apollo-client 源码解析

1. Apollo简介

> Apollo（阿波罗）是一款可靠的分布式配置管理中心，诞生于携程框架研发部，能够集中化管理应用不同环境、不同集群的配置，配置修改后能够实时推送到应用端，并且具备规范的权限、流程治理等特性，适用于微服务配置管理场景。
>
> 服务端基于Spring Boot和Spring Cloud开发，打包后可以直接运行，不需要额外安装Tomcat等应用容器。
>
> Java客户端不依赖任何框架，能够运行于所有Java运行时环境，同时对Spring/Spring Boot环境也有较好的支持。
>
> 以下为Apollo客户端的实现原理
>
> 1. 客户端和服务端保持了一个长连接，从而能第一时间获得配置更新的推送。（通过Http Long Polling实现）
> 2. 客户端还会定时从Apollo配置中心服务端拉取应用的最新配置。
>   - 这是一个fallback机制，为了防止推送机制失效导致配置不更新
>   - 客户端定时拉取会上报本地版本，所以一般情况下，对于定时拉取的操作，服务端都会返回304 - Not Modified
>   - 定时频率默认为每5分钟拉取一次，客户端也可以通过在运行时指定System Property: `apollo.refreshInterval`来覆盖，单位为分钟。
> 3. 客户端从Apollo配置中心服务端获取到应用的最新配置后，会保存在内存中
> 4. 客户端会把从服务端获取到的配置在本地文件系统缓存一份
>   - 在遇到服务不可用，或网络不通的时候，依然能从本地恢复配置
> 5. 应用程序可以从Apollo客户端获取最新的配置、订阅配置更新通知
>
> https://www.apolloconfig.com/#/zh/README

apollo可以统一管理不同环境，不同集群的配置，并且配置修改后也能够实时生效，客户端能够实时接收到最新的配置，并通知到程序。

本篇文章着重通过于研究Apollo与spring的整合，以及Apollo如何做到配置热更新。为节省篇幅，关于Spring源码部分不做详细解读。

2.源码解析

在Spring-boot项目中，我们可以通过自动装配的方式引入Apollo，打开apollo-client的spring.factories文件

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.ctrip.framework.apollo.spring.boot.ApolloAutoConfiguration
org.springframework.context.ApplicationContextInitializer=\
com.ctrip.framework.apollo.spring.boot.ApolloApplicationContextInitializer
org.springframework.boot.env.EnvironmentPostProcessor=\
com.ctrip.framework.apollo.spring.boot.ApolloApplicationContextInitializer
```

2.0 EnvironmentPostProcessor

ApolloApplicationContextInitializer 中实现了EnvironmentPostProcessor接口，可以在refresh前执行postProcessEnvironment方法，这让我们可以尽快装载apollo的配置(具体实现可以在此打断点查看调用栈)。

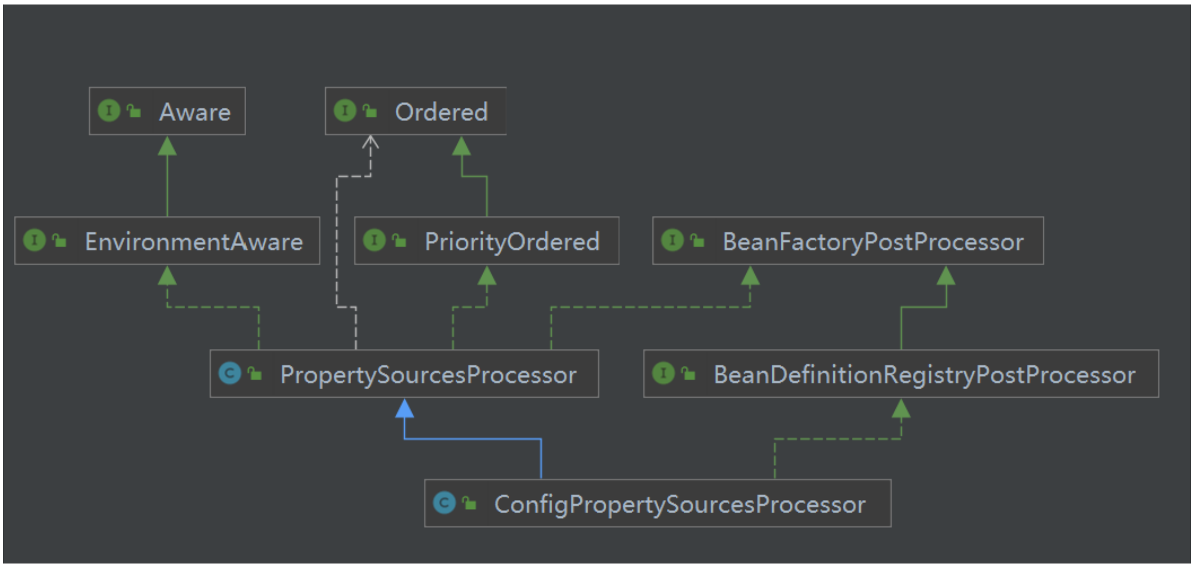
```java
/**
   *
   * In order to load Apollo configurations as early as even before Spring loading logging system phase,
   * this EnvironmentPostProcessor can be called Just After ConfigFileApplicationListener has succeeded.
   *
   * <br />
   * The processing sequence would be like this: <br />
   * Load Bootstrap properties and application properties -----> load Apollo configuration properties ----> Initialize Logging systems
   *
   * @param configurableEnvironment
   * @param springApplication
   */
  @Override
  public void postProcessEnvironment(ConfigurableEnvironment configurableEnvironment, SpringApplication springApplication) {

