化并且可扩展的接口。

- 对于输入与输出的绑定类型，在 1.0 版本中仅支持 MessageChannel，但是在接口中通过泛型定义，所以在未来可以对其进行扩展。
- 对于属性配置也提供了可扩展的定义，我们可以为特定的 Binder 以类型安全的方式来补充一些特有的属性。

一个典型的 Binder 绑定器实现一般包含以下内容。

- 一个实现 Binder 接口的类。
- 一个 Spring 配置加载类，用来创建连接消息中间件的基础结构使用的实例。
- 一个或多个能够在 classpath 下的 META-INF/spring.binders 路径找到的绑定器定义文件。比如我们可以在 spring-cloud-starter-stream-rabbit 中找到该文件，该文件中存储了当前绑定器要使用的自动化配置类的路径：

```
rabbit:\
org.springframework.cloud.stream.binder.rabbit.config.RabbitServiceAutoConfigur
ation
```

自动化配置

Spring Cloud Stream 通过绑定器 SPI 的实现将应用程序逻辑上的输入输出通道连接到物理上的消息中间件。消息中间件之间通常都会有或多或少的差异性，所以为了适配不同的消息中间件，需要为它们实现各自独有的绑定器。目前，Spring Cloud Stream 中默认实现了对 RabbitMQ、Kafka 的绑定器，在上面的示例中我们引入的 spring-cloud- starter-stream-rabbit 依赖中就包含了 RabbitMQ 的绑定器 spring-cloud-stream-binder-rabbit。

默认情况下，Spring Cloud Stream 也遵循 Spring Boot 自动化配置的特性。如果在 classpath 中能够找到单个绑定器的实现，那么 Spring Cloud Stream 会自动加载它。而我们在 classpath 中引入绑定器的方法也非常简单，只需要在 pom.xml 中增加对应消息中间件的绑定器依赖即可，比如：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
```

如果使用 Kafka，则引入：

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-stream-binder-kafka</artifactId>
</dependency>
```

多绑定器配置

当应用程序的 classpath 下存在多个绑定器时，Spring Cloud Stream 在为消息通道做绑定操作时，无法判断应该使用哪个具体的绑定器，所以我们需要为每个输入或输出通道指定具体的绑定器。

我们在一个应用程序中使用多个绑定器时，往往其中一个绑定器会是主要使用的，而第二个可能是为了适应一些特殊要求（比如性能等原因）。我们可以先通过设置默认绑定器来为大部分的通道设置绑定器。比如，使用 RabbitMQ 设置默认绑定器：

```
spring.cloud.stream.defaultBinder=rabbit
```

在设置了默认绑定器之后，再为其他一些少数的消息通道单独设置绑定器，比如：

```
spring.cloud.stream.bindings.input.binder=kafka
```

需要注意的是，上面我们设置参数时用来指定具体绑定器的值并不是消息中间件的名称，而是在每个绑定器实现的 META-INF/spring.binders 文件中定义的标识（一个绑定器实现的标识可以定义多个，以逗号分隔），所以上面配置的 rabbit 和 kafka 分别来自于各自的配置定义，它们的具体内容如下所示：

```
rabbit:\
org.springframework.cloud.stream.binder.rabbit.config.RabbitServiceAutoConfigur
ation

kafka:\
org.springframework.cloud.stream.binder.kafka.config.KafkaBinderConfiguration
```

另外，当需要在一个应用程序中使用同一类型不同环境的绑定器时，我们也可以通过配置轻松实现通道绑定。比如，当需要连接两个不同的 RabbitMQ 实例的时候，可以参照如下配置：

```
spring.cloud.stream.bindings.input.binder=rabbit1
spring.cloud.stream.bindings.output.binder=rabbit2

spring.cloud.stream.binders.rabbit1.type=rabbit
spring.cloud.stream.binders.rabbit1.environment.spring.rabbitmq.host=192.168.0.101
spring.cloud.stream.binders.rabbit1.environment.spring.rabbitmq.port=5672
spring.cloud.stream.binders.rabbit1.environment.spring.rabbitmq.username=springcloud
spring.cloud.stream.binders.rabbit1.environment.spring.rabbitmq.password=123456

spring.cloud.stream.binders.rabbit2.type=rabbit
spring.cloud.stream.binders.rabbit2.environment.spring.rabbitmq.host=192.168.0.102
spring.cloud.stream.binders.rabbit2.environment.spring.rabbitmq.port=5672
spring.cloud.stream.binders.rabbit2.environment.spring.rabbitmq.username=springcloud
spring.cloud.stream.binders.rabbit2.environment.spring.rabbitmq.password=123456
```

从上面的配置中，我们可以看到对于输入输出通道指定的绑定器采用了显式别名的配置方式，其中 input 通道的绑定指定了名为 rabbit1 的配置，而 output 通道的绑定指定了名为 rabbit2 的配置。当采用显式配置方式时会自动禁用默认的绑定器配置，所以当定义了显式配置别名后，对于这些绑定器的配置需要通过 spring.cloud.stream.

binders.<configurationName>属性来进行设置。对于绑定器的配置主要有下面 4 个参数。

- spring.cloud.stream.binders.<configurationName>.type 指定了绑定器的类型，可以是 rabbit、kafak 或者其他自定义绑定器的标识名，绑定器标识名的定义位于绑定器的 META-INF/spring.binders 文件中。
- spring.cloud.stream.binders.<configurationName>.environment 参数可以直接用来设置各绑定器的属性，默认为空。
- spring.cloud.stream.binders.<configurationName>.inheritEnvironment 参数用来配置当前绑定器是否继承应用程序自身的环境配置，默认为 true。
- spring.cloud.stream.binders.<configurationName>.defaultCandidate 参数用来设置当前绑定器配置是否被视为默认绑定器的候选项，默认为 true。当需要让当前配置不影响默认配置时，可以将该属性设置为 false。

RabbitMQ 与 Kafka 绑定器

在之前的章节中我们多次提到，Spring Cloud Stream 自身就提供了对 RabbitMQ 和 Kafka 的绑定器实现。由于 RabbitMQ 和 Kafka 自身的实现结构有所不同，理解绑定器实现与消息中间件自有概念之间的对应关系，对于正确使用绑定器和消息中间件会有非常大的帮助。下面就来分别说说 RabbitMQ 与 Kafka 的绑定器是如何使用消息中间件中不同概念来实现消息的生产与消费的。

- RabbitMQ 绑定器：在 RabbitMQ 中，通过 Exchange 交换器来实现 Spring Cloud Stream 的主题概念，所以消息通道的输入输出目标映射了一个具体的 Exchange 交换器。而对于每个消费组，则会为对应的 Exchange 交换器绑定一个 Queue 队列进行消息收发。
- Kafka 绑定器：由于 Kafka 自身就有 Topic 概念，所以 Spring Cloud Stream 的主题直接采用了 Kafka 的 Topic 主题概念，每个消费组的通道目标都会直接连接 Kafka 的主题进行消息收发。

配置详解

在 Spring Cloud Stream 中对绑定通道和绑定器提供了通用的属性配置项，一些绑定器还允许使用附加属性来对消息中间件的一些独有特性进行配置。这些属性的配置可以通过 Spring Boot 支持的任何配置方式来进行，包括使用环境变量、YAML 或者 properties 配置文件等。

基础配置

下表是Spring Cloud Stream应用级别的通用基础属性，这些属性都以spring.cloud.stream.为前缀。

参数名	说明	默认值
instanceCount	应用程序部署的实例数量。当使用 Kafka 的时候需要设置分区	1
instanceIndex	应用程序实例的索引，该值从 0 开始，最大值设置为-1。当使用分区和 Kafka 的时候使用	
dynamicDestinations	动态绑定的目标列表，该列表默认为空，当设置了具体列表之后，只有列表中的目标才能被发现	空
defaultBinder	默认绑定器配置，在应用程序中有多个绑定器时使用	空
overrideCloudConnectors	该属性只适用于激活 cloud 配置并且提供了 Spring Cloud Connectors 的应用。当使用默认属性 false 时，绑定器会自动检测合适的服务来绑定（比如，在 Cloud Foundry 中绑定的 RabbitMQ 服务）。当设置为 true 时，绑定器将忽略绑定的服务，而是依赖应用程序中的设置属性来进行绑定和连接	false

绑定通道配置

对于绑定通道的属性配置，在之前的示例中已经有过一些介绍，这些配置可以在属性文件中通过 spring.cloud.stream.bindings.<channelName>.<property>=<value>格式的参数来进行设置。其中<channelName>代表在绑定接口中定义的通道名称，比如，Sink 中的 input、Source 中的 output。

由于绑定通道分为输入通道和输出通道，所以在绑定通道的配置中包含了三类面向不同通道类型的配置：通用配置、消费者配置、生产者配置。

在下面介绍各具体配置属性时将省略 spring.cloud.stream.bindings.<channelName>.前缀，但在实际使用的时候记得使用完整的参数名称进行配置。

通用配置

对于绑定通道的通用配置，它们既适用于输入通道，也适用于输出通道，它们通过spring.cloud.stream.bindings.<channelName>.前缀来进行设置，具体可配置的属性如下表所示。

参数名	说明	默认值
destination	该参数用来配置消息通道绑定在消息中间件中的目标名称，比如 RabbitMQ 的 Exchange 或 Kafka 的 Topic。如果配置的绑定通道是一个消费者（输入通道），那么它可以绑定多个目标，这些目标名称通过逗号来分隔。如果没有设置该属性，将使用通道名	
group	该参数用来设置绑定通道的消费组，该参数主要作用于输入通道，以保证同一消息组中的消息只会有一个消费示例接收和处理	null
contentType	该参数用来设置绑定通道的消息类型	null
binder	当存在多个绑定器时使用该参数来指定当前通道使用哪个具体的绑定器	null

消费者配置

下面这些配置仅对输入通道的绑定有效，它们以 spring.cloud.stream.bindings.<channelName>.consumer.格式作为前缀。

参数名	说明	默认值
concurrency	输入通道消费者的并发数	1
partitioned	来自消息生产者的数据是否采用了分区	false
headerMode	当设置为 raw 的时候将禁用对消息头的解析。该属性只有在使用不支持消息头功能的中间件时有效，因为 Spring Cloud Stream 默认会解析嵌入的头部信息	embeddedHeaders
maxAttempts	对输入通道消息处理的最大重试次数	3
backOffInitialInterval	重试消息处理的初始间隔时间	1000
backOffMaxInterval	重试消息处理的最大间隔时间	10000
backOffMultiplier	重试消息处理时间间隔的递增乘数	2.0

生产者配置

下面这些配置仅对输出通道的绑定有效，它们以 spring.cloud.stream.bindings.<channelName>.producer.格式作为前缀。

参数名	说明	默认值
partitionKeyExpression	该参数用来配置输出通道数据分区键的 SpEL 表达式。当设置该属性之后，将对当前绑定通道的输出数据进行分区处理。同时，partitionCount 参数必须大于 1 才能生效。该参数与 partitionKeyExtractorClass 参数互斥，不能同时设置	null

续表

参数名	说明	默认值
partitionKeyExtractorClass	该参数用来配置分区键提取策略接口 PartitionKeyExtractorStrategy 的实现。当设置该属性之后，将对当前绑定通道的输出数据进行分区处理。同时，partitionCount 参数必须大于 1 才能生效。该参数与 partitionKeyExpression 参数互斥，不能同时设置	null
partitionSelectorClass	该参数用来指定分区选择器接口 PartitionSelectorStrategy 的实现。它与 partitionSelectorExpression 参数互斥，不能同时设置。如果两者都不设置，那么分区选择计算规则为 hashCode(key)%partitionCount，这里的 key 根据 partitionKeyExpression 或 partition-KeyExtractorClass 的配置计算得到	null
partitionSelectorExpression	该参数用来配置自定义分区选择器的 SpEL 表达式。它与 partitionSelectorClass 参数互斥，不能同时设置。如果两者都不设置，那么分区选择计算规则为 hashCode(key)%partitionCount，这里的 key 根据 partitionKeyExpression 或 partition-KeyExtractorClass 的配置计算得到	null
partitionCount	当分区功能开启时，使用该参数来配置消息数据的分区数。如果消息生产者已经配置了分区键的生成策略，那么它的值必须大于 1	1
headerMode	当设置为 raw 的时候将在输出时禁用嵌入式头部信息。该属性只有在使用不支持消息头功能的中间件时有效，因为 Spring Cloud Stream 会默认地为其嵌入头部信息	embeddedHeaders

绑定器配置

由于 Spring Cloud Stream 目前只实现了对 RabbitMQ 和 Kafka 的绑定器，所以对于绑定器的属性配置主要是针对这两个中间件。同时因为这两个中间件自身结构的不同，所以会

有不同的附加属性来配置各自的一些特殊功能和基础设施。

RabbitMQ 配置

RabbitMQ 绑定器的配置同绑定通道的配置一样，分为三种不同类型：通用配置、消费者配置以及生产者配置。

通用配置

由于 RabbitMQ 绑定器默认使用了 Spring Boot 的 ConnectionFactory，所以 RabbitMQ 绑定器支持在 Spring Boot 中的配置选项，它们以 spring.rabbitmq.为前缀。在之前的示例中，我们使用这些配置来指定具体的 RabbitMQ 地址、端口、用户信息等，更多配置可见 Spring Boot 文档中对 RabbitMQ 支持的章节内容，或是通过 spring-boot-starter-amqp 模块中 RabbitProperties 类的源码来查看。

在 Spring Cloud Stream 对 RabbitMQ 实现的绑定器中主要有下面几个属性，它们都以 spring.cloud.stream.rabbit.binder.为前缀。这些属性可以在 org.springframework.cloud.stream.binder.rabbit.config.RabbitBinderConfigurationProperties 中找到它们。

参数名	说明	默认值
adminAddresses	该参数用来配置 RabbitMQ 管理插件的 URL，当需要配置多个时用逗号分隔。该参数只有在 nodes 参数包含多个时使用，并且这里配置的内容必须在 spring.rabbitmq.addresses 中存在	
nodes	该参数用来配置 RabbitMQ 的节点名称，当需要配置多个时用逗号分隔。在配置多个的情况下，可以用来定位队列所在的服务器地址。这里配置的内容必须在 spring.rabbitmq.addresses 中存在	
compressionLevel	绑定通道的压缩级别，它的具体可选值及含义可见 java.util.zip.Deflater 中的定义	1

消费者配置

下面这些配置仅对 RabbitMQ 输入通道的绑定有效，它们以 spring.cloud.stream.rabbit.bindings.<channelName>.consumer.格式作为前缀。

参数名	说明	默认值
acknowledgeMode	用来设置消息的确认模式，可选配置包含：NONE、MANUAL、AUTO	AUTO

续表

参数名	说明	默认值
autoBindDlq	用来设置是否自动声明 DLQ（Dead-Letter-Queue），并绑定到 DLX（Dead-Letter-Exchange）上	false
durableSubscription	用来设置订阅是否被持久化，该参数仅在 group 被设置的时候有效	true
maxConcurrency	用来设置消费者的最大并发数	1
prefetch	用来设置预取数量，它表示在一次会话中从消息中间件中获取的消息数量，该值越大消息处理越快，但是会导致非顺序处理的风险	1
prefix	用来设置统一的目标和队列名称前缀	
recoveryInterval	用来设置恢复连接的尝试时间间隔，以毫秒为单位	5000
requeueRejected	用来设置消息传递失败时重传	true
requestHeaderPatterns	用来设置需要被传递的请求头信息	[STANDARD_REQUEST_HEADERS,'*']
replyHeaderPatterns	用来设置需要被传递的响应头信息	[STANDARD_REPLY_HEADERS,'*']
republishToDlq	默认情况下，消息在重试也失败之后会被拒绝。如果 DLQ 被配置的时候，RabbitMQ 会将失败的消息路由到 DLQ 中。如果该参数设置为 true，总线会将失败的消息附加一些头信息（包括异常消息、引起失败的跟踪堆栈）之后重新发布到 DLQ 中	

续表

参数名	说明	默认值
transacted	用来设置是否启用 channel-transacted，即是否在消息中使用事务	false
txSize	用来设置 transaction-size 的数量，当 acknowledge-Mode 被设置为 AUTO 时，容器会在处理 txSize 数目消息之后才开始应答	1

生产者配置

下面这些配置仅对 RabbitMQ 输出通道的绑定有效，它们以 spring.cloud.stream.rabbit.bindings.<channelName>.producer.格式作为前缀。

参数名	说明	默认值
autoBindDlq	用来设置是否自动声明 DLQ（Dead-Letter-Queue），并绑定到 DLX（Dead-Letter-Exchange）上	false
batchingEnabled	是否启用消息批处理	false
batchSize	当批处理开启时，用来设置缓存的批处理消息数量	100
batchBufferLimit	批处理缓存限制	10000
batchTimeout	批处理超时时间	5000
compress	消息发送时是否启用压缩	false
deliveryMode	消息发送模式	PERSISTENT
prefix	用来设置统一的目标前缀	
requestHeaderPatterns	用来设置需要被传递的请求头信息	[STANDARD_REQUEST_HEADERS,'*']
replyHeaderPatterns	用来设置需要被传递的响应头信息	[STANDARD_REPLY_HEADERS,'*']

