Consul简介

Consul是HashiCorp公司推出的开源工具，用于实现分布式系统的服务发现与配置。Consul是分布式的、高可用的、 可横向扩展的。它具备以下特性:

- 服务发现: 通过 DNS 或 HTTP 对依赖的服务的进行发现。
- 健康检测: Consul提供了健康检查的机制，可以用来避免流量被转发到有故障的服务上。
- Key/Value存储: 应用程序可以根据自己的需要使用Consul提供的Key/Value存储。 Consul提供了简单易用的HTTP接口，结合其他工具可以实现动态配置、功能标记、领袖选举等等功能。
- 多数据中心: Consul支持开箱即用的多数据中心. 这意味着用户不需要担心需要建立额外的抽象层让业务扩展到多个区域。

Consul 的基本概念

- Agent
  组成 Consul 集群的每个节点都是一个 agent，agent 可以 server 或 client 的模式启动

- Client
  负责转发所有的 RPC 到 server 节点，本身无状态且轻量级，因此可以部署大量的 client 节点

- Server
  负责的数据存储和数据的复制集，一般 Consul server 由 3-5 个结点组成来达到高可用，server 之间能够进行相互选举。一个 datacenter 推荐至少有一个 Consul server 的集群。

这里存在两个数据中心：DATACENTER1、DATACENTER2。每个数据中心有着 3 到 5 台 server（该数量使得在故障转移和性能之间达到平衡）。

单个数据中心的所有节点都参与 `LAN Gossip` 池，也就是说该池包含了这个数据中心的所有节点。这有几个目的：

- 不需要给客户端配置服务器地址，发现自动完成。
- 检测节点故障的工作不是放在服务器上，而是分布式的。这使得故障检测比心跳方案更具可扩展性。
- 事件广播，以便在诸如领导选举等重要事件发生时通知。

通常，不会在不同的Consul数据中心之间复制数据。当对另一个数据中心中的资源发出请求时，本地Consul服务器会将RPC请求转发给该资源的远程Consul服务器并返回结果。如果远程数据中心不可用，那么这些资源也将不可用，但这不会影响本地数据中心。在某些特殊情况下，可以复制有限的数据子集，例如使用Consul的内置 ACL复制功能，或者像consul-replicate这样的外部工具

consul 端口说明

端口	说明
TCP/8300	8300 端口用于服务器节点。客户端通过该端口 RPC 协议调用服务端节点。服务器节点之间相互调用
TCP/UDP/8301	8301 端口用于单个数据中心所有节点之间的互相通信，即对 LAN 池信息的同步。它使得整个数据中心能够自动发现服务器地址，分布式检测节点故障，事件广播（如领导选举事件）。
TCP/UDP/8302	8302 端口用于单个或多个数据中心之间的服务器节点的信息同步，即对 WAN 池信息的同步。它针对互联网的高延迟进行了优化，能够实现跨数据中心请求。
8500	8500 端口基于 HTTP 协议，用于 API 接口或 WEB UI 访问。
8600	8600 端口作为 DNS 服务器，它使得我们可以通过节点名查询节点信息。

安装Consul

如何安装Consul，可以参见其官方文档。

运行consul

-dev表示开发模式运行，使用-client 参数可指定允许客户端使用什么ip去访问，例如-client 0.0.0.0 表示任意可以使用。

consul agent -dev -client 0.0.0.0

这样就可以在8500端口开启一个代理服务器，可以通过`http://localhost:8500`访问UI。

Spring Cloud Consul

该项目主要是为了通过Spring Boot自动配置的机制整合Consul，并能够绑定到Spring Environment，以及其他Spring组件中。通过几个简单的注解，可以快速启用和配置应用程序中的常见模式，并使用基于Consul的组件构建大型分布式系统。提供的模式包括服务发现，控制总线和配置。智能路由（Zuul）和客户端负载平衡（Ribbon）,通过与Spring Cloud Netflix的集成提供。

服务发现与注册

要开始使用Consul服务发现，得先加入相应的依赖

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

当一个客户端注册到Consul，它会带上一些关于自身情况的元数据，如：主机地址、端口、id、名字以及一些额外标签。默认情况下Consul会每隔10秒，通过一个HTTP接口`/health`来检测节点的健康情况。 如果健康检测失败，那服务实例就会被标记成critical。

如果Consul客户端位于`localhost:8500`以外的位置，则需要配置来定位客户端。例：

application.yml

```
spring:
  application:
    name: service-admin
  cloud:
    consul:
      host: http://10.3.1.15
      port: 8500
```

警告： 如果使用了Spring Cloud Consul Config，那上面的值需要配置到bootstrap.yml而不是application.yml。

从Environment获取的默认服务名称，实例ID和端口分别为${spring.application.name}，Spring上下文ID和${server.port}。