    // should always initialize system properties like app.id in the first place
    initializeSystemProperty(configurableEnvironment);

    Boolean eagerLoadEnabled = configurableEnvironment.getProperty(PropertySourcesConstants.APOLLO_BOOTSTRAP_EAGER_LOAD_ENABLED, Boolean.class, false);

    //EnvironmentPostProcessor should not be triggered if you don't want Apollo Loading before Logging System Initialization
    if (!eagerLoadEnabled) {
      return;
    }

    Boolean bootstrapEnabled = configurableEnvironment.getProperty(PropertySourcesConstants.APOLLO_BOOTSTRAP_ENABLED, Boolean.class, false);

    if (bootstrapEnabled) {
      initialize(configurableEnvironment);
    }

  }

 /**
   * Initialize Apollo Configurations Just after environment is ready.
   *
   * @param environment
   */
  protected void initialize(ConfigurableEnvironment environment) {
    if
(environment.getPropertySources().contains(PropertySourcesConstants.APOLLO_BOOTS
TRAP_PROPERTY_SOURCE_NAME)) {
      //already initialized
      return;
    }

    String namespaces =
environment.getProperty(PropertySourcesConstants.APOLLO_BOOTSTRAP_NAMESPACES,
ConfigConsts.NAMESPACE_APPLICATION);
    logger.debug("Apollo bootstrap namespaces: {}", namespaces);
    List<String> namespaceList = NAMESPACE_SPLITTER.splitToList(namespaces);

    CompositePropertySource composite = new
CompositePropertySource(PropertySourcesConstants.APOLLO_BOOTSTRAP_PROPERTY_SOURC
E_NAME);
    for (String namespace : namespaceList) {
        //从服务器获取配置
      Config config = ConfigService.getConfig(namespace);


 composite.addPropertySource(configPropertySourceFactory.getConfigPropertySource
(namespace, config));
    }
//将其优先级调至最高
    environment.getPropertySources().addFirst(composite);
  }
```

查看initialize方法，可以看到其通过 ConfigService.getConfig(namespace);从apollo服务器上获取了配置，并将其优先级调为最高，关于如何获取配置，我们后续再讨论。

2.1 refresh

查看ApolloAutoConfiguration配置项

```
@Configuration
@ConditionalOnProperty(PropertySourcesConstants.APOLLO_BOOTSTRAP_ENABLED)
@ConditionalOnMissingBean(PropertySourcesProcessor.class)
public class ApolloAutoConfiguration {

  @Bean
  public ConfigPropertySourcesProcessor configPropertySourcesProcessor() {
    return new ConfigPropertySourcesProcessor();
  }
}

public class ConfigPropertySourcesProcessor extends PropertySourcesProcessor
    implements BeanDefinitionRegistryPostProcessor {

  private ConfigPropertySourcesProcessorHelper helper =
ServiceBootstrap.loadPrimary(ConfigPropertySourcesProcessorHelper.class);