Kafka 配置

Kafka 绑定器的配置在类别上与 RabbitMQ 一样，分为三种不同类型：通用配置、消费

者配置以及生产者配置。但是由于 RabbitMQ 与 Kafka 自身有一些差异，所以它们的配置也不一样。

通用配置

Spring Cloud Stream 实现的 Kafka 绑定器包含下面这些通用配置，它们都以 spring.cloud.stream.kafka.binder.为前缀。

参数名	说明	默认值
brokers	Kafka 绑定器连接的消息中间件列表。需要配置多个时用逗号分隔，每个地址可以是单独的 host，也可以是 host:port 的形式	localhost
defaultBrokerPort	用来设置默认的消息中间件端口号。当 brokers 中的配置地址没有包含端口信息时，将使用该参数配置的默认端口进行连接	9092
zkNodes	Kafka 绑定器使用的 ZooKeeper 节点列表。需要配置多个时用逗号分隔，每个地址可以是单独的 host，也可以是 host:port 的形式	localhost
defaultZkPort	用来设置默认的 ZooKeeper 端口号。当 zkNodes 中的配置地址没有包含端口信息时，将使用该参数配置的默认端口进行连接	2181
headers	用来设置会被传输的自定义头信息	
offsetUpdateTimeWindow	用来设置 offset 的更新频率，以毫秒为单位，如果设置为 0 则忽略	10000
offsetUpdateCount	用来设置 offset 以次数表示的更新频率，如果为 0 则忽略，该参数与 offsetUpdateTimeWindow 互斥	0
requiredAcks	用来设置确认消息的数量	1
minPartitionCount	该参数仅在设置了 autoCreateTopics 和 autoAddPartitions 时生效，用来设置该绑定器所使用主题的全局分区最小数量。如果当生产者的 partitionCount 参数或 instanceCount * concurrency 设置大于该参数配置时，该参数值将被覆盖	1
replicationFactor	当 autoCreateTopics 参数为 true 时候，用来配置自动创建主题的副本数量	1
autoCreateTopics	该参数默认为 true，绑定器会自动地创建新主题。如果设置为 false，那么绑定器将使用已经配置的主题，但是在这种情况下，如果需要使用的主题不存在，绑定器会启动失败	true

续表

参数名	说明	默认值
autoAddPartitions	该参数默认为 false，绑定器会根据已经配置的主题分区来实现，如果目标主题的分区数小于预期值，那么绑定器会启动失败。如果该参数设置为 true，绑定器将在需要的时候自动创建新的分区	false
socketBufferSize	该参数用来设置 Kafka 的 Socket 缓存大小	2097152

消费者配置

下面这些配置仅对 Kafka 输入通道的绑定有效，它们以 spring.cloud.stream.kafka.bindings.<channelName>.consumer.格式作为前缀。

参数名	说明	默认值
autoCommitOffset	用来设置是否在处理消息时自动提交 offset。如果设置为 false，在消息头中会加入 ACK 头信息以实现延迟确认	true
autoCommitOnError	该参数只有在 autoCommitOffset 设置为 true 时才有效。当设置为 false 的时候，引起错误的消息不会自动提交 offset，仅提交成功消息的 offset。如果设置为 true，不论消息是否成功，都会自动提交。当不设置该值时，它实际上具有与 enableDlq 相同的配置值	
recoveryInterval	尝试恢复连接的时间间隔，以毫秒为单位	5000
resetOffsets	是否使用提供的 startOffset 值来重置消费者的 offset 值	false
startOffset	用来设置新建组的起始 offset，该值也会在 resetOffsets 开始时被使用	null
enableDlq	该参数设置为 true 时，将为消费者启用 DLQ 行为，引起错误的消息将被发送到名为 error.<destination>.<group>的主题中去	false

生产者配置

下面这些配置仅对 Kafka 输出通道的绑定有效，它们以 spring.cloud.stream.kafka.bindings.<channelName>.producer.格式作为前缀。

参数名	说明	默认值
bufferSize	Kafka 批量发送前的缓存数据上限，以字节为单位	16384
sync	该参数用来设置 Kafka 消息生产者的发送模式，默认为 false，即采用 async 配置，允许批量发送数据。当设置为 true 时，将采用 sync 配置，消息将不会被批量发送，而是一条一条地发送	false

续表

参数名	说明	默认值
batchTimeout	消息生产者批量发送时，为了积累更多发送数据而设置的等待时间。通常情况下，生产者基本不会等待，而是直接发送所有在前一批次发送时积累的消息数据。当我们设置一个非 0 值时，可以以延迟为代价来增加系统的吞吐量	0

第 11 章

分布式服务跟踪：Spring Cloud Sleuth

通过之前各章节介绍的 Spring Cloud 组件，实际上我们已经能够通过使用它们搭建起一个基础的微服务架构系统来实现业务需求了。但是，随着业务的发展，系统规模也会变得越来越大，各微服务间的调用关系也变得越来越错综复杂。通常一个由客户端发起的请求在后端系统中会经过多个不同的微服务调用来协同产生最后的请求结果，在复杂的微服务架构系统中，几乎每一个前端请求都会形成一条复杂的分布式服务调用链路，在每条链路中任何一个依赖服务出现延迟过高或错误的时候都有可能引起请求最后的失败。这时候，对于每个请求，全链路调用的跟踪就变得越来越重要，通过实现对请求调用的跟踪可以帮助我们快速发现错误根源以及监控分析每条请求链路上的性能瓶颈等。

针对上面所述的分布式服务跟踪问题，Spring Cloud Sleuth 提供了一套完整的解决方案。在本章中，我们将详细介绍如何使用 Spring Cloud Sleuth 来为微服务架构增加分布式服务跟踪的能力。

快速入门

在介绍各种概念与原理之前，我们先通过实现一个简单的示例，为存在服务调用的应用增加一些 Sleuth 的配置以实现基本的服务跟踪功能，以此来对 Spring Cloud Sleuth 有一个初步的了解，随后再逐步展开，介绍实现过程中的各个细节。

准备工作

在引入 Sleuth 之前，我们先按照之前章节学习的内容来做一些准备工作，构建一些基

础的设施和应用。

- 服务注册中心：eureka-server，这里不做赘述，直接使用之前构建的工程即可。
- 微服务应用：trace-1，实现一个 REST 接口/trace-1，调用该接口后将触发对 trace-2 应用的调用。具体实现如下所述。
  - 创建一个基础的 Spring Boot 应用，在 pom.xml 中增加下面的依赖：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/>
</parent>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-ribbon</artifactId>
</dependency>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

  - 创建应用主类，实现/trace-1 接口，并使用 RestTemplate 调用 trace-2 应用的接口。具体如下：

```
@RestController
@EnableDiscoveryClient
@SpringBootApplication
public class TraceApplication {
```

```
    private final Logger logger = Logger.getLogger(getClass());

    @Bean
    @LoadBalanced
    RestTemplate restTemplate() {
        return new RestTemplate();
    }

    @RequestMapping(value = "/trace-1", method = RequestMethod.GET)
    public String trace() {
        logger.info("===call trace-1===");
        return restTemplate().getForEntity("http://trace-2/trace-2",
String.class).getBody();
    }

    public static void main(String[] args) {
        SpringApplication.run(TraceApplication.class, args);
    }

}
```

- 在 application.properties 中，将 eureka.client.serviceUrl.defaultZone 参数指向 eureka-server 的地址，具体如下：

```
spring.application.name=trace-1
server.port=9101
eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

- 微服务应用：trace-2，实现一个 REST 接口/trace-2，供 trace-1 调用。具体实现如下所示。
  - 创建一个基础的 Spring Boot 应用，pom.xml 中的依赖与 trace-1 相同。
  - 创建应用主类，并实现/trace-2 接口，具体实现如下：

```
@RestController
@EnableDiscoveryClient
@SpringBootApplication
public class TraceApplication {

    private final Logger logger = Logger.getLogger(getClass());

    @RequestMapping(value = "/trace-2", method = RequestMethod.GET)
    public String trace() {
        logger.info("===<call trace-2>===");
        return "Trace";
    }
```

```
    public static void main(String[] args) {
        SpringApplication.run(TraceApplication.class, args);
    }

}
```

- 在application.properties中，将eureka.client.serviceUrl.defaultZone参数指向 eureka-server 的地址，另外还需要设置不同的应用名和端口号，具体如下：

```
spring.application.name=trace-2
server.port=9102
eureka.client.serviceUrl.defaultZone=http://localhost:1111/eureka/
```

在实现了上面的内容之后，我们可以将 eureka-server、trace-1、trace-2 三个应用都启动起来，并通过 postman 或 curl 等工具来对 trace-1 的接口发送请求 http://localhost:9101/trace-1。可以得到返回值 Trace，同时还能在它们的控制台中分别获得下面的输出：

```
-- trace-1
INFO 25272 --- [nio-9101-exec-2] ication$$EnhancerBySpringCGLIB$$36e12c68 :
===<call trace-1>===

-- trace-2
INFO 7136 --- [nio-9102-exec-1] ication$$EnhancerBySpringCGLIB$$52a02f0b : ===<call
trace-2>===
```

实现跟踪

在完成了准备工作之后，接下来我们开始进行本章的主题内容，为上面的 trace-1 和 trace-2 添加服务跟踪功能。通过 Spring Cloud Sleuth 的封装，我们为应用增加服务跟踪能力的操作非常简单，只需在 trace-1 和 trace-2 的 pom.xml 依赖管理中增加 spring-cloud-starter-sleuth 依赖即可，具体如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-sleuth</artifactId>
</dependency>
```

到这里，实际上我们已经为 trace-1 和 trace-2 实现服务跟踪做好了基础的准备，重启 trace-1 和 trace-2，再对 trace-1 的接口发送请求 http://localhost:9101/trace-1。此时，我们可以从它们的控制台输出中窥探到 Sleuth 的一些端倪。

```
-- trace-1
INFO [trace-1,f410ab57afd5c145,a9f2118fa2019684,false] 25028 --- [nio-9101-exec-1]
```

```
ication$$EnhancerBySpringCGLIB$$d8228493 : ===<call trace-1>===

    -- trace-2
    INFO [trace-2,f410ab57afd5c145,e9a377dc2268bc29,false] 23112 --- [nio-9102-exec-1]
ication$$EnhancerBySpringCGLIB$$e6cb4078 : ===<call trace-2>===
```

从上面的控制台输出内容中，我们可以看到多了一些形如 [trace-1, f410ab57afd5c145,a9f2118fa2019684,false]的日志信息，而这些元素正是实现分布式服务跟踪的重要组成部分，每个值的含义如下所述。

- 第一个值：trace-1，它记录了应用的名称，也就是 application.properties 中 spring.application.name 参数配置的属性。
- 第二个值：f410ab57afd5c145，Spring Cloud Sleuth 生成的一个 ID，称为 Trace ID，它用来标识一条请求链路。一条请求链路中包含一个 Trace ID，多个 Span ID。
- 第三个值：a9f2118fa2019684，Spring Cloud Sleuth 生成的另外一个 ID，称为 Span ID，它表示一个基本的工作单元，比如发送一个 HTTP 请求。
- 第四个值：false，表示是否要将该信息输出到 Zipkin 等服务中来收集和展示。

上面四个值中的 Trace ID 和 Span ID 是 Spring Cloud Sleuth 实现分布式服务跟踪的核心。在一次服务请求链路的调用过程中，会保持并传递同一个 Trace ID，从而将整个分布于不同微服务进程中的请求跟踪信息串联起来。以上面输出内容为例，trace-1 和 trace-2 同属于一个前端服务请求来源，所以它们的 Trace ID 是相同的，处于同一条请求链路中。

跟踪原理

分布式系统中的服务跟踪在理论上并不复杂，它主要包括下面两个关键点。

- 为了实现请求跟踪，当请求发送到分布式系统的入口端点时，只需要服务跟踪框架为该请求创建一个唯一的跟踪标识，同时在分布式系统内部流转的时候，框架始终保持传递该唯一标识，直到返回给请求方为止，这个唯一标识就是前文中提到的 Trace ID。通过 Trace ID 的记录，我们就能将所有请求过程的日志关联起来。
- 为了统计各处理单元的时间延迟，当请求到达各个服务组件时，或是处理逻辑到达某个状态时，也通过一个唯一标识来标记它的开始、具体过程以及结束，该标识就是前文中提到的 Span ID。对于每个 Span 来说，它必须有开始和结束两个节点，通过记录开始 Span 和结束 Span 的时间戳，就能统计出该 Span 的时间延迟，除了时间戳记录之外，它还可以包含一些其他元数据，比如事件名称、请求信息等。

在快速入门示例中，我们轻松实现了日志级别的跟踪信息接入，这完全归功于

spring-cloud-starter-sleuth 组件的实现。在 Spring Boot 应用中，通过在工程中引入 spring-cloud-starter-sleuth 依赖之后，它会自动为当前应用构建起各通信通道的跟踪机制，比如：

- 通过诸如 RabbitMQ、Kafka（或者其他任何 Spring Cloud Stream 绑定器实现的消息中间件）传递的请求。
- 通过 Zuul 代理传递的请求。
- 通过 RestTemplate 发起的请求。

在快速入门示例中，由于 trace-1 对 trace-2 发起的请求是通过 RestTemplate 实现的，所以 spring-cloud-starter-sleuth 组件会对该请求进行处理。在发送到 trace-2 之前，Sleuth 会在该请求的 Header 中增加实现跟踪需要的重要信息，主要有下面这几个（更多关于头信息的定义可以通过查看 org.springframework.cloud.sleuth.Span 的源码获取）。

- X-B3-TraceId：一条请求链路（Trace）的唯一标识，必需的值。
- X-B3-SpanId：一个工作单元（Span）的唯一标识，必需的值。
- X-B3-ParentSpanId：标识当前工作单元所属的上一个工作单元，Root Span（请求链路的第一个工作单元）的该值为空。
- X-B3-Sampled：是否被抽样输出的标志，1 表示需要被输出，0 表示不需要被输出。
- X-Span-Name：工作单元的名称。

可以通过对 trace-2 的实现做一些修改来输出这些头部信息，具体如下：

```
@RequestMapping(value = "/trace-2", method = RequestMethod.GET)
public String trace(HttpServletRequest request) {
    logger.info("===<call trace-2, TraceId={}, SpanId={}>===",
            request.getHeader("X-B3-TraceId"), request.getHeader("X-B3-SpanId"));
    return "Trace";
}
```

通过上面的改造，我们再运行快速入门的示例内容，并发起对 trace-1 的接口访问，可以得到如下输出内容。其中在 trace-2 的控制台中，输出了当前正在处理的 TraceID 和 SpanId 信息。

```
    -- trace-1
    INFO [trace-1,a6e9175ffd5d2c88,8524f519b8a9e399,true] 10532 --- [nio-9101-exec-2]
icationEnhancerBySpringCGLIB27aa9624 : ===<call trace-1>===

    -- trace-2
    INFO [trace-2,a6e9175ffd5d2c88,ce60dcf1e2ed918f,true] 1208 --- [nio-9102-exec-3]
icationEnhancerBySpringCGLIBa7d84797 : ===<call trace-2, TraceId=a6e9175ffd5d2c88,
SpanId=be4949ec115e554e>===
```

为了更直观地观察跟踪信息，我们还可以在 application.properties 中增加下面的配置：

```
logging.level.org.springframework.web.servlet.DispatcherServlet=DEBUG
```

通过将 Spring MVC 的请求分发日志级别调整为 DEBUG 级别，我们可以看到更多跟踪信息：

```
-- trace-1
    2016-11-27 09:26:52.663 DEBUG [trace-1,a6e9175ffd5d2c88,a6e9175ffd5d2c88,true]
10532 --- [nio-9101-exec-2] o.s.web.servlet.DispatcherServlet       :
DispatcherServlet with name 'dispatcherServlet' processing GET request for [/trace-1]
    2016-11-27 09:26:52.666 DEBUG [trace-1,a6e9175ffd5d2c88,a6e9175ffd5d2c88,true]
10532 --- [nio-9101-exec-2] o.s.web.servlet.DispatcherServlet       : Last-Modified
value for [/trace-1] is: -1
    2016-11-27 09:26:52.685 DEBUG [trace-1,a6e9175ffd5d2c88,8524f519b8a9e399,true]
10532 --- [nio-9101-exec-2] o.s.web.servlet.DispatcherServlet       : Null ModelAndView
returned to DispatcherServlet with name 'dispatcherServlet': assuming HandlerAdapter
completed request handling
    2016-11-27 09:26:52.685 DEBUG [trace-1,a6e9175ffd5d2c88,a6e9175ffd5d2c88,true]
10532 --- [nio-9101-exec-2] o.s.web.servlet.DispatcherServlet       : Successfully
completed request

-- trace-2
    2016-11-27 09:26:52.673 DEBUG [trace-2,a6e9175ffd5d2c88,be4949ec115e554e,true]
1208 --- [nio-9102-exec-3] o.s.web.servlet.DispatcherServlet       : DispatcherServlet
with name 'dispatcherServlet' processing GET request for [/trace-2]
    2016-11-27 09:26:52.679 DEBUG [trace-2,a6e9175ffd5d2c88,be4949ec115e554e,true]
1208 --- [nio-9102-exec-3] o.s.web.servlet.DispatcherServlet       : Last-Modified
value for [/trace-2] is: -1
    2016-11-27 09:26:52.682 DEBUG [trace-2,a6e9175ffd5d2c88,ce60dcf1e2ed918f,true]
1208 --- [nio-9102-exec-3] o.s.web.servlet.DispatcherServlet       : Null ModelAndView
returned to DispatcherServlet with name 'dispatcherServlet': assuming HandlerAdapter
completed request handling
    2016-11-27 09:26:52.683 DEBUG [trace-2,a6e9175ffd5d2c88,be4949ec115e554e,true]
1208 --- [nio-9102-exec-3] o.s.web.servlet.DispatcherServlet       : Successfully
completed request
```