要禁用Consul Discovery Client，您可以设置spring.cloud.consul.discovery.enabled为false。

要禁用服务注册，您可以设置spring.cloud.consul.discovery.register为false。

配置中心

Consul提供了键值对的方式来存放配置信息和其他元数据。 Spring Cloud Consul Config是一个用于替代Spring Cloud服务端和客户端配置的方案。 配置信息会在某个的"bootstrap"阶段，被加载到Spring Environment中。 配置信息默认存放在/config目录下。 基于应用名字以及Spring Cloud Config的配置文件方式来创建多个PropertySource实例。 例如，一个应用名为：“testApp”，有一个“dev”的环境配置，那就会创建下列配置源：

- config/testApp,dev/
- config/testApp/
- config/application,dev/
- config/application/

`config/application`目录会被所有使用consul配置信息的应用所使用。 `config/testApp`目录仅会被服务为“testApp”的实例使用。

配置信息是可以在应用启动的时候读取到的。 发送一个HTTP POST请求到`/refresh`会自动重新加载配置信息。

要开始使用`Consul Configuration`，得先加入相应的依赖

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

这将启用Spring Cloud Consul Config的自动配置。

可以使用以下属性自定义Consul配置：

bootstrap.yml

```
spring:
  application:
    name: service-admin
  cloud:
    consul:
      host: http://10.3.1.15
      port: 8500
  config:
    #快速失败
  fail-fast: false
```

```
#文件格式
 format: yaml
#表示consul上面的KEY值(或者说文件的名字) 默认是data
data-key: data
  #prefix设置配置值的基本文件夹
prefix: config
  #defaultContext设置所有应用程序使用的文件夹名称
default-context:  ${spring.application.name}
```

Config Watch

Consul Config Watch功能可以对某个key的配置信息进行监听。 当前应用的某个配置信息改动后，一个Refresh事件会被发布，并且Config Watch会触发阻塞式的HTTP接口调用来。 这等价于手动调用/refresh接口。

Config Watch延迟可以通过spring.cloud.consul.config.watch.delay进行修改。默认为1000，单位：毫秒。

可以设置spring.cloud.consul.config.watch.enabled=false禁用Config Watch。

Spring Cloud 动态刷新配置

ContextRefresher

ContextRefresher 用于刷新 Spring 上下文，在以下场景会调用其 refresh 方法。

- 请求 `/refresh` Endpoint。
- Spring Cloud Config/Consul Config提供了一个ConfigWatch功能，可以定时轮询配置的状态，如果状态发生变化，则refresh
- 集成 Spring Cloud Bus 后，收到 `RefreshRemoteApplicationEvent` 事件（任意集成 Bus 的应用，请求 `/bus/refresh`Endpoint 后都会将事件推送到整个集群）。

这个方法包含了整个刷新逻辑

```
public synchronized Set<String> refresh() {
   //获取目前系统的配置
 Map<String, Object> before = extract(
            this.context.getEnvironment().getPropertySources());
 //加载最新配置
 addConfigFilesToEnvironment();
 //对比目前系统配置和最新配置，返回修改后的属性
 Set<String> keys = changes(before,
 extract(this.context.getEnvironment().getPropertySources())).keySet();
 //重新绑定上下文中所有使用了 @ConfigurationProperties 注解的 Spring Bean。
 this.context.publishEvent(new EnvironmentChangeEvent(context, keys));
 //刷新@RefreshScope bean
 this.scope.refreshAll();
 return keys;
}
```

首先是第一步 extract，这个方法接收了当前环境中的所有属性源（PropertySource），并将其中的非标准属性源的所有属性汇总到一个 Map 中返回。

这里的标准属性源指的是 StandardEnvironment 和 StandardServletEnvironment，前者会注册系统变量（System Properties）和环境变量（System Environment），

后者会注册 Servlet 环境下的 Servlet Context 和 Servlet Config 的初始参数（Init Params）和 JNDI 的属性。因为这些属性无法改变，所以不进行刷新。

第二步 addConfigFilesToEnvironment 是核心逻辑，通过重新创建一个SpringBoot环境（非WEB），等到SpringBoot环境启动时就相当于重新启动了一个非web版的服务器。

此时config会自动加载到最新的配置。这个过程类似于启动服务器。 等到服务器启动成功后，最后将新加载的属性源替换掉原属性源，至此属性源本身已经完成更新了。

```
SpringApplicationBuilder builder = new SpringApplicationBuilder(Empty.class)
      .bannerMode(Mode.OFF).web(false).environment(environment);
// Just the listeners that affect the environment (e.g. excluding logging
// listener because it has side effects)
builder.application()
      .setListeners(Arrays.asList(new BootstrapApplicationListener(),
 new ConfigFileApplicationListener()));
capture = builder.run();
```

此时属性源虽然已经更新了，但是配置项都已经注入到了对应的 Spring Bean 中，需要重新进行绑定，所以又触发了两个操作：

- 将刷新后发生更改的 Key 收集起来，发送一个 `EnvironmentChangeEvent` 事件。
- 调用 `RefreshScope.refreshAll` 方法。