  @Override
  public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry)
throws BeansException {
    helper.postProcessBeanDefinitionRegistry(registry);
  }
}
```



ApolloAutoConfiguration引入了ConfigPropertySourcesProcessor 类。查看该类的继承关系，其中BeanFactoryPostProcssor接口和BeanDefinitionRegistryPostProcessor接口的方法在Spring容器启动的过程中会被调用。查看这两个接口实现的方法。

```
  private ConfigPropertySourcesProcessorHelper helper =
ServiceBootstrap.loadPrimary(ConfigPropertySourcesProcessorHelper.class);

@Override
  public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry)
throws BeansException {
    helper.postProcessBeanDefinitionRegistry(registry);
  }
@Override
  public void postProcessBeanFactory(ConfigurableListableBeanFactory
beanFactory) throws BeansException {
    initializePropertySources();
    initializeAutoUpdatePropertiesFeature(beanFactory);
  }
```

首先看postProcessBeanDefinitionRegistry，这个方法交由ConfigPropertySourcesProcessorHelper去实现，查看ConfigPropertySourcesProcessorHelper相关的代码

```
Map<String, Object> propertySourcesPlaceholderPropertyValues = new HashMap<>();
    // to make sure the default PropertySourcesPlaceholderConfigurer's priority
is higher than PropertyPlaceholderConfigurer
    propertySourcesPlaceholderPropertyValues.put("order", 0);

    BeanRegistrationUtil.registerBeanDefinitionIfNotExists(registry,
PropertySourcesPlaceholderConfigurer.class.getName(),
        PropertySourcesPlaceholderConfigurer.class,
propertySourcesPlaceholderPropertyValues);
```

```
    BeanRegistrationUtil.registerBeanDefinitionIfNotExists(registry,
ApolloAnnotationProcessor.class.getName(),
        ApolloAnnotationProcessor.class);
    BeanRegistrationUtil.registerBeanDefinitionIfNotExists(registry,
SpringValueProcessor.class.getName(),
        SpringValueProcessor.class);
    BeanRegistrationUtil.registerBeanDefinitionIfNotExists(registry,
ApolloJsonValueProcessor.class.getName(),
        ApolloJsonValueProcessor.class);

    processSpringValueDefinition(registry);
```

这部分注册了几个与Apollo实现功能相关的BeanDefinition，接下来我们解析其中比较重要的部分

2.1.1 PropertySourcesPlaceholderConfigurer

Spring 3.1之后通过PropertySourcesPlaceholderConfigurer这个类来实现对配置的解析(如@value这个注解)，具体可跟踪 DefaultListableBeanFactory#doResolveDependency方法。

回到Apollo源码

```
 // to make sure the default PropertySourcesPlaceholderConfigurer's priority is
higher than PropertyPlaceholderConfigurer
    propertySourcesPlaceholderPropertyValues.put("order", 0);