抽样收集

通过 Trace ID 和 Span ID 已经实现了对分布式系统中的请求跟踪，而记录的跟踪信息最终会被分析系统收集起来，并用来实现对分布式系统的监控和分析功能，比如，预警延迟过长的请求链路、查询请求链路的调用明细等。此时，我们在对接分析系统时就会碰到一个问题：分析系统在收集跟踪信息的时候，需要收集多少跟踪信息才合适呢？

理论上来说，我们收集的跟踪信息越多就可以越好地反映出系统的实际运行情况，并给出更精准的预警和分析。但是在高并发的分布式系统运行时，大量的请求调用会产生海

量的跟踪日志信息，如果收集过多的跟踪信息将会对整个分布式系统的性能造成一定的影响，同时保存大量的日志信息也需要不少的存储开销。所以，在 Sleuth 中采用了抽象收集的方式来为跟踪信息打上收集标记，也就是我们之前在日志信息中看到的第 4 个布尔类型的值，它代表了该信息是否要被后续的跟踪信息收集器获取和存储。

Sleuth 中的抽样收集策略是通过 Sampler 接口实现的，它的定义如下：

```
public interface Sampler {
    /**
     * @return true if the span is not null and should be exported to the tracing
system
     */
    boolean isSampled(Span span);
}
```

通过实现 isSampled 方法，Spring Cloud Sleuth 会在产生跟踪信息的时候调用它来为跟踪信息生成是否要被收集的标志。需要注意的是，即使 isSampled 返回了 false，它仅代表该跟踪信息不被输出到后续对接的远程分析系统（比如 Zipkin），对于请求的跟踪活动依然会进行，所以我们在日志中还是能看到收集标识为 false 的记录。

默认情况下，Sleuth 会使用 PercentageBasedSampler 实现的抽样策略，以请求百分比的方式配置和收集跟踪信息。我们可以通过在 application.properties 中配置下面的参数对其百分比值进行设置，它的默认值为 0.1，代表收集 10%的请求跟踪信息。

```
spring.sleuth.sampler.percentage=0.1
```

在开发调试期间，通常会收集全部跟踪信息并输出到远程仓库，我们可以将其值设置为 1，或者也可以通过创建 AlwaysSampler 的 Bean（它实现的 isSampled 方法始终返回 true）来覆盖默认的 PercentageBasedSampler 策略，比如：

```
@Bean
public AlwaysSampler defaultSampler() {
    return new AlwaysSampler();
}
```

在实际使用时，通过与 Span 对象中存储信息的配合，我们可以根据实际情况做出更贴近需求的抽样策略，比如实现一个仅包含指定 Tag 的抽样策略：

```
public class TagSampler implements Sampler {

    private String tag;

    public TagSampler(String tag) {
        this.tag = tag;
    }
```

```
    @Override
    public boolean isSampled(Span span) {
        return span.tags().get(tag) != null;
    }
}
```

由于跟踪日志信息数据的价值往往仅在最近的一段时间内非常有用，比如一周。那么我们在设计抽样策略时，主要考虑在不对系统造成明显性能影响的情况下，以在日志保留时间窗内充分利用存储空间的原则来实现抽样策略。

与 Logstash 整合

通过之前的准备与整合，我们已经为 `trace-1` 和 `trace-2` 引入了 Spring Cloud Sleuth 的基础模块 `spring-cloud-starter-sleuth`，实现了在各个微服务的日志信息中添加跟踪信息的功能。但是，由于日志文件都离散地存储在各个服务实例的文件系统之上，仅仅通过查看日志文件来分析我们的请求链路依然是一件相当麻烦的事，所以我们还需要一些工具来帮助集中收集、存储和搜索这些跟踪信息。引入基于日志的分析系统是一个不错的选择，比如 ELK 平台，它可以轻松地帮助我们收集和存储这些跟踪日志，同时在需要的时候我们也可以根据 Trace ID 来轻松地搜索出对应请求链路相关的明细日志。

ELK 平台主要由 ElasticSearch、Logstash 和 Kibana 三个开源工具组成。

- ElasticSearch 是一个开源分布式搜索引擎，它的特点有：分布式，零配置，自动发现，索引自动分片，索引副本机制，RESTful 风格接口，多数据源，自动搜索负载等。
- Logstash 是一个完全开源的工具，它可以对日志进行收集、过滤，并将其存储供以后使用。
- Kibana 也是一个开源和免费的工具，它可以为 Logstash 和 ElasticSearch 提供日志分析友好的 Web 界面，可以帮助汇总、分析和搜索重要数据日志。

Spring Cloud Sleuth 在与 ELK 平台整合使用时，实际上只要实现与负责日志收集的 Logstash 完成数据对接即可，所以我们需要为 Logstash 准备 JSON 格式的日志输出。由于 Spring Boot 应用默认使用 logback 来记录日志，而 Logstash 自身也有对 logback 日志工具的支持工具，所以我们可以直接通过在 logback 的配置中增加对 Logstash 的 Appender，就能非常方便地将日志转换成以 JSON 的格式存储和输出了。

下面我们来详细介绍一下在快速入门示例的基础上，如何实现面向 Logstash 的日志输出配置。

- 在 `pom.xml` 依赖中引入 `logstash-logback-encoder` 依赖，具体如下：

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>4.6</version>
</dependency>
```

- 在工程/resource 目录下创建 bootstrap.properties 配置文件，将 spring.application.name=trace-1 配置移动到该文件中去。由于 logback-spring.xml 的加载在 application.properties 之前，所以之前的配置 logback-spring.xml 无法获取 spring.application.name 属性，因此这里将该属性移动到最先加载的 bootstrap.properties 配置文件中。
- 在工程/resource 目录下创建 logback 配置文件 logback-spring.xml，具体内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <include resource="org/springframework/boot/logging/logback/defaults.xml"/>

  <springProperty scope="context" name="springAppName" source="spring.
application.name"/>
  <!-- 日志在工程中的输出位置 -->
  <property name="LOG_FILE" value="${BUILD_FOLDER:-build}/${springAppName}"/>
<!-- 控制台的日志输出样式 -->
  <property name="CONSOLE_LOG_PATTERN"
          value="%clr(%d{yyyy-MM-dd HH:mm:ss.SSS}){faint} %clr(${LOG_LEVEL_PATTERN:
-%5p}) %clr([${springAppName:-},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-Span-Expo
rt:-}]){yellow} %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %c
lr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wE
x}"/>

  <!-- 控制台 Appender -->
  <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
    <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
      <level>INFO</level>
    </filter>
    <encoder>
      <pattern>${CONSOLE_LOG_PATTERN}</pattern>
      <charset>utf8</charset>
    </encoder>
  </appender>

  <!-- 为 logstash 输出的 JSON 格式的 Appender -->
  <appender name="logstash" class="ch.qos.logback.core.rolling.RollingFileAppender">
```

```
        <file>${LOG_FILE}.json</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>${LOG_FILE}.json.%d{yyyy-MM-dd}.gz</fileNamePattern>
            <maxHistory>7</maxHistory>
        </rollingPolicy>
        <encoder class="net.logstash.logback.encoder.LoggingEventCompositeJsonEncoder">
            <providers>
                <timestamp>
                    <timeZone>UTC</timeZone>
                </timestamp>
                <pattern>
                    <pattern>
                        {
                        "severity": "%level",
                        "service": "${springAppName:-}",
                        "trace": "%X{X-B3-TraceId:-}",
                        "span": "%X{X-B3-SpanId:-}",
                        "exportable": "%X{X-Span-Export:-}",
                        "pid": "${PID:-}",
                        "thread": "%thread",
                        "class": "%logger{40}",
                        "rest": "%message"
                        }
                    </pattern>
                </pattern>
            </providers>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="console"/>
        <appender-ref ref="logstash"/>
    </root>
```

对 Logstash 的支持主要通过名为 `logstash` 的 Appender 实现，内容并不复杂，主要是对日志信息的格式化处理，上面为了方便调试和查看，我们先将 JSON 格式的日志输出到文件中。

完成上面的改造之后，我们再将快速入门的示例运行起来，并发起对 `trace-1` 的接口访问。此时可以在 `trace-1` 和 `trace-2` 的工程目录下发现有一个 `build` 目录，下面分别创建了以各自应用名称命名的 JSON 文件，该文件就是在 `logback-spring.xml` 中配置的名为 `logstash` 的 Appender 输出的日志文件，其中记录了类似下面格式的 JSON

日志：

```
    {"@timestamp":"2016-12-04T06:57:58.970+00:00","severity":"INFO","service":"trac
e-1","trace":"589ee5f7b860132f","span":"a9e891273affb7fc","exportable":"false","pid
":"19756","thread":"http-nio-9101-exec-1","class":"c.d.TraceApplication$$EnhancerBy
SpringCGLIB$$a9604da6","rest":"===<call trace-1>===="}
    {"@timestamp":"2016-12-04T06:57:59.061+00:00","severity":"INFO","service":"trac
e-1","trace":"589ee5f7b860132f","span":"2df8511ddf3d79a2","exportable":"false","pid
":"19756","thread":"http-nio-9101-exec-1","class":"o.s.c.a.AnnotationConfigApplicat
ionContext","rest":"Refreshing org.springframework.context.annotation.Annotation-
ConfigApplicationContext@64951f38: startup date [Sun Dec 04 14:57:59 CST 2016]; parent:
org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationCon
text@4b8c8f15"}
```

除了可以通过上面的方式生成 JSON 文件外，还可以使用 LogstashTcpSocket-Appender 将日志内容直接通过 Tcp Socket 输出到 Logstash 服务端，比如：

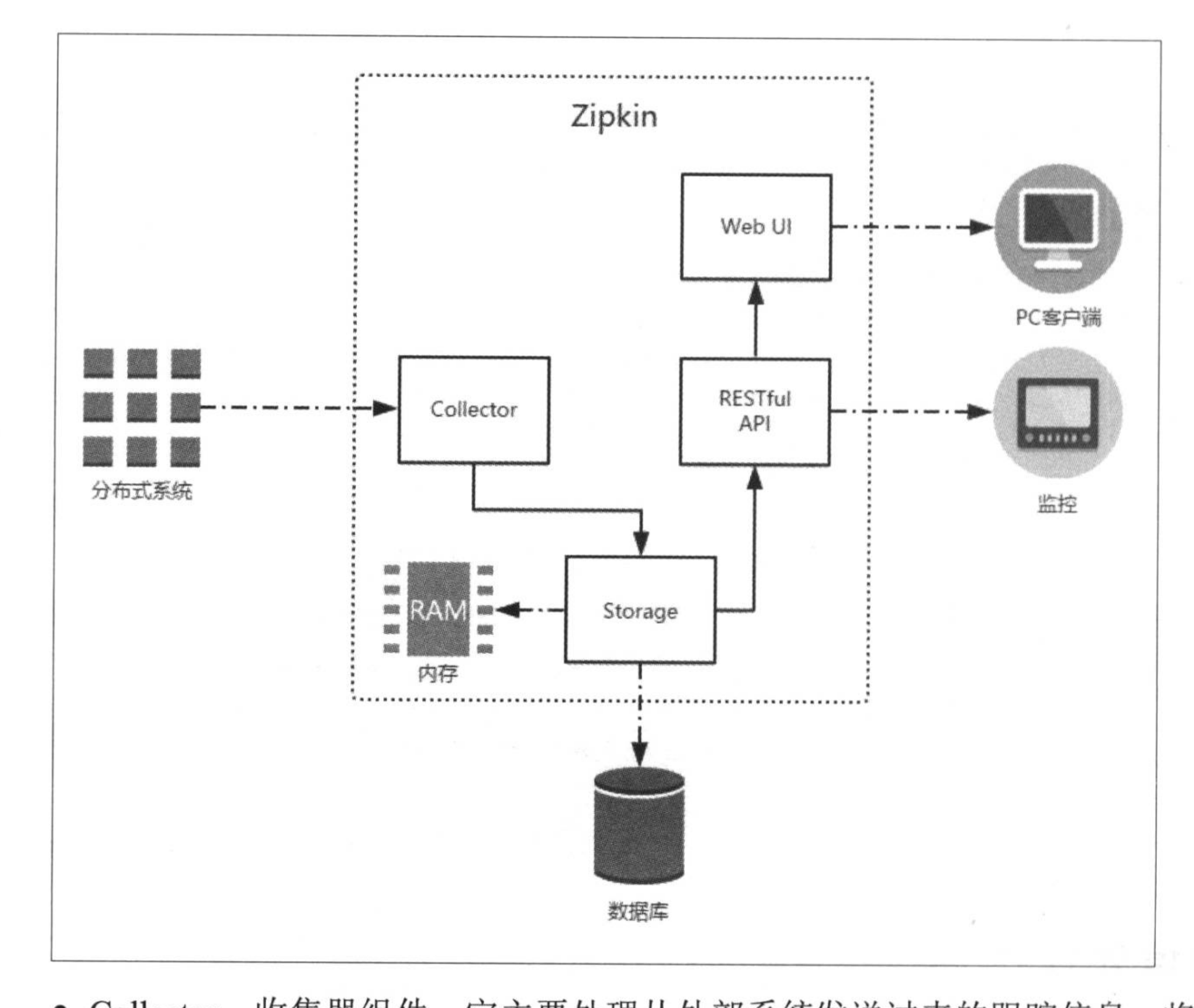
```
    <appender name="logstash" class="net.logstash.logback.appender.LogstashTcpSocket
Appender">
        <destination>127.0.0.1:9250</destination>
        ...
    </appender>
```

与 Zipkin 整合

虽然通过 ELK 平台提供的收集、存储、搜索等强大功能，我们对跟踪信息的管理和使用已经变得非常便利。但是，在 ELK 平台中的数据分析维度缺少对请求链路中各阶段时间延迟的关注，很多时候我们追溯请求链路的一个原因是为了找出整个调用链路中出现延迟过高的瓶颈源，或为了实现对分布式系统做延迟监控等与时间消耗相关的需求，这时候类似 ELK 这样的日志分析系统就显得有些乏力了。对于这样的问题，我们就可以引入 Zipkin 来得以轻松解决。

Zipkin 是 Twitter 的一个开源项目，它基于 Google Dapper 实现。我们可以使用它来收集各个服务器上请求链路的跟踪数据，并通过它提供的 REST API 接口来辅助查询跟踪数据以实现对分布式系统的监控程序，从而及时发现系统中出现的延迟升高问题并找出系统性能瓶颈的根源。除了面向开发的 API 接口之外，它还提供了方便的 UI 组件来帮助我们直观地搜索跟踪信息和分析请求链路明细，比如可以查询某段时间内各用户请求的处理时间等。

下图展示了 Zipkin 的基础架构，它主要由 4 个核心组件构成。



- Collector：收集器组件，它主要处理从外部系统发送过来的跟踪信息，将这些信息转换为 Zipkin 内部处理的 Span 格式，以支持后续的存储、分析、展示等功能。
- Storage：存储组件，它主要处理收集器接收到的跟踪信息，默认会将这些信息存储在内存中。我们也可以修改此存储策略，通过使用其他存储组件将跟踪信息存储到数据库中。
- RESTful API：API 组件，它主要用来提供外部访问接口。比如给客户端展示跟踪信息，或是外接系统访问以实现监控等。
- Web UI：UI 组件，基于 API 组件实现的上层应用。通过 UI 组件，用户可以方便而又直观地查询和分析跟踪信息。

HTTP 收集

在 Spring Cloud Sleuth 中对 Zipkin 的整合进行了自动化配置的封装，所以我们可以很轻松地引入和使用它。下面我们来详细介绍一下 Sleuth 与 Zipkin 的基础整合过程，主要分为以下两步。