EnvironmentChangeEvent

在上文中，ContextRefresher 发布了一个 EnvironmentChangeEvent 事件，这个事件主要会触发两个行为：

- 重新绑定上下文中所有使用了 @ConfigurationProperties 注解的 Spring Bean。
- 如果 logging.level.* 配置发生了改变，重新设置日志级别。

这两段逻辑分别可以在 ConfigurationPropertiesRebinder 和 LoggingRebinder 中看到。

ConfigurationPropertiesRebinder监听了EnvironmentChangeEvent事件，调用rebind方法进行配置重新加载

```
public boolean rebind(String name) {
 Object bean = this.applicationContext.getBean(name);
 this.applicationContext.getAutowireCapableBeanFactory().destroyBean(bean);
 this.applicationContext.getAutowireCapableBeanFactory()
            .initializeBean(bean, name);
 return true;
}
```

`initializeBean` 方法其中主要做了三件事：

- `applyBeanPostProcessorsBeforeInitialization`：调用所有 `BeanPostProcessor` 的 `postProcessBeforeInitialization` 方法。
- `invokeInitMethods`：如果 Bean 继承了 `InitializingBean`，执行 `afterPropertiesSet` 方法，如果 Bean 指定了 `init-method` 属性，如果有则调用对应方法
- `applyBeanPostProcessorsAfterInitialization`：调用所有 `BeanPostProcessor` 的 `postProcessAfterInitialization`方法。

之后 ConfigurationPropertiesRebinder 就完成整个重新绑定流程了。

RefreshScope

- 所有 `@RefreshScope` 的 Bean 都是延迟加载的，只有在第一次访问时才会初始化
- 会先清空缓存然后销毁bean，下次访问时会创建一个新的对象

`refreshAll`方法

```
public void refreshAll() {
 super.destroy();
 this.context.publishEvent(new RefreshScopeRefreshedEvent());
}
```

GenericScope类中有一个成员变量 cache，用于缓存所有已经生成的 Bean，在调用 get 方法时尝试从缓存加载，如果没有的话就生成一个新对象放入缓存，并通过 getBean 初始化其对应的 Bean

```
public Object get(String name, ObjectFactory<?> objectFactory) {
 BeanLifecycleWrapper value = this.cache.put(name,
 new BeanLifecycleWrapper(name, objectFactory));
 locks.putIfAbsent(name, new ReentrantReadWriteLock());
 return value.getBean();
}
```

销毁时只需要将整个缓存清空，下次获取对象时自然就可以重新生成新的对象，也就自然绑定了新的属性

```
public void destroy(){
 List<Throwable> errors = new ArrayList<>();
 Collection<BeanLifecycleWrapper> wrappers = this.cache.clear();
 for (BeanLifecycleWrapper wrapper : wrappers) {
        wrapper.destroy();
 }
 this.errors.clear();
}
```

清空缓存后，下次访问对象时就会重新创建新的对象并放入缓存了。

在清空缓存销毁对象后，它还会发出一个 RefreshScopeRefreshedEvent 事件，在某些 Spring Cloud 的组件中会监听这个事件并作出一些反馈

Zuul

Zuul 在收到这个事件后，会将自身的路由设置为 dirty 状态：

```
private static class ZuulRefreshListener
      implements ApplicationListener<ApplicationEvent> {

   @Autowired
 private ZuulHandlerMapping zuulHandlerMapping;

 @Override
 public void onApplicationEvent(ApplicationEvent event) {
      if (event instanceof ContextRefreshedEvent
            || event instanceof RefreshScopeRefreshedEvent
            || event instanceof RoutesRefreshedEvent)
         this.zuulHandlerMapping.setDirty(true);
 }

}
```

并且当路由实现为 RefreshableRouteLocator 时，会尝试刷新路由：

```
public void setDirty(boolean dirty) {
 this.dirty = dirty;
 if (this.routeLocator instanceof RefreshableRouteLocator) {
      ((RefreshableRouteLocator) this.routeLocator).refresh();
  }
}
```

当状态为 dirty 时，Zuul 会在下一次接受请求时重新注册路由，以更新配置：

```
if (this.dirty) {
 registerHandlers();
 this.dirty = false;
}
```

总结

- 通过使用 `ContextRefresher` 可以进行手动的热更新，而不需要依靠 Bus 或是 Endpoint。
- 热更新会对两类 Bean 进行配置刷新，一类是使用了 `@ConfigurationProperties` 的对象，另一类是使用了 `@RefreshScope` 的对象。
- 这两种对象热更新的机制不同，前者在同一个对象中重新绑定了所有属性，后者则是利用了 `RefreshScope` 的缓存和延迟加载机制，生成了新的对象。
- 通过自行监听 `EnvironmentChangeEvent` 事件，也可以获得更改的配置项，以便实现自己的热更新逻辑。