    BeanRegistrationUtil.registerBeanDefinitionIfNotExists(registry,
PropertySourcesPlaceholderConfigurer.class.getName(),
        PropertySourcesPlaceholderConfigurer.class,
propertySourcesPlaceholderPropertyValues);
```

上面的代码注册了PropertySourcesPlaceholderConfigurer的beandefinition，并将其Order设为0，这一步的目的是确保其执行的优先级高于PropertyPlaceholderConfigurer

接着看剩下的部分

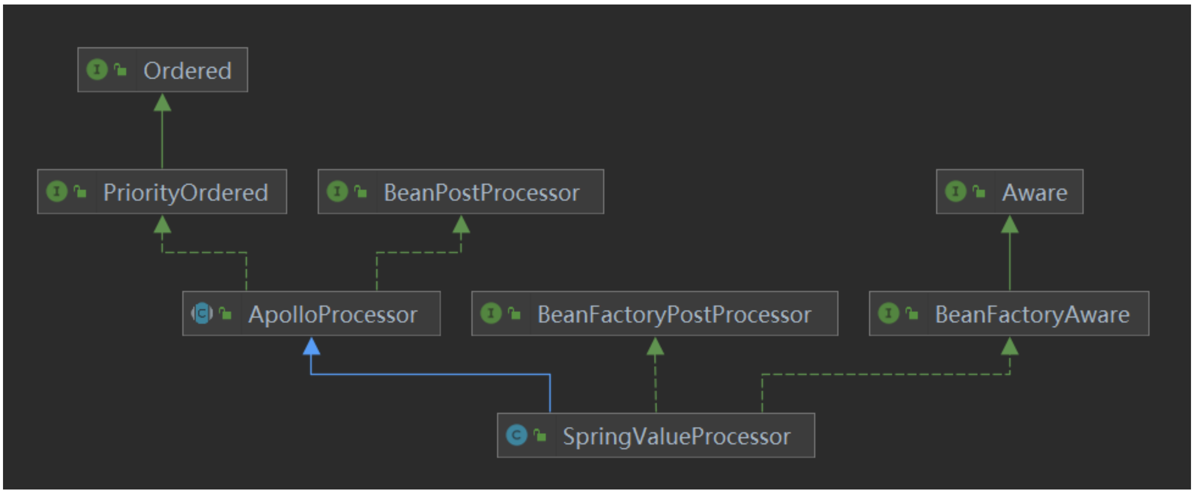
```
    BeanRegistrationUtil.registerBeanDefinitionIfNotExists(registry,
ApolloAnnotationProcessor.class.getName(),
        ApolloAnnotationProcessor.class);
    BeanRegistrationUtil.registerBeanDefinitionIfNotExists(registry,
SpringValueProcessor.class.getName(),
        SpringValueProcessor.class);
    BeanRegistrationUtil.registerBeanDefinitionIfNotExists(registry,
ApolloJsonValueProcessor.class.getName(),
        ApolloJsonValueProcessor.class);
```

2.2.2 ApolloAnnotationProcessor

ApolloAnnotationProcessor这个类创建了处理@ApolloConfig和@ApolloConfigChangeListener这两个注解相关的processor的beandefinition，这两个注解可以帮助我们监听配置的变化，这里我们不做讨论。

2.2.3 SpringValueProcessor

接下来再看SpringValueProcessor

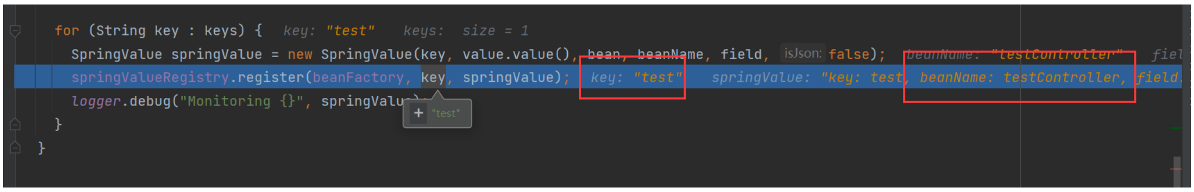


其实现了BeanPostProcessor这个接口，我们查看它实现的方法

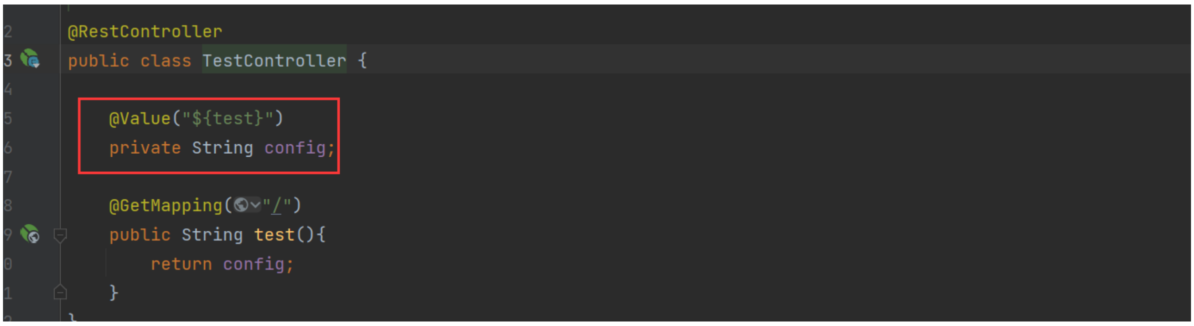
我们把重点放到com.ctrip.framework.apollo.spring.annotation.SpringValueProcessor#processField这个方法