第一步：搭建 Zipkin Server

- 创建一个基础的 Spring Boot 应用，命名为 zipkin-server，并在 pom.xml 中引入 Zipkin Server 的相关依赖，具体如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.7.RELEASE</version>
    <relativePath/>
</parent>
<dependencies>
    <dependency>
        <groupId>io.zipkin.java</groupId>
        <artifactId>zipkin-server</artifactId>
    </dependency>
    <dependency>
        <groupId>io.zipkin.java</groupId>
        <artifactId>zipkin-autoconfigure-ui</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Brixton.SR5</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

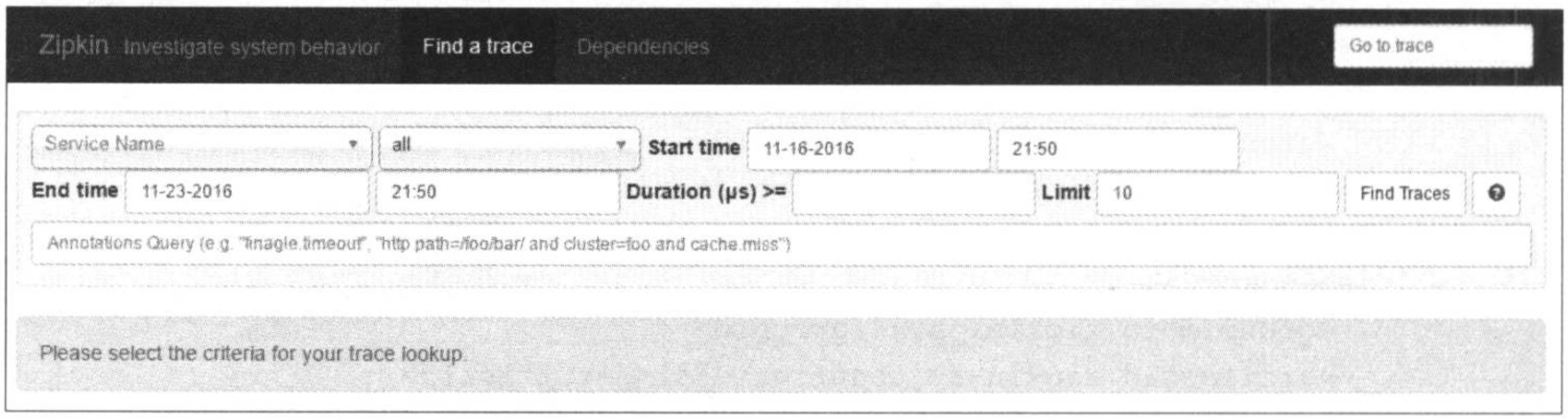
- 创建应用主类 ZipkinApplication，使用@EnableZipkinServer 注解来启动 Zipkin Server，具体如下：

```
@EnableZipkinServer
@SpringBootApplication
public class ZipkinApplication {

    public static void main(String[] args) {
        SpringApplication.run(ZipkinApplication.class, args);
    }

}
```

- 在 application.properties 中做一些简单配置，比如设置服务端口号为 9411（客户端整合时，自动化配置会连接 9411 端口，所以在服务端设置了端口为 9411 的话，客户端可以省去这个配置）。

```
spring.application.name=zipkin-server
server.port=9411
```

创建完上述工程之后，我们将其启动起来，并访问 http://localhost:9411/，可以看到如下图所示的 Zipkin 管理页面：



第二步：为应用引入和配置 Zipkin 服务

在完成了 Zipkin Server 的搭建之后，我们还需要对应用做一些配置，以实现将跟踪信息输出到 Zipkin Server。我们以之前实现的 trace-1 和 trace-2 为例，对它们做以下改造。

- 在 trace-1 和 trace-2 的 pom.xml 中引入 spring-cloud-sleuth-zipkin 依赖，具体如下所示。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
</dependency>
```

- 在 trace-1 和 trace-2 的 application.properties 中增加 Zipkin Server 的配置信息，具体如下所示（如果在 zip-server 应用中，我们将其端口设置为 9411，并且均在本地调试的话，也可以不配置该参数，因为默认值就是 http://localhost:9411）。

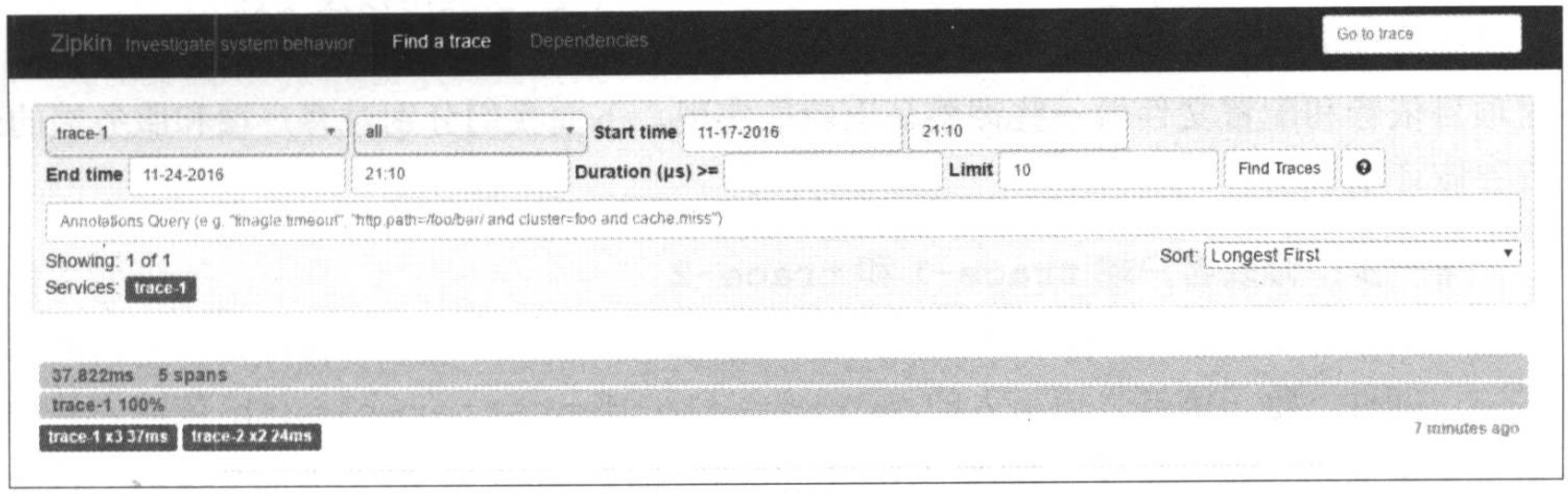
```
spring.zipkin.base-url=http://localhost:9411
```

测试与分析

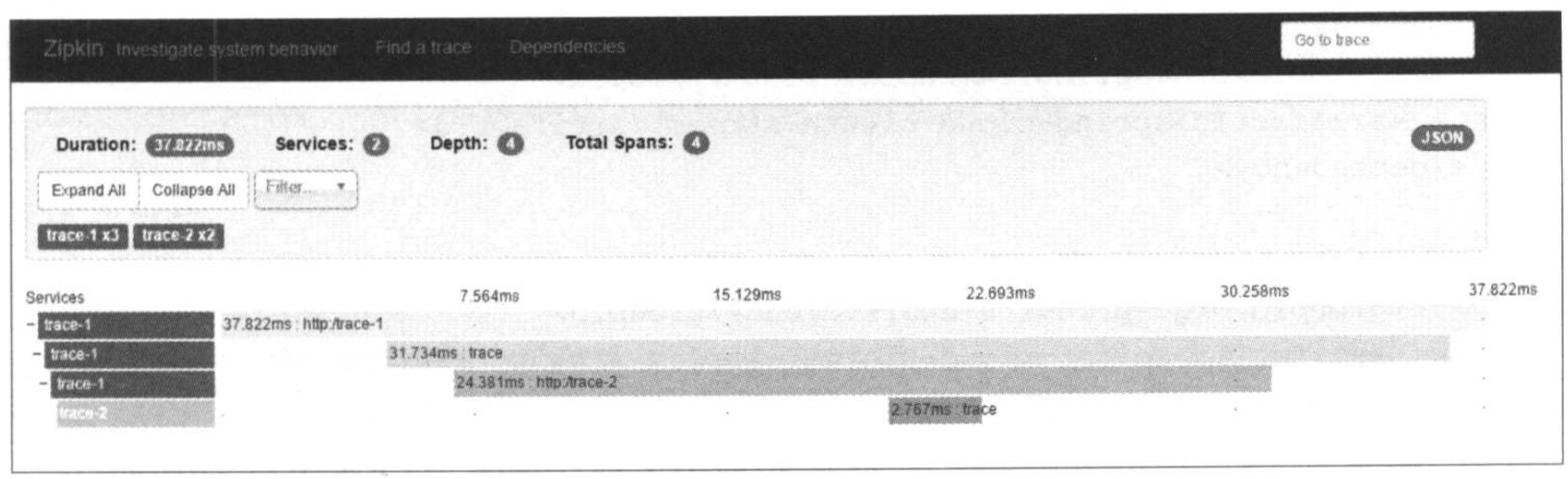
到这里我们已经完成了接入 Zipkin Server 的所有基本工作，可以继续将 eureka-server、trace-1 和 trace-2 启动起来，然后做一些测试，以对它的运行机

制有一些初步的理解。

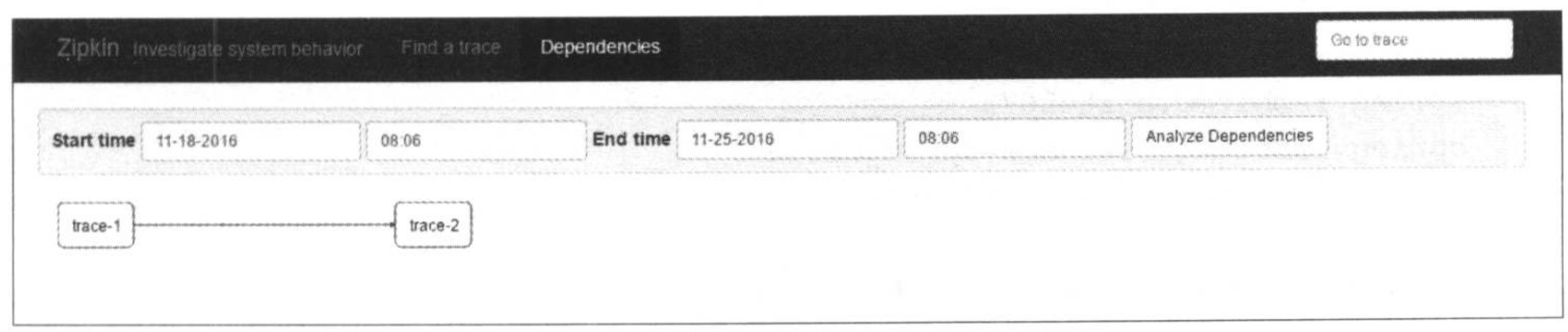
我们先来向 trace-1 的接口发送几个请求 http://localhost:9101/trace-1。当在日志中出现跟踪信息的最后一个值为 true 的时候，说明该跟踪信息会输出给 Zipkin Server，所以此时可以在 Zipkin Server 的管理页面中选择合适的查询条件，单击 Find Traces 按钮，就可以查询出刚才在日志中出现的跟踪信息了（也可以根据日志中的 Trace ID，在页面右上角的输入框中来搜索），页面如下所示。



单击下方 trace-1 端点的跟踪信息，还可以得到 Sleuth 跟踪到的详细信息，其中包括我们关注的请求时间消耗等。



单击导航栏中的 Dependencies 菜单，还可以查看 Zipkin Server 根据跟踪信息分析生成的系统请求链路依赖关系图，如下图所示。



消息中间件收集

Spring Cloud Sleuth 在整合 Zipkin 时，不仅实现了以 HTTP 的方式收集跟踪信息，还实现了通过消息中间件来对跟踪信息进行异步收集的封装。通过结合 Spring Cloud Stream，我们可以非常轻松地让应用客户端将跟踪信息输出到消息中间件上，同时 Zipkin 服务端从消息中间件上异步地消费这些跟踪信息。

接下来，我们基于之前实现的 `trace-1` 和 `trace-2` 应用以及 `zipkin-server` 服务端做一些改造，以实现通过消息中间件来收集跟踪信息。改造的内容非常简单，只需要对项目依赖和配置文件做一些调整马上就能实现。下面我们分别对客户端和服务端的改造内容做详细说明。

第一步：修改客户端 `trace-1` 和 `trace-2`

- 为了让 `trace-1` 和 `trace-2` 在产生跟踪信息之后，能够将抽样记录输出到消息中间件，除了需要之前引入的 `spring-cloud-starter-sleuth` 依赖之外，还需要引入 Zipkin 对 Spring Cloud Stream 的扩展依赖 `spring-cloud-sleuth-stream` 以及基于 Spring Cloud Stream 实现的消息中间件绑定器依赖。以使用 RabbitMQ 为例，我们可以加入如下依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-stream</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

- 在 `application.properties` 配置中去掉 HTTP 方式实现时使用的 `spring.zipkin.base-url` 参数，并根据实际部署情况，增加消息中间件的相关配置，比如下面这些关于 RabbitMQ 的配置信息：

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=springcloud
spring.rabbitmq.password=123456
```

第二步：修改 `zipkin-server` 服务端

为了让 `zipkin-server` 服务端能够从消息中间件中获取跟踪信息，我们只需要在 `pom.xml` 中引入针对消息中间件收集封装的服务端依赖 `spring-cloud-sleuth-zipkin-`

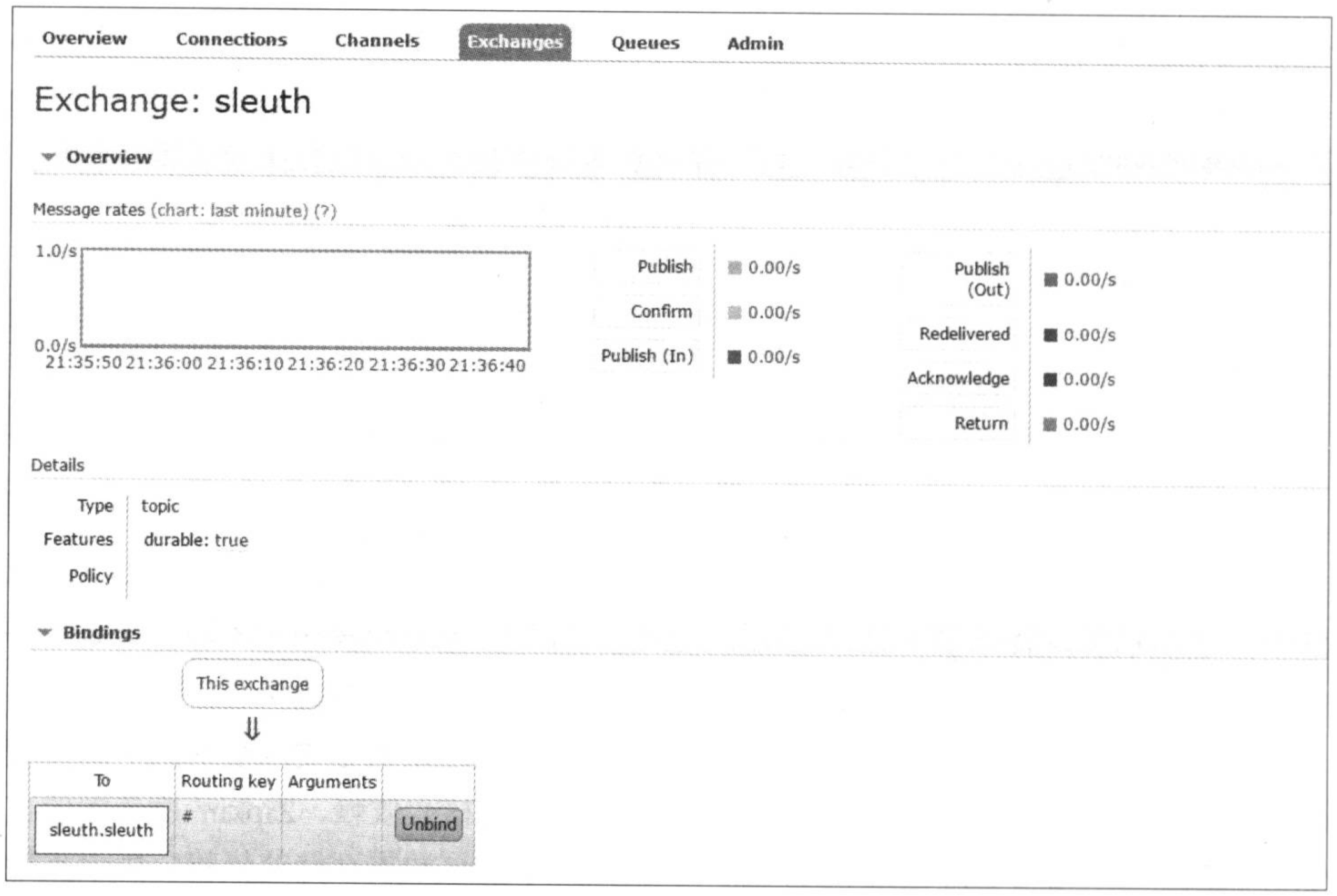
stream，同时为了支持具体使用的消息中间件，我们还需要引入针对消息中间件的绑定器实现。比如以使用 RabbitMQ 为例，我们可以在依赖中增加如下内容：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>