```
 @Override
  public Object postProcessBeforeInitialization(Object bean, String beanName)
      throws BeansException {
    Class clazz = bean.getClass();
    for (Field field : findAllField(clazz)) {
      processField(bean, beanName, field);
    }
    for (Method method : findAllMethod(clazz)) {
      processMethod(bean, beanName, method);
    }
    return bean;
  }

  @Override
  protected void processField(Object bean, String beanName, Field field) {
    // register @Value on field
    Value value = field.getAnnotation(Value.class);
    if (value == null) {
      return;
    }
    Set<String> keys = placeholderHelper.extractPlaceholderKeys(value.value());

    if (keys.isEmpty()) {
      return;
    }

    for (String key : keys) {
      SpringValue springValue = new SpringValue(key, value.value(), bean,
beanName, field, false);
      springValueRegistry.register(beanFactory, key, springValue);
      logger.debug("Monitoring {}", springValue);
    }
  }
```

可以看到，在bean 初始化之后，会调用到processField方法，这个方法会把某些参数注册到springValueRegistry上，我们在这里打个断点查看具体是什么。

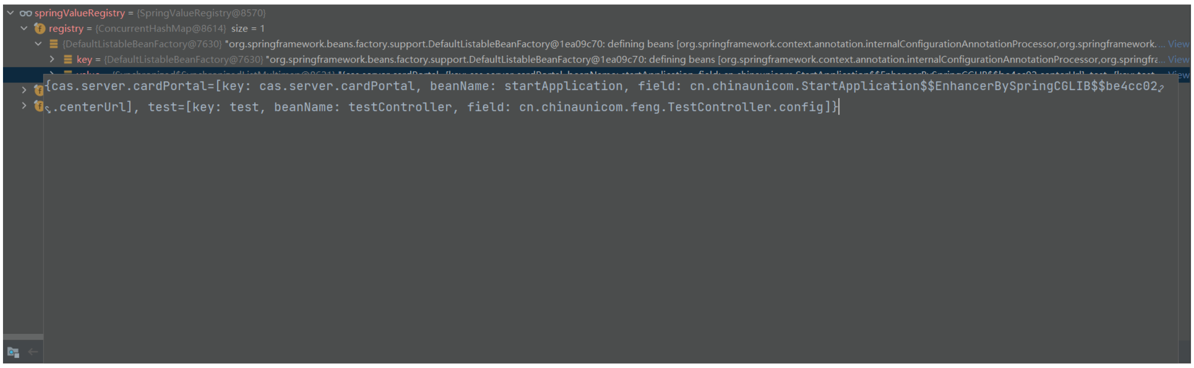


```
  for (String key : keys) {   key: "test"    keys:  size = 1
    SpringValue springValue = new SpringValue(key, value.value(), bean, beanName, field,  isJson: false);   beanName: "testController"   fiel
    springValueRegistry.register(beanFactory, key, springValue);   key: "test"    springValue: "key: test, beanName: testController, field.
    logger.debug("Monitoring {}", springValu
  }
}
```



```
@RestController
public class TestController {

    @Value("${test}")
    private String config;

    @GetMapping("/")
    public String test(){
        return config;
    }
```

从代码中可以看到，这里把我们标注了@Value注解的字段注册到了springValueRegistry中。



```
springValueRegistry = {SpringValueRegistry@8570}
  registry = {ConcurrentHashMap@8614}  size = 1
    {DefaultListableBeanFactory@7630} "org.springframework.beans.factory.support.DefaultListableBeanFactory@1ea09c70: defining beans [org.springframework.context.annotation.internalConfigurationAnnotationProcessor,org.springframework.... View
      key = {DefaultListableBeanFactory@7630} "org.springframework.beans.factory.support.DefaultListableBeanFactory@1ea09c70: defining beans [org.springframework.context.annotation.internalConfigurationAnnotationProcessor,org.springfra... View
{cas.server.cardPortal=[key: cas.server.cardPortal, beanName: startApplication, field: cn.chinaunicom.StartApplication$$EnhancerBySpringCGLIB$$be4cc02
.centerUrl], test=[key: test, beanName: testController, field: cn.chinaunicom.feng.TestController.config]}
```

可以看到，springValueRegistry中维护了配置项key以及配置项所在的类的关系，这让我们可以在配置更新的时候通过反射去更新相关的字段。

2.2.4 PropertySourcesProcessor

回到ConfigPropertySourcesProcessor以及其继承/实现的方法