<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</artifactId>
</dependency>
```

其中，spring-cloud-sleuth-zipkin-stream 依赖是实现从消息中间件收集跟踪信息的核心封装，其中包含了用于整合消息中间件的核心依赖、Zipkin 服务端的核心依赖，以及一些其他通常会被使用的依赖（比如，用于扩展数据存储的依赖、用于支持测试的依赖等）。但是，需要注意的是，这个包里并没有引入 zipkin 的前端依赖 zipkin-autoconfigure-ui，为了方便使用，我们在这里也引用了它。

测试与分析

在完成了上述改造内容之后，我们继续将 eureka-server、trace-1 和 trace-2、zipkin-server 都启动起来，同时确保 RabbitMQ 也处于运行状态。此时，我们可以在 RabbitMQ 的控制页面中看到一个名为 sleuth 的交换器，它就是 Zipkin 的消息中间件收集器实现使用的默认主题。



最后，我们使用之前的验证方法，通过向 trace-1 的接口发送几个请求 http://localhost:9101/trace-1。当有被抽样收集的跟踪信息时（调试时我们可以设置 AlwaysSampler 抽样机制来让每个跟踪信息都被收集），我们可以在 RabbitMQ 的控制页面中发现有消息被发送到了 sleuth 交换器中。同时再到 Zipkin 服务端的 Web 页面中也能够搜索到相应的跟踪信息，那么我们使用消息中间件来收集跟踪信息的任务到这里就完成了。

收集原理

在本节内容之前，我们已经对如何引入 Sleuth 跟踪信息和搭建 Zipkin 服务端分析跟踪延迟的过程做了详细的介绍，相信大家对于 Sleuth 和 Zipkin 已经有了一定的感性认识。接下来，我们介绍一下关于 Zipkin 收集跟踪信息的过程细节，以帮助我们更好地理解 Sleuth 生产跟踪信息以及输出跟踪信息的整体过程和工作原理。

数据模型

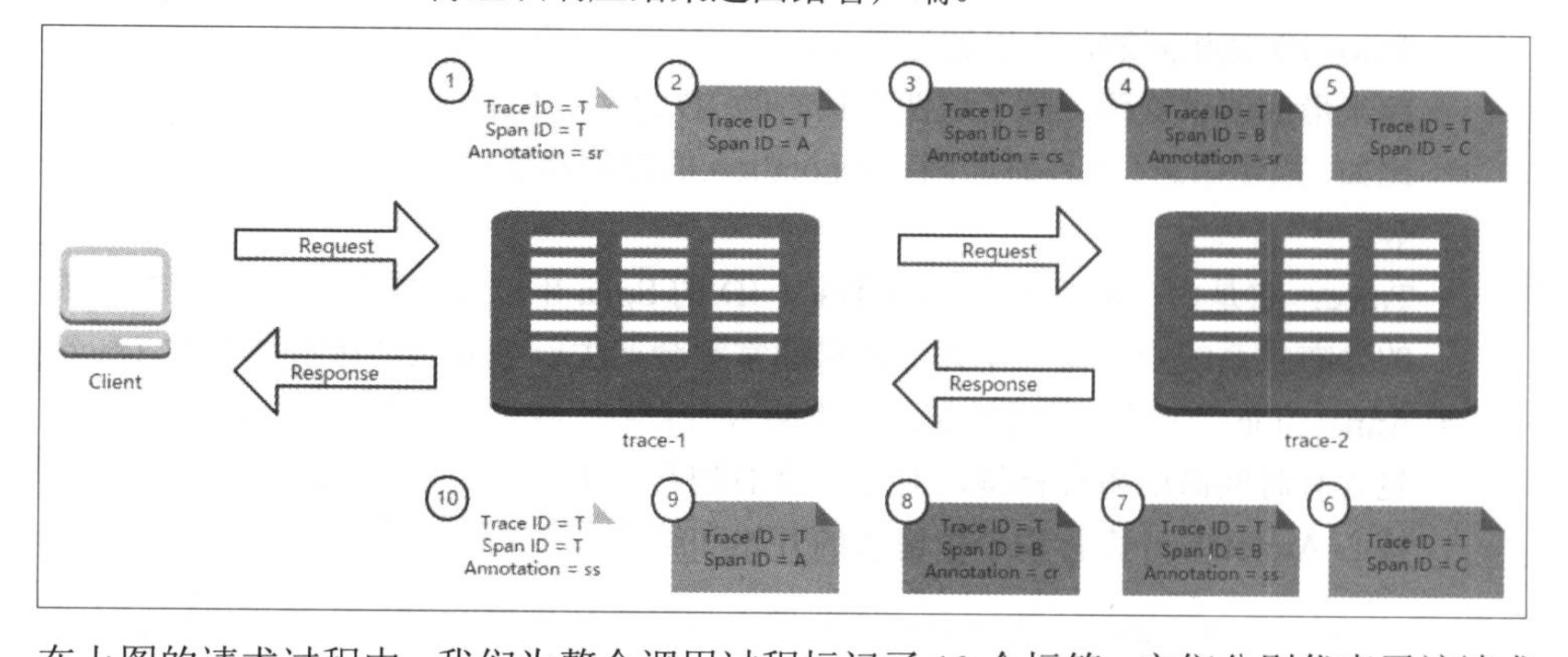
我们先来看看 Zipkin 中关于跟踪信息的一些基础概念。由于 Zipkin 的实现借鉴了 Google 的 Dapper，所以它们有着类似的核心术语，主要有下面几项内容。

- Span：它代表了一个基础的工作单元。我们以 HTTP 请求为例，一次完整的请求过程在客户端和服务端都会产生多个不同的事件状态（比如下面所说的 4 个核心 Annotation 所标识的不同阶段）。对于同一个请求来说，它们属于一个工作单元，所以同一 HTTP 请求过程中的 4 个 Annotation 同属于一个 Span。每一个不同的工作单元都通过一个 64 位的 ID 来唯一标识，称为 Span ID。另外，在工作单元中还存储了一个用来串联其他工作单元的 ID，它也通过一个 64 位的 ID 来唯一标识，称为 Trace ID。在同一条请求链路中的不同工作单元都会有不同的 Span ID，但是它们的 Trace ID 是相同的，所以通过 Trace ID 可以将一次请求中依赖的所有依赖请求串联起来形成请求链路。除了这两个核心的 ID 之外，Span 中还存储了一些其他信息，比如，描述信息、事件时间戳、Annotation 的键值对属性、上一级工作单元的 Span ID 等。
- Trace：它是由一系列具有相同 Trace ID 的 Span 串联形成的一个树状结构。在复杂的分布式系统中，每一个外部请求通常都会产生一个复杂的树状结构的 Trace。
- Annotation：它用来及时地记录一个事件的存在。我们可以把 Annotation 理解为一个包含有时间戳的事件标签，对于一个 HTTP 请求来说，在 Sleuth 中定义了下面 4 个核心 Annotation 来标识一个请求的开始和结束。
  - cs（Client Send）：该 Annotation 用来记录客户端发起了一个请求，同时它也标识了这个 HTTP 请求的开始。
  - sr（Server Received）：该 Annotation 用来记录服务端接收到了请求，并准备开始处理它。通过计算 sr 与 cs 两个 Annotation 的时间戳之差，我们可以得到当前 HTTP 请求的网络延迟。
  - ss（Server Send）：该 Annotation 用来记录服务端处理完请求后准备发送请求响应信息。通过计算 ss 与 sr 两个 Annotation 的时间戳之差，我们可以得到当前服务端处理请求的时间消耗。
  - cr（Client Received）：该 Annotation 用来记录客户端接收到服务端的回复，同时它也标识了这个 HTTP 请求的结束。通过计算 cr 与 cs 两个 Annotation 的时间戳之差，我们可以得到该 HTTP 请求从客户端发起到接收服务端响应的总时间消耗。
- BinaryAnnotation：它用来对跟踪信息添加一些额外的补充说明，一般以键值对的方式出现。比如，在记录 HTTP 请求接收后执行具体业务逻辑时，此时并没有默认的 Annotation 来标识该事件状态，但是有 BinaryAnnotation 信息对其进行补充。

收集机制

在理解了 Zipkin 的各个基本概念之后，下面我们结合前面章节中实现的例子来详细介绍和理解 Spring Cloud Sleuth 是如何对请求调用链路完成跟踪信息的生产、输出和后续处理的。

首先，我们来看看 Sleuth 在请求调用时是怎样来记录和生成跟踪信息的。下图展示了本节中实现示例的运行全过程：客户端发送了一个 HTTP 请求到 trace-1，trace-1 依赖于 trace-2 的服务，所以 trace-1 再发送一个 HTTP 请求到 trace-2，待 trace-2 返回响应之后，trace-1 再组织响应结果返回给客户端。



在上图的请求过程中，我们为整个调用过程标记了 10 个标签，它们分别代表了该请求链路运行过程中记录的几个重要事件状态。根据事件发生的时间顺序我们为这些标签做了从小到大的编号，1 代表请求的开始、10 代表请求的结束。每个标签中记录了一些上面提到过的核心元素：Trace ID、Span ID 以及 Annotation。由于这些标签都源自一个请求，所以它们的 Trace ID 相同，而标签 1 和标签 10 是起始和结束节点，它们的 Trace ID 与 Span ID 是相同的。

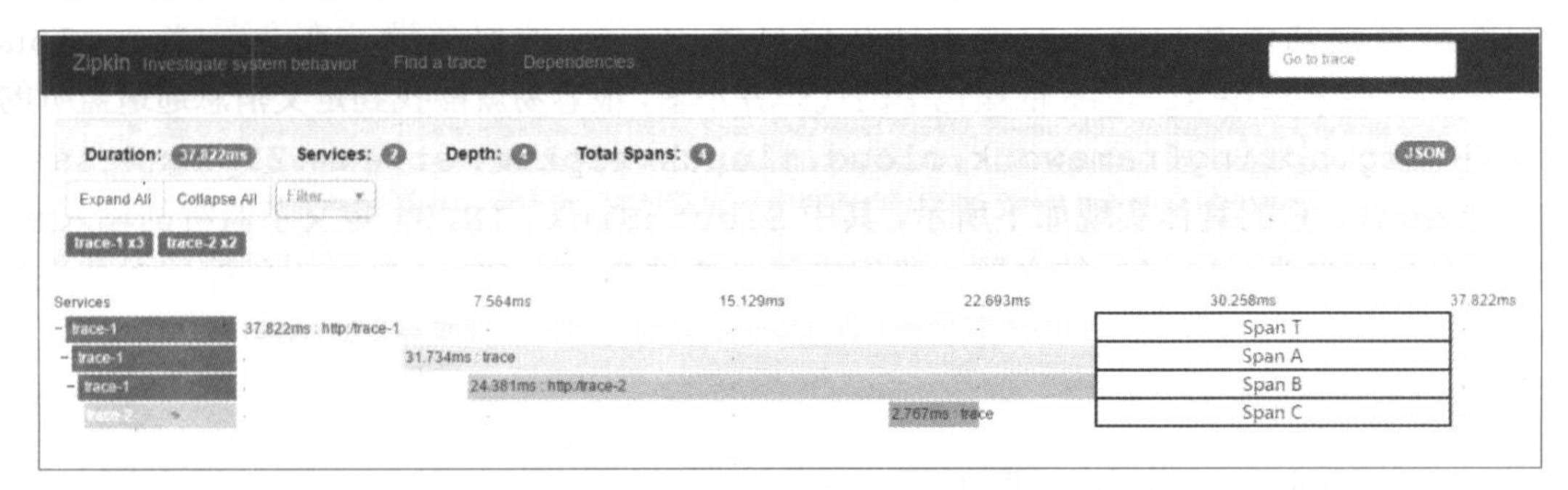
根据 Span ID，我们可以发现在这些标签中一共产生了 4 个不同 ID 的 Span，这 4 个 Span 分别代表了这样 4 个工作单元。

- Span T：记录了客户端请求到达 trace-1 和 trace-1 发送请求响应的两个事件，它可以计算出客户端请求响应过程的总延迟时间。
- Span A：记录了 trace-1 应用在接收到客户端请求之后调用处理方法的开始和结束两个事件，它可以计算出 trace-1 应用用于处理客户端请求时，内部逻辑花费的时间延迟。
- Span B：记录了 trace-1 应用发送请求给 trace-2 应用、trace-2 应用接收请求，trace-2 应用发送响应、trace-1 应用接收响应 4 个事件，它可以计算出 trace-1 调用 trace-2 的总体依赖时间（cr－cs），也可以计算出 trace-1 到

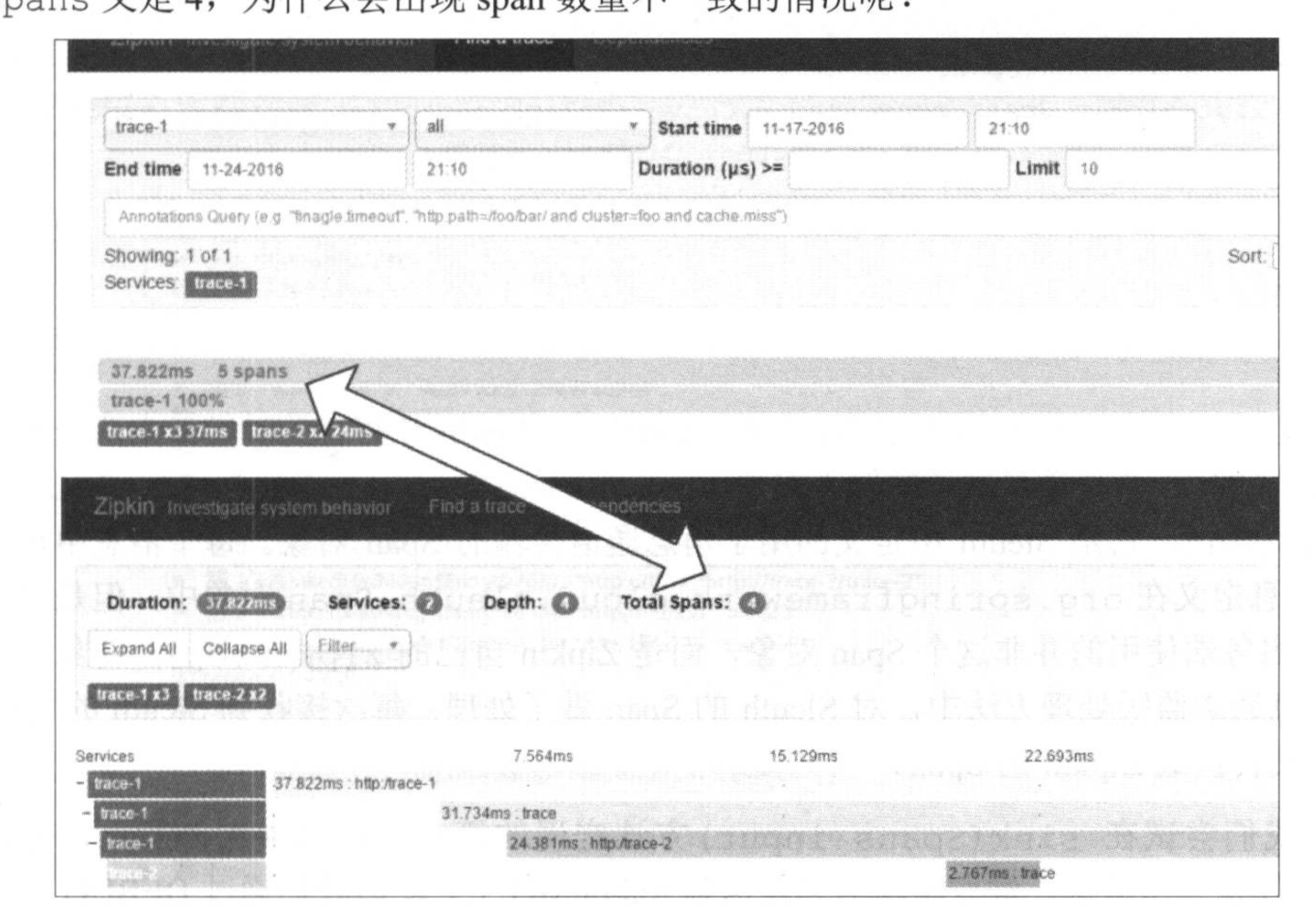
trace-2 的网络延迟（sr－cs），还可以计算出 trace-2 应用用于处理客户端请求的内部逻辑花费的时间延迟（ss－sr）。

- Span C：记录了 trace-2 应用在接收到 trace-1 的请求之后调用处理方法的开始和结束两个事件，它可以计算出 trace-2 应用处理来自 trace-1 的请求时，内部逻辑花费的时间延迟。

在图中展现的这个 4 个 Span 正好对应了 Zipkin 查看跟踪详细页面中的显示内容，它们的对应关系如下图所示。



仔细的读者可能还有这样一个疑惑：我们在 Zipkin 服务端查询跟踪信息时（如下图所示），在查询结果页面中显示的 spans 是 5，而单击进入跟踪明细页面时，显示的 Total Spans 又是 4，为什么会出现 span 数量不一致的情况呢？



实际上这两边的 span 数量内容有不同的含义，查询结果页面中的 5 spans 代表了总共接收的 Span 数量，而在详细页面中的 Total Spans 则是对接收 Span 进行合并后的结果，也就是前文中我们介绍的 4 个不同 ID 的 Span 内容。下面我们来详细研究一下 Zipkin 服务端收集客户端跟踪信息的过程，看看它到底收到了哪些具体的 Span 内容，从而来理解 Zipkin 中收集到的 Span 总数量。

为了更直观地观察 Zipkin 服务端的收集过程，我们可以对之前实现的消息中间件方式收集跟踪信息的程序进行调试。通过在 Zipkin 服务端的消息通道监听程序中增加断点，我们能清楚地知道客户端都发送了什么信息到 Zipkin 的服务端。在 spring-cloud-sleuth-zipkin-stream 依赖包中的代码并不多，很容易就能找到定义消息通道监听的实现类：org.springframework.cloud.sleuth.zipkin. stream.ZipkinMessageListener。它的具体实现如下所示，其中 SleuthSink. INPUT 定义了监听的输入通道，默认会使用名为 sleuth 的主题，我们也可以通过 Spring Cloud Stream 的配置对其进行修改。

```
@MessageEndpoint
@Conditional(NotSleuthStreamClient.class)
public class ZipkinMessageListener {

    final Collector collector;

    @ServiceActivator(inputChannel = SleuthSink.INPUT)
    public void sink(Spans input) {
        List<zipkin.Span> converted = ConvertToZipkinSpanList.convert(input);
        this.collector.accept(converted, Callback.NOOP);
    }

    ...

}
```

从通道监听方法的定义中我们可以看到，Sleuth 与 Zipkin 在整合的时候是由两个不同的 Span 定义的，一个是消息通道的输入对象 org.springframework.cloud.sleuth.stream.Spans，它是 Sleuth 中定义的用于消息通道传输的 Span 对象。每个消息中包含的 Span 信息定义在 org.springframework.cloud.sleuth.Span 对象中，但是真正在 Zipkin 服务端使用的并非这个 Span 对象，而是 Zipkin 自己的 zipkin.Span 对象。所以，在消息通道监听处理方法中，对 Sleuth 的 Span 做了处理，每次接收到 Sleuth 的 Span 之后就将其转换成 Zipkin 的 Span。

下面我们尝试在 sink(Spans input) 方法实现的第一行代码中加入断点，并向 trace-1 发送一个请求，触发跟踪信息发送到 RabbitMQ 上。此时我们通过 DEBUG 模式

可以发现消息通道中都接收到了两次输入，一次来自 trace-1，一次来自 trace-2。下面两张图分别展示了来自 trace-1 和 trace-2 输出的跟踪消息，其中 trace-1 的跟踪消息包含了 3 条 Span 信息，trace-2 的跟踪消息包含了 2 条 Span 信息，所以在这个请求调用链上，一共发送了 5 个 Span 信息，也就是我们在 Zipkin 搜索结果页面中看到的 Spans 的数量信息。

```
▼ input = {Spans@8609}
  ▼ host = {Host@8610}
    ▶ serviceName = "trace-1"
    ▶ address = "192.168.20.65"
    ▶ port = {Integer@8621} "9101"
  ▼ spans = {ArrayList@8611} size = 3
    ▶ 0 = {Span@8616} "[Trace: e9a933ec50d180d6, Span: 1ae2e9a317faa422, Parent: c48122fa096bffe8, exportable:true]"
    ▶ 1 = {Span@8617} "[Trace: e9a933ec50d180d6, Span: c48122fa096bffe8, Parent: e9a933ec50d180d6, exportable:true]"
    ▶ 2 = {Span@8618} "[Trace: e9a933ec50d180d6, Span: e9a933ec50d180d6, Parent: null, exportable:true]"

▼ input = {Spans@8587}
  ▼ host = {Host@8589}
    ▶ serviceName = "trace-2"
    ▶ address = "192.168.20.65"
    ▶ port = {Integer@8593} "9102"
  ▼ spans = {ArrayList@8590} size = 2
    ▶ 0 = {Span@8595} "[Trace: e9a933ec50d180d6, Span: 36194e4182985c4e, Parent: 1ae2e9a317faa422, exportable:true]"
    ▶ 1 = {Span@8596} "[Trace: e9a933ec50d180d6, Span: 1ae2e9a317faa422, Parent: c48122fa096bffe8, exportable:true]"
```

点开一个具体的 Span 内容，我们可以看到如下所示的结构，它记录了 Sleuth 中定义的 Span 详细信息，包括该 Span 的开始时间、结束时间、Span 的名称、Trace ID、Span ID、Tags（对应 Zipkin 中的 BinaryAnnotation）、Logs（对应 Zipkin 中的 Annotation）等之前提到过的核心跟踪信息。

```
▼ spans = {ArrayList@8611} size = 3
  ▼ 0 = {Span@8616} "[Trace: e9a933ec50d180d6, Span: 1ae2e9a317faa422, Parent: c48122fa096bffe8, exportable:true]"
      begin = 1480467275817
      end = 1480467277375
    ▶ name = "http:/trace-2"
      traceId = -1609698301747035946
    ▼ parents = {ArrayList@8627} size = 1
      ▶ 0 = {Long@8670} "-4287106912984760344"
      spanId = 1937367676413977634
      remote = false
      exportable = true
    ▼ tags = {LinkedHashMap@8628} size = 4
      ▶ 0 = {LinkedHashMap$Entry@8636} "http.path" -> "/trace-2"
      ▶ 1 = {LinkedHashMap$Entry@8637} "http.url" -> "http://trace-2/trace-2"
      ▶ 2 = {LinkedHashMap$Entry@8638} "http.method" -> "GET"
      ▶ 3 = {LinkedHashMap$Entry@8639} "http.host" -> "trace-2"
      processId = null
  ▼ logs = {ArrayList@8629} size = 2
      ▶ 0 = {Log@8649} "Log{timestamp=1480467275818, event='cs'}"
      ▶ 1 = {Log@8650} "Log{timestamp=1480467277375, event='cr'}"
      savedSpan = null
    ▶ startNanos = {Long@8630} "46627595152818"
    ▶ durationMicros = {Long@8631} "1558124"
```

介绍到这里仔细的读者可能会有一个疑惑，在明细信息中展示的 Trace ID 和 Span ID 的值为什么与列表展示的概要信息中的 Trace ID 和 Span ID 的值不一样呢？实际上，Trace ID 和 Span ID 都是使用 long 类型存储的，在 DEBUG 模式下查看其明细时自然是 long 类型，也就是它的原始值，但是在查看 Span 对象的时候，我们看到的是通过 `toString()` 函数处理过的值。从 Sleuth 的 Span 源码中我们可以看到如下定义，在输出 Trace ID 和 Span ID 时都调用了 `idToHex` 函数将 long 类型的值转换成了十六进制的字符串值，所以在 DEBUG 时我们会看到两个不一样的值。

```
public String toString() {
    return "[Trace: " + idToHex(this.traceId) + ", Span: " + idToHex(this.spanId)
        + ", Parent: " + getParentIdIfPresent() + ", exportable:" + this.exportable + "]";
}

public static String idToHex(long id) {
    return Long.toHexString(id);
}
```

在接收到 Sleuth 之后，我们将程序继续执行下去，可以看到经过转换后的 Zipkin 的 Span 内容，它们保存在一个名为 `converted` 的列表中，具体内容如下所示：

```
▼ converted = {ArrayList@8672} size = 3
  ▶ 0 = {Span@8687} "{"traceId":"e9a933ec50d180d6","name":"http:/trace-2","id":"1ae2e9a317faa422","parentId":"c48122fa096bffe8","timestamp":1480467275817000,"
  ▶ 1 = {Span@8688} "{"traceId":"e9a933ec50d180d6","name":"trace","id":"c48122fa096bffe8","parentId":"e9a933ec50d180d6","timestamp":1480467275762000,"duratic
  ▶ 2 = {Span@8689} "{"traceId":"e9a933ec50d180d6","name":"http:/trace-1","id":"e9a933ec50d180d6","timestamp":1480467275716000,"duration":1695040,"annotation

▼ converted = {ArrayList@8672} size = 3
  ▼ 0 = {Span@8687} "{"traceId":"e9a933ec50d180d6","name":"http:/trace-2","id":"1ae2e9a317faa422","parentId":"c48122fa096bffe8","timestamp":1480467275817000
      traceId = -1609698301747035946
    ▶ name = "http:/trace-2"
      id = 1937367676413977634
    ▶ parentId = {Long@8670} "-4287106912984760344"
    ▶ timestamp = {Long@8693} "1480467275817000"
    ▶ duration = {Long@8694} "1557000"
    ▼ annotations = {Collections$UnmodifiableRandomAccessList@8695} size = 2
      ▶ 0 = {Annotation@8712} "{"endpoint":{"serviceName":"trace-1","ipv4":"192.168.20.65","port":9101},"timestamp":1480467275818000,"value":"cs"}"
      ▶ 1 = {Annotation@8713} "{"endpoint":{"serviceName":"trace-1","ipv4":"192.168.20.65","port":9101},"timestamp":1480467277375000,"value":"cr"}"
    ▼ binaryAnnotations = {Collections$UnmodifiableRandomAccessList@8696} size = 5
      ▶ 0 = {BinaryAnnotation@8701} "{"key":"http.host","value":"trace-2","endpoint":{"serviceName":"trace-1","ipv4":"192.168.20.65","port":9101}}"
      ▶ 1 = {BinaryAnnotation@8702} "{"key":"http.method","value":"GET","endpoint":{"serviceName":"trace-1","ipv4":"192.168.20.65","port":9101}}"
      ▶ 2 = {BinaryAnnotation@8703} "{"key":"http.path","value":"/trace-2","endpoint":{"serviceName":"trace-1","ipv4":"192.168.20.65","port":9101}}"
      ▶ 3 = {BinaryAnnotation@8704} "{"key":"http.url","value":"http://trace-2/trace-2","endpoint":{"serviceName":"trace-1","ipv4":"192.168.20.65","port":9101}}"
      ▶ 4 = {BinaryAnnotation@8705} "{"key":"sa","value":true,"endpoint":{"serviceName":"trace-1","ipv4":"192.168.20.65","port":9101}}"
      debug = null
```

上图展示了转换后的 Zipkin Span 对象的详细内容，可以看到很多熟悉的名称，也就是之前介绍的关于 Zipkin 中的各个基本概念。而这些基本概念的值我们也都可以在之前 Sleuth 的原始 Span 中找到，其中 Annotations 和 BinaryAnnotations 有一些特殊。在 Sleuth 定义的 Span 中没有使用相同的名称，而是使用了 logs 和 tags 来命名。从这里的详细信息中，我们可以直观地看到 Annotations 和 BinaryAnnotations 的作用，其中 Annotations 中存储了当前 Span 包含的各种事件状态以及对应事件状态的时间戳，而 BinaryAnnotations 则存储了对事件的补充信息，比如上图中存储的就是该 HTTP 请求的细节描述信息，除此之外，它也可

以存储对调用函数的详细描述（如下图所示）。

```
binaryAnnotations = {Collections$UnmodifiableRandomAccessList@8239} size = 3
  0 = {BinaryAnnotation@8272} "{"key":"lc","value":"unknown","endpoint":{"serviceName":"trace-2","ipv4":"192.168.31.189","port":9102}}"
  1 = {BinaryAnnotation@8273} "{"key":"mvc.controller.class","value":"TraceApplication","endpoint":{"serviceName":"trace-2","ipv4":"192.168.31.189","port":9102}}"
  2 = {BinaryAnnotation@8274} "{"key":"mvc.controller.method","value":"trace","endpoint":{"serviceName":"trace-2","ipv4":"192.168.31.189","port":9102}}"
```

下面我们再来详细看看通过调试消息监听程序接收到的这 5 个 Span 内容。首先，可以发现，每个 Span 中都包含有 3 个 ID 信息，其中除了标识 Span 自身的 ID 以及用来标识整条链路的 traceId 之外，还有一个之前没有提过的 parentId，它是用来标识各 Span 父子关系的 ID（它的值来自于上一步执行单元 Span 的 ID）。通过 parentId 的定义我们可以为每个 Span 找到它的前置节点，从而定位每个 Span 在请求调用链中的确切位置。在每条调用链路中都有一个特殊的 Span，它的 parentId 为 null，这类 Span 我们称它为 Root Span，也就是这条请求调用链的根节点。为了弄清楚这些 Span 之间的关系，我们可以从 Root Span 开始来整理出整条链路的 Span 内容。下表展示了我们从 Root Span 开始，根据各个 Span 的父子关系整理出的结果：

Host	Span ID	Parent Span ID	Annotation	BinaryAnnotation
trace-1	e9a933ec50d180d6	null	[sr, ss]	
trace-1	c48122fa096bffe8	e9a933ec50d180d6		[trace-1,TraceApplication,trace]
trace-1	1ae2e9a317faa422	c48122fa096bffe8	[cs, cr]	
trace-2	1ae2e9a317faa422	c48122fa096bffe8	[sr, ss]	
trace-2	36194e4182985c4e	1ae2e9a317faa422		[trace-2,TraceApplication,trace]

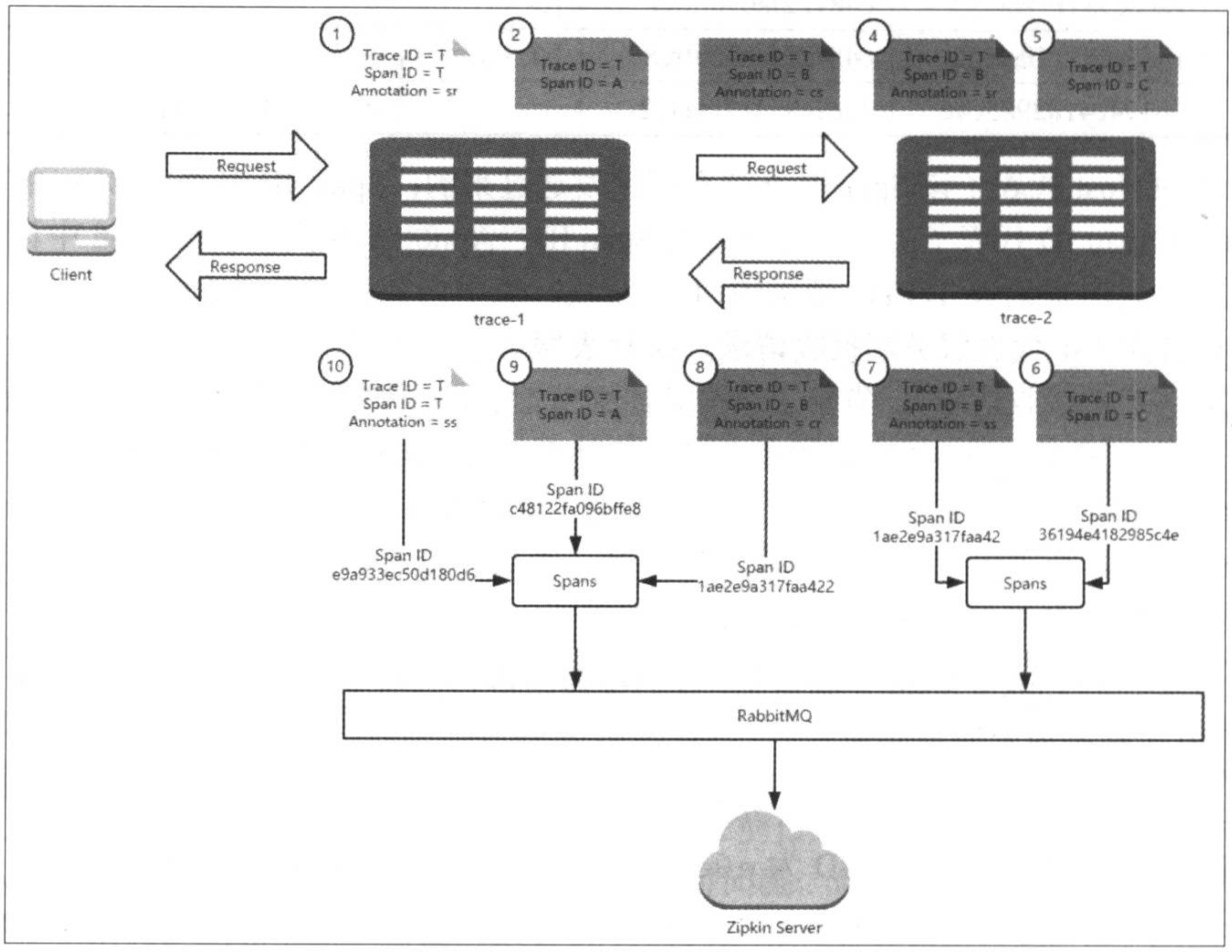
上表中的 Host 代表了该 Span 是从哪个应用发送过来的；Span ID 是当前 Span 的唯一标识；Parent Span ID 代表了上一执行单元的 Span ID；Annotation 代表了该 Span 中记录的事件（这里主要用来记录 HTTP 请求的 4 个阶段，表中内容进行了省略处理，只记录了 Annotation 名称（sr 代表服务端接收请求，ss 代表服务端发送请求，cs 代表客户端发送请求，cr 代表客户端接收请求），省略了一些其他细节信息，比如服务名、时间戳、IP 地址、端口号等信息）；BinaryAnnotation 代表了事件的补充信息（Sleuth 的原始 Span 记录更为详细，Zipkin 的 Span 处理后会去掉一些内容，对于有 Annotation 标识的信息，不再使用 Binary Annotation 补充，在上表中我们只记录了服务名、类名、方法名，同样省略了一些其他信息，比如时间戳、IP 地址、端口号等信息）。

通过收集到的 Zipkin Span 详细信息，我们很容易将它们与本节开始时介绍的一次调用链路中的 10 个标签内容联系起来。

- `Span ID = T` 的标签有 2 个，分别是序号 1 和 10，它们分别表示这次请求的开始和结束。它们对应了上表中 ID 为 `e9a933ec50d180d6` 的 Span，该 Span 的内容在标签 10 执行结束后，由 `trace-1` 将标签 1 和 10 合并成一个 Span 发送给 Zipkin Server。

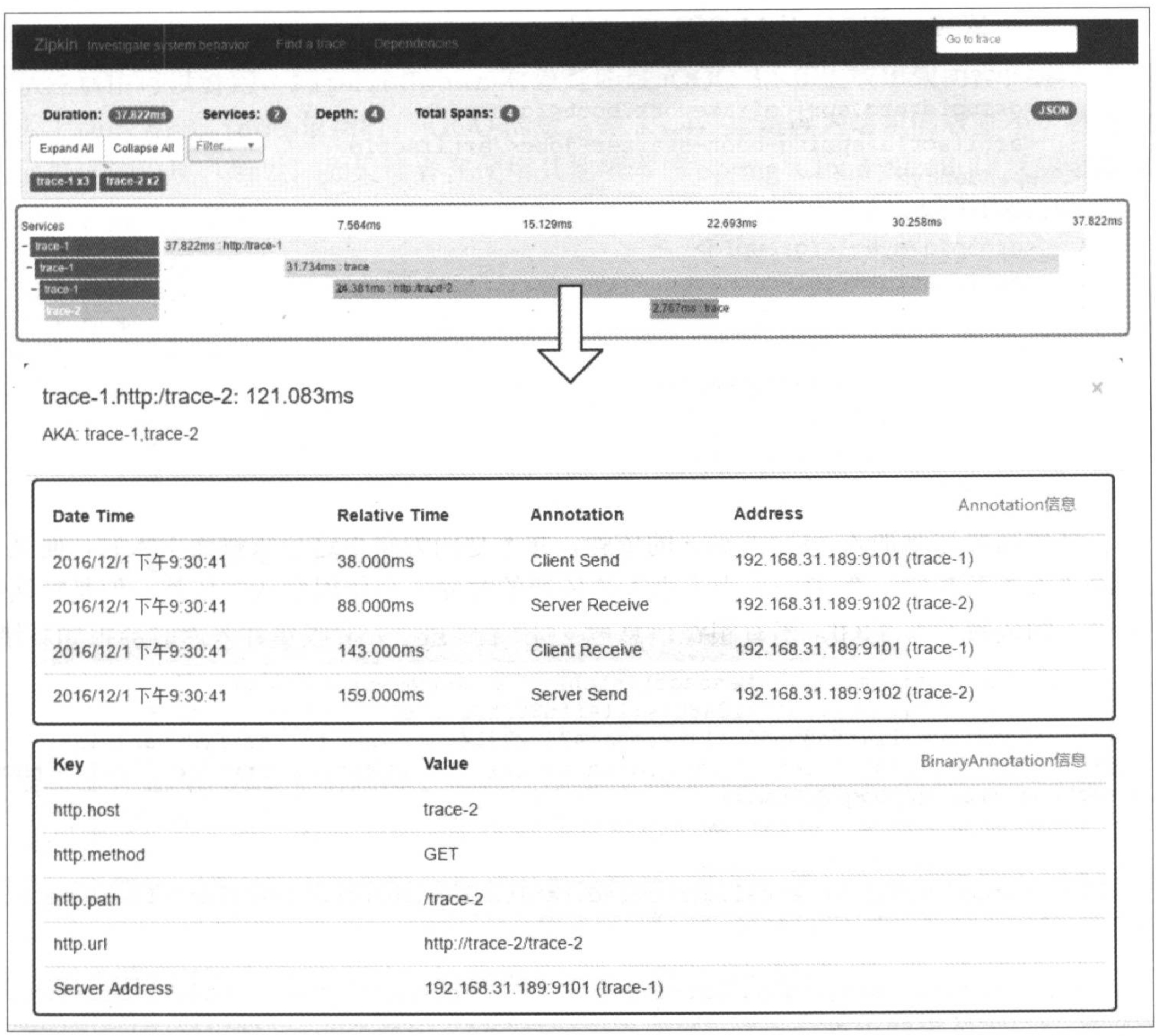
- Span ID = A 的标签有 2 个，分别是序号 2 和 9，它们分别表示了 trace-1 请求接收后，具体处理方法调用的开始和结束。该 Span 的内容在标签 9 执行结束后，由 trace-1 将标签 2 和 9 合并成一个 Span 发送给 Zipkin Server。
- Span ID = B 的标签有 4 个，分别是序号 3、4、7、8，该 Span 比较特殊，它的产生跨越了两个实例，其中标签 3 和 8 是由 trace-1 生成的，而标签 4 和 7 则是由 trace-2 生成的，所以该标签会拆分成两个 Span 内容发送给 Zipkin Server。trace-1 会在标签 8 结束的时候将标签 3 和 8 合并成一个 Span 发送给 Zipkin Server，而 trace-2 会在标签 7 结束的时候将标签 4 和 7 合并成一个 Span 发送给 Zipkin Server。
- Span ID = C 的标签有 2 个，分别是序号 5 和 6，它们分别表示了 trace-2 请求接收后，具体处理方法调用的开始和结束。该 Span 的内容在标签 6 执行结束后，由 trace-2 将标签 2 和 9 合并成一个 Span 发送给 Zipkin Server。

所以，根据上面的分析，Zipkin 总共会收到 5 个 Span：一个 Span T，一个 Span A，两个 Span B，一个 Span C。结合之前请求链路的标签图和这里的 Span 记录，我们可以总结出如下图所示的 Span 收集过程，读者可以参照此图来理解 Span 收集过程的处理逻辑以及各个 Span 之间的关系。



虽然，Zipkin 服务端接收到了 5 个 Span，但就如前文中分析的那样，其中有两个 `Span ID=B` 的标签，由于它们来自于同一个 HTTP 请求（`trace-1` 对 `trace-2` 的服务调用），概念上它们属于同一个工作单元，因此 Zipkin 服务端在前端展现分析详情时会将这两个 Span 合并显示，而合并后的 Span 数量就是在请求链路详情页面中 `Total Spans` 的数量。

下图是本章示例的一个请求链路详情页面，在页面中显示了各个 Span 的延迟统计，其中第三条 Span 信息就是 `trace-1` 对 `trace-2` 的 HTTP 请求调用，通过单击它可以查看该 Span 的详细信息。单击后会以模态框的方式弹出 Span 详细信息（如图下半部分所示），在弹出框中详细展示了 Span 的 Annotation 和 BinaryAnnotation 信息，在 Annotation 区域我们可以看到它同时包含了 `trace-1` 和 `trace-2` 发送的 Span 信息，而在 BinaryAnnotation 区域则展示了该 HTTP 请求的详细信息。



数据存储

默认情况下，Zipkin Server 会将跟踪信息存储在内存中，每次重启 Zipkin Server 都会使之前收集的跟踪信息丢失，并且当有大量跟踪信息时我们的内存存储也会成为瓶颈，所以通常情况下我们都需要将跟踪信息对接到外部存储组件中去，比如使用 MySQL 存储。

Zipkin 的 Storage 组件中默认提供了对 MySQL 的支持，所以我们可以很轻松地为 zipkin-server 增加 MySQL 存储功能。下面我们详细介绍基于消息中间件实现的 zipkin-server 应用，对其进行 MySQL 存储扩展的详细过程。

第一步：为 zipkin-server 添加依赖

为了让 zipkin-server 能够访问 MySQL 数据库，我们需要在它的 pom.xml 文件中增加如下依赖，以支持对 MySQL 的访问：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
    <groupId>org.jooq</groupId>
    <artifactId>jooq</artifactId>
    <version>3.8.0</version>
</dependency>
```

这里需要注意加入 org.jooq 的依赖，并不是因为缺少这个依赖而引入它，而是为了解决该版本存在的一个 Bug。当不引入该依赖的时候，会使用默认的依赖，但是默认情况下引入的依赖并非 3.8.0，当有跟踪信息被 zipkin-server 收集并入库时会报如下错误：

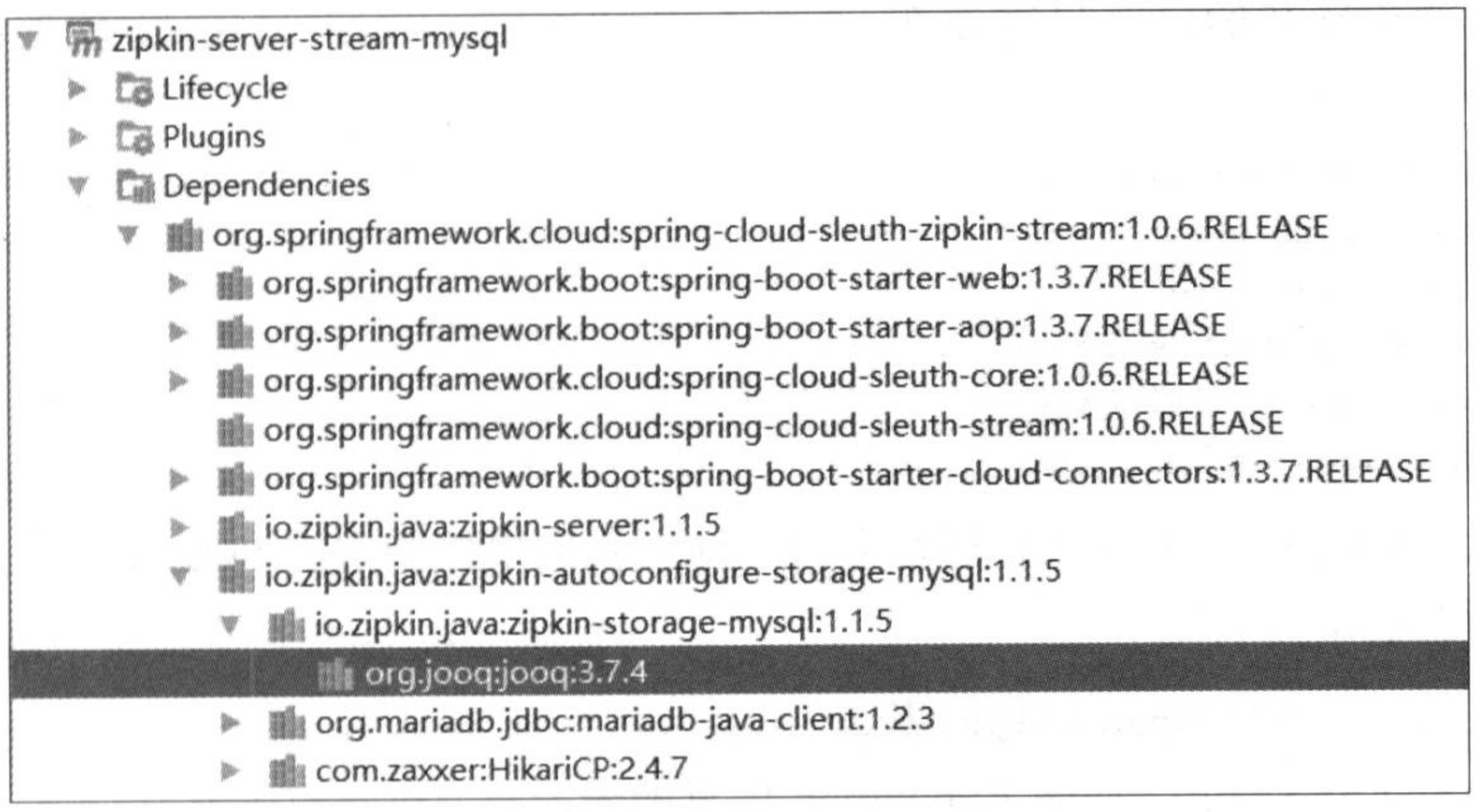
```
    o.s.c.s.z.stream.ZipkinMessageListener   : Cannot store spans
[e7aca79a855ff80a.30671a9177f24801<:114ffd85950b117b,
e7aca79a855ff80a.114ffd85950b117b<:d8ac47672741466c] due to VerifyError(class
zipkin.storage.mysql.internal.generated.tables.ZipkinSpans overrides final method
getSchema.()Lorg/jooq/Schema;)

    java.lang.VerifyError: class
zipkin.storage.mysql.internal.generated.tables.ZipkinSpans overrides final method
getSchema.()Lorg/jooq/Schema;
    ...
    o.s.c.s.i.m.MessagingSpanExtractor       : Deprecated trace headers detected.
Please upgrade Sleuth to 1.1 or start sending headers present in the TraceMessageHeaders
class
    o.s.c.s.z.stream.ZipkinMessageListener   : Cannot store spans
```

```
[8ce053ed8c2410a0.d6b63783a69485d8<:9ebc8131ebdb07fc,
8ce053ed8c2410a0.9ebc8131ebdb07fc<:8ce053ed8c2410a0,
8ce053ed8c2410a0.8ce053ed8c2410a0<:8ce053ed8c2410a0] due to
VerifyError(zipkin/storage/mysql/internal/generated/tables/ZipkinSpans)
```

从报错信息中我们可以看到 VerifyError(zipkin/storage/mysql/internal/generated/tables/ZipkinSpans)这一句，从 ZipkinSpans 的源码中我们可以看到如下内容：

```
@Generated(
    value = {
        "http://www.jooq.org",
        "jOOQ version:3.8.0"
    },
    comments = "This class is generated by jOOQ"
)
```

从注解中可以看到，ZipkinSpans 对象是通过 jOOQ 3.8.0 生成出来的，但是当工程不加入上面的 org.jooq 依赖时，jOOQ 的版本是 3.7.4，正是因为这里的版本不一致导致了上面错误的出现。所以，如果读者在使用其他版本的 Spring Cloud Sleuth 出现类似错误时，可以看看是否是由这个原因引起的。

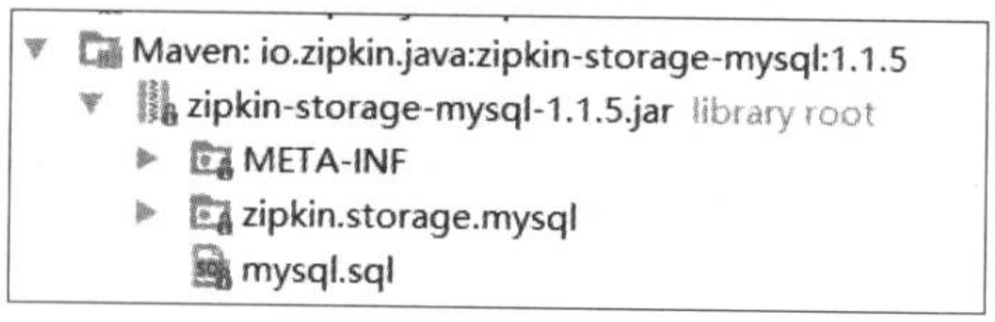


第二步：在 MySQL 中创建用于 Zipkin 存储的 Schema

这里需要注意一点，Zipkin Server 实现关系型数据库存储时，不同版本对数据库表结构都有一些变化。当跨版本使用时很容易出现各种整合问题，所以尽量使用对应版本的脚本来创建 Schema。同时，Zipkin 实现的 MySQL 存储仅在 MySQL 5.6-5.7 版本中测试过，所以尽量使用对应版本的 MySQL，以避免产生不可预知的问题。

本书中我们使用了 Spring Cloud 的 Brixton.SR5 版本，通过查看依赖关系可以知道它使用的 Zipkin 版本为 1.1.5，所以我们可以从 GitHub 上找到该版本并下载创建表结构的脚本。

另外，也可以在本地依赖中找到该脚本，下图展示了 mysql.sql 脚本在本地依赖中的具体位置：



从上图中我们还可以知道，MySQL 的存储支持是通过 zipkin-autoconfigure-storage-mysql 依赖实现的，但是我们之前为什么没有引入该依赖呢？这是由于我们改造的工程基础是消息中间件实现的示例，之前提到过，在该示例中引入的 spring-cloud-sleuth-zipkin-stream 依赖包含了各个常用的依赖组件，其中就包括了对 MySQL 的支持依赖，所以我们这里并不需要再手工添加它。当然，如果不是基于该示例，而是通过 HTTP 实现的收集示例来改造时，就需要自己引入对 zipkin-autoconfigure-storage-mysql 的依赖了。

在获取创建表结构的 SQL 文件之后，我们可以在 MySQL 中手工创建名为 zipkin 的 Schema，并运行 mysql.sql 脚本来创建表结构。除此之外，也可以通过在程序中进行配置的方式让其自动初始化，只需要在 application.properties 中增加如下配置：

```
spring.datasource.schema=classpath:/mysql.sql
spring.datasource.url=jdbc:mysql://localhost:3306/zipkin
spring.datasource.username=root
spring.datasource.password=123456
spring.datasrouce.continueOnError=true
spring.datasrouce.initialize=true
```

通过启动程序，Spring Boot 的 JDBC 模块会自动地为我们根据指定的 SQL 文件来创建表结构。我们可以得到如下两张表。

- zipkin_spans：存储 Span 信息的表。
- zipkin_annotations：存储 Annotation 信息的表。

第三步：切换存储类型

通过第二步，我们已经让 zipkin-server 连接到 MySQL，并且创建了用于存储跟踪信息的 Schema。下面我们只需要再做一个简单配置，让 zipkin-server 的存储切换到 MySQL 即可，具体配置如下：

```
zipkin.storage.type=mysql
```

测试与验证

到这里，我们就已经完成了将 zipkin-server 从内存存储跟踪信息切换为 MySQL 存储跟踪信息的改造。最后，我们继续使用之前的验证方法，通过向 trace-1 的接口发送几个请求 http://localhost:9101/trace-1，当有被抽样收集的跟踪信息时（调试时可以设置 AlwaysSampler 抽样机制来让每个跟踪信息都被收集），查看 MySQL 中的两张表，可以得到类似下面的数据信息。

- zipkin_spans 表

trace_id	id	name	parent_id	debug	start_ts	duration
8972329404569714412	-3039763029959719084	trace	7221594919888949682	(NULL)	1480759142774000	5239
8972329404569714412	1210319574519141610	trace	8972329404569714412	(NULL)	1480759142759000	23632
8972329404569714412	7221594919888949682	http:/trace-2	1210319574519141610	(NULL)	1480759142767000	13538
8972329404569714412	8972329404569714412	http:/trace-1	(NULL)	(NULL)	1480759142751000	32404

- zipkin_annotations 表

trace_id	span_id	a_key	a_value		a_type	a_timestamp	endpoint_ipv4	endpoint_port	endpoint_service_name
8972329404569714412	7221594919888949682	cs	(NULL)	0K	-1	1480759142761000	-1062731674	9101	trace-1
8972329404569714412	7221594919888949682	cr	(NULL)	0K	-1	1480759142779000	-1062731674	9101	trace-1
8972329404569714412	7221594919888949682	http.host	trace-2	7B	6	1480759142761000	-1062731674	9101	trace-1
8972329404569714412	7221594919888949682	http.method	GET	3B	6	1480759142761000	-1062731674	9101	trace-1
8972329404569714412	7221594919888949682	http.path	/trace-2	8B	6	1480759142761000	-1062731674	9101	trace-1
8972329404569714412	7221594919888949682	http.url	http://trace-2...	22B	6	1480759142761000	-1062731674	9101	trace-1
8972329404569714412	7221594919888949682	sa	□	1B	0	1480759142761000	-1062731674	9101	trace-1
8972329404569714412	1210319574519141610	lc	unknown	7B	6	1480759142759000	-1062731674	9101	trace-1
8972329404569714412	1210319574519141610	mvc.controller.class	TraceApplication	16B	6	1480759142759000	-1062731674	9101	trace-1
8972329404569714412	1210319574519141610	mvc.controller.method	trace	5B	6	1480759142759000	-1062731674	9101	trace-1
8972329404569714412	8972329404569714412	sr	(NULL)	0K	-1	1480759142751000	-1062731674	9101	trace-1
8972329404569714412	8972329404569714412	ss	(NULL)	0K	-1	1480759142783000	-1062731674	9101	trace-1
8972329404569714412	-3039763029959719084	lc	unknown	7B	6	1480759142774000	-1062731674	9102	trace-2
8972329404569714412	-3039763029959719084	mvc.controller.class	TraceApplication	16B	6	1480759142774000	-1062731674	9102	trace-2
8972329404569714412	-3039763029959719084	mvc.controller.method	trace	5B	6	1480759142774000	-1062731674	9102	trace-2
8972329404569714412	7221594919888949682	sr	(NULL)	0K	-1	1480759142767000	-1062731674	9102	trace-2
8972329404569714412	7221594919888949682	ss	(NULL)	0K	-1	1480759142780000	-1062731674	9102	trace-2

表中所存储的信息我们已经非常熟悉，之前分析的内容都可以在这两张表中体现出来。比如 Span 的数量问题，从 zipkin_spans 表中，我们可以看到一次请求调用链路的跟踪信息产生了 4 条 span 数据，也就是说，在入 zipkin_spans 表的时候，已经对收集的 span 信息进行了合并，所以在查询详细信息时，不需要每次都来计算合并 span。而在 zipkin_annotations 表中，通过 span_id 字段可以关联到每个具体工作单元的详细信息，同时根据 endpoint_service_name 和 span_id 字段还可以计算出一次请求调用链路中总共接收到的 span 数量。

Zipkin 在存储方面除了对 MySQL 有扩展组件之外，还实现了对 Cassandra 和 ElasticSearch 的支持扩展。具体的整合方式与 MySQL 的整合类似，读者可自行查阅 Zipkin 的官方文档做进一步的了解。

API 接口

Zipkin 不仅提供了 UI 模块让用户可以使用 Web 页面来方便地查看跟踪信息，它还提供了丰富的 RESTful API 接口供用户在第三方系统中调用来定制自己的跟踪信息展示或监控。我们可以在 Zipkin Server 启动时的控制台或日志中找到 Zipkin 服务端提供的 RESTful API 定义，比如下面的日志片段：

```
    s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/api/v1/dependencies],methods=[GET],produces=[application/json]}" ...
    s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/api/v1/trace/{traceId}],methods=[GET],produces=[application/json]}" ...
    s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/api/v1/traces],methods=[GET],produces=[application/json]}" ...
    s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/api/v1/services],methods=[GET]}" ...
    s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/api/v1/spans],methods=[GET]}" ...
    s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/api/v1/spans],methods=[POST]}" ...
    s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped
"{[/api/v1/spans],methods=[POST],consumes=[application/x-thrift]}" ...
```

可以看到 Zipkin Server 提供的 API 接口都以`/api/v1`路径作为前缀，它们的具体功能整理如下：

接口路径	请求方式	接口描述
`/dependencies`	GET	用来获取通过收集到的 Span 分析出的依赖关系
`/services`	GET	用来获取服务列表
`/spans`	GET	根据服务名来获取所有的 Span 名
`/spans`	POST	向 Zipkin Server 上传 Span
`/trace/{traceId}`	GET	根据 Trace ID 获取指定跟踪信息的 Span 列表
`/traces/`	GET	根据指定条件查询并返回符合条件的 trace 清单

更多关于接口的请求参数和请求返回格式等细节说明，可以通过访问 Zipkin 官方的 API 页面 `http://zipkin.io/zipkin-api/`来查看，帮助我们根据自身系统架构来访问 Zipkin Server 以定制自己的 Dashboard 或监控系统。实际上，Zipkin 的 UI 模块也是基于 RESTful API 接口来实现的，有兴趣的读者可以通过浏览器的开发者模式来查看每个页面发起的请求，以此作为调用样例来参考。

附录 A

Starter POMs

下面整理了针对 Spring Boot 1.3.7 版本的 Starter POMs 模块。

模块名	功能描述
spring-boot-starter	Spring Boot 核心模块，包括自动配置支持、日志、YAML
spring-boot-starter-actuator	监控模块，可以帮助我们监控和管理应用
spring-boot-starter-amqp	通过 spring-rabbit 支持 Advanced Message Queuing Protocol
spring-boot-starter-aop	对 AOP 面向切面编程的支持，包括 spring-aop 和 AspectJ
spring-boot-starter-artemis	通过 Apache Artemis 来支持 Java Message Service API
spring-boot-starter-batch	对 Spring Batch 的支持，包含 HSQLDB 数据库
spring-boot-starter-cache	对 Spring Cache 的抽象支持
spring-boot-starter-cloud-connectors	对 Spring Cloud Connectors 的支持，可以通过它简单地连接到云平台的服务，比如 Cloud Foundry 和 Heroku
spring-boot-starter-data-elasticsearch	对 ElasticSearch 搜索和分析引擎的支持，包括 spring-data-elasticsearch
spring-boot-starter-data-gemfire	对 GemFire 分布式存储的支持，包含 spring-data-gemfire
spring-boot-starter-data-jpa	对 Java Persistence API 的支持，包括 spring-data-jpa、spring-orm 和 Hibernate
spring-boot-starter-data-mongodb	对 MongoDB NoSQL 数据库的支持，包括 spring-data-mongodb
spring-boot-starter-data-rest	通过 spring-data-rest-webmvc 支持以 REST 形式暴露 Spring Data 的 repositories
spring-boot-starter-data-solr	对 Apache Solr 搜索平台的支持，包括 spring-data-solr
spring-boot-starter-freemarker	对 FreeMarker 模板引擎的支持
spring-boot-starter-groovy-templates	对 Groovy 模板引擎的支持
spring-boot-starter-hateoas	通过 spring-hateoas 支持基于 HATEOAS 的 RESTful 服务
spring-boot-starter-hornetq	通过 HornetQ 支持 Java Message Service API

续表

模块名	功能描述
spring-boot-starter-integration	对 spring-integration 模块的支持
spring-boot-starter-jdbc	对 JDBC 数据库连接的支持
spring-boot-starter-jersey	支持 Jersey 的 RESTful Web Services 框架
spring-boot-starter-jta-atomikos	通过 Atomikos 实现 JTA 分布式事务
spring-boot-starter-jta-bitronix	通过 Bitronix 实现 JTA 分布式事务
spring-boot-starter-mail	对 javax.mail 的支持
spring-boot-starter-mobile	对 spring-mobile 的支持
spring-boot-starter-mustache	对 Mustache 模板引擎的支持
spring-boot-starter-redis	对 Redis 数据库的支持，包含 spring-redis
spring-boot-starter-security	对 spring-security 的支持
spring-boot-starter-social-facebook	对 spring-social-facebook 的支持
spring-boot-starter-social-linkedin	对 spring-social-linkedin 的支持
spring-boot-starter-social-twitter	对 spring-social-twitter 的支持
spring-boot-starter-test	支持常用的测试框架依赖模块，包括 JUnit、Hamcrest 和 Mockito
spring-boot-starter-thymeleaf	对 thymeleaf 模板引擎的支持，包含与 Spring 的整合
spring-boot-starter-velocity	对 Velocity 模板引擎的支持
spring-boot-starter-web	对全栈 Web 开发的支持，包含 Tomcat 和 spring-webmvc
spring-boot-starter-websocket	对 WebSocket 开发的支持
spring-boot-starter-ws	对 Spring Web Service 的支持

后记

2017 年 1 月 8 日，总算赶在过年前基本完成了本书的编写。自博客上发表第一篇关于 Spring Cloud 的文章开始，持续地研究、实践与输出相关内容已经有半年之多。而编写本书是从 2016 年 9 月开始的，本以为采用当时最新的 Release 版本可以让本书内容紧跟潮流，然而 Spring Cloud 的发展远比我所想象的要快得多。在这 4 个月的时间中，Brixton 版本从 SR5 升级到了 SR7，而 Camden 版本也开始进入 Release 阶段，并持续升级到了 SR3。惊叹 Spring Cloud 高速发展的同时，也让我坚定了对 Spring Cloud 的支持。相信在未来的几年内，在企业应用架构的选择上，Spring Cloud 一定会开始崭露头角。

虽然本书内容是基于 Brixton SR5 版本实现的，但经调试均可以兼容目前最新的 Brixton SR7 和 Camden SR3 版本。不过，对于部分写法在 Camden 中被标注了废弃状态，比如 Spring Cloud Stream 中的一些注解等。对于 Spring Cloud 的更新内容，相信也是大部分读者希望可以持续获得的信息，由于笔者自身也在实践这部分内容，因此在完成本书之后，我也会持续关注 Spring Cloud 的发展，并继续开始更新博客内容，主要关注于 Spring Cloud 的新特性以及实践中碰到的一些问题。读者可以关注我的博客：http://blog.didispace.com，或是我的微信公众号：didispace，以获得最新的分享信息。

在去年创建的 Spring Cloud 中文社区论坛（http://bbs.springcloud.com.cn），目前在许进的牵头和帮助下，已经与一些其他优秀资源做了整合，目前归入 Spring Cloud 中国社区（http://springcloud.cn）统一管理。由于 2016 年大家都忙于各自的工作，还没有对社区的运作（包括文档贡献和知识分享等）做出一定的规划。在 2017 年，我们会抽出时间来规划一些对 Spring Cloud 知识分享、文档贡献等相关的公益性活动，以帮助所有 Spring Cloud 爱好者与实践者共同成长。

下面是关于本书的代码案例资源，主要维护于 GitHub 和开源中国上。如果读者在阅读过程中对文章内容或示例有任何疑问均可以在这里向我提问。

- GitHub：https://github.com/dyc87112/SpringCloudBook
- 开源中国：http://git.oschina.net/didispace/SpringCloudBook

在本书的最后，我要特别感谢我的家人在这段时间内给予的支持，让我能够挤出更多的时间来完成本书的编写。同时，也要感谢 Spring Cloud 中国社区的朋友、关注我博客的读者，以及交流群中给我提出过不少优秀建议的网友，正是因为有了大家的支持，才能让我一直坚持写完本书。即便如此，本书还是无法覆盖 Spring Cloud 的全部，但是后续我会通过博客等其他渠道紧跟 Spring Cloud 的发展，分享更多实战经验和研究成果。

Broadview®
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

Spring Cloud
微服务实战

这本书让我对 Spring Cloud 各组件的使用提升了一个层次，也让我对其实现原理有了初步认识，建议所有打算将 Spring Cloud 用于生产的朋友好好读一下。

——王天青 DaoCloud 首席架构师

为数不多全面讲解 Spring Cloud 微服务的中文书，囊括实施微服务需要深入了解的各个轮子，是一本需要仔细研读的精品之作。

——朱清 冰鉴科技信息技术部总监

书如其人，的确是一本好书，特别是基于技术实践的阐述中又不失对“微服务化”理论层面的讲解以及发展演进的说明，我会推荐给更多的人。

——李俊涛 上海驻云科技执行总监

本书从微服务架构概念出发，结合 Spring Cloud 的解决方案，深入浅出地剖析了其在构建微服务架构中所需的各个基础设施和技术要点，适合正在或打算实施微服务的团队。

——南志文 百联全渠道研发总监

Spring Cloud 是没有足够资金投入或者技术储备的技术团队的福音，其一站式解决方案可从容应对业务的快速发展。这本实战书籍给我们带来了无限惊喜。

——覃罗春 德比软件产品开发负责人

Spring Boot/Cloud 生态的出现为行业提供了一站式解决方案。而要想真正解决架构选型和维护的难题，每一位开发者和架构师都应重视本书中介绍的案例和源码分析。

——程超 合众支付资深技术专家

作为 Spring Cloud 中国社区联合发起人和首批 Spring Cloud 实践与布道者，本书作者积累了大量生产实战经验，并将其通过书中案例奉献给实施微服务架构的企业和开发者。

——许进（xujin.org） Spring Cloud 中国社区创始人/中间件高级研发工程师

微服务概念深入人心，而“全家桶” Spring Cloud 几乎可实现所有微服务功能。这本用于快速上手的工作手册，犹如一颗种子，生根发芽后必将长成一棵参天大树。

——吴峻申 上海青客机器人有限公司架构师

上架建议：Java微服务

ISBN 978-7-121-31301-1

9 787121 313011 >

定价：89.00元

策划编辑：张春雨
责任编辑：刘 舫
封面设计：李 玲