```
  private ConfigPropertySourcesProcessorHelper helper = 
ServiceBootstrap.loadPrimary(ConfigPropertySourcesProcessorHelper.class);

@Override
  public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry)
throws BeansException {
    helper.postProcessBeanDefinitionRegistry(registry);
  }
@Override
  public void postProcessBeanFactory(ConfigurableListableBeanFactory
beanFactory) throws BeansException {
    initializePropertySources();
    initializeAutoUpdatePropertiesFeature(beanFactory);
  }
```

刚才我们解读了postProcessBeanDefinitionRegistry这个方法，现在看postProcessBeanFactory这个方法

```
@Override
  public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
throws BeansException {
    initializePropertySources();
    initializeAutoUpdatePropertiesFeature(beanFactory);
  }
```

查看第一个方法

```
private void initializePropertySources() {
    if
(environment.getPropertySources().contains(PropertySourcesConstants.APOLLO_PROPE
RTY_SOURCE_NAME)) {
      //already initialized
      return;
    }
    CompositePropertySource composite = new
CompositePropertySource(PropertySourcesConstants.APOLLO_PROPERTY_SOURCE_NAME);

    //sort by order asc
    ImmutableSortedSet<Integer> orders =
ImmutableSortedSet.copyOf(NAMESPACE_NAMES.keySet());
    Iterator<Integer> iterator = orders.iterator();

    while (iterator.hasNext()) {
      int order = iterator.next();
      for (String namespace : NAMESPACE_NAMES.get(order)) {
        Config config = ConfigService.getConfig(namespace);


composite.addPropertySource(configPropertySourceFactory.getConfigPropertySource(
namespace, config));
      }
    }

    // clean up
    NAMESPACE_NAMES.clear();

    // add after the bootstrap property source or to the first
    if (environment.getPropertySources()

.contains(PropertySourcesConstants.APOLLO_BOOTSTRAP_PROPERTY_SOURCE_NAME)) {

      // ensure ApolloBootstrapPropertySources is still the first
      ensureBootstrapPropertyPrecedence(environment);

      environment.getPropertySources()

.addAfter(PropertySourcesConstants.APOLLO_BOOTSTRAP_PROPERTY_SOURCE_NAME,
composite);
    } else {
      environment.getPropertySources().addFirst(composite);
    }
  }
```

这个方法创建了一个Apollo的CompositePropertySource，并确保了其优先级，这使得apollo的配置高于其他的来源，但是我们在spring上下文refresh之前的prepareEnviroment(详情查看2.0)就已经添加过一个apollo配置源，所以这个方法会判断，如果之前添加过apollo配置源，也就是APOLLO_BOOTSTRAP_PROPERTY_SOURCE_NAME对应的配置。则将这次初始化的配置源放到第二位，由此可见代码里确保了Apollo配置的优先级，所以如果项目里的其他配置源配置某个值，apollo配置中心也配置了相同的配置值，spring会以apollo的为准。

再看第二个方法

```
private void
initializeAutoUpdatePropertiesFeature(ConfigurableListableBeanFactory
beanFactory) {
    if (!configUtil.isAutoUpdateInjectedSpringPropertiesEnabled() ||
        !AUTO_UPDATE_INITIALIZED_BEAN_FACTORIES.add(beanFactory)) {
      return;
    }

    AutoUpdateConfigChangeListener autoUpdateConfigChangeListener = new
AutoUpdateConfigChangeListener(
        environment, beanFactory);

    List<ConfigPropertySource> configPropertySources =
configPropertySourceFactory.getAllConfigPropertySources();
    for (ConfigPropertySource configPropertySource : configPropertySources) {
      configPropertySource.addChangeListener(autoUpdateConfigChangeListener);
    }
  }
```

这个方法创建了跟配置自动更新有关的AutoUpdateConfigChangeListener，这个才我们配置更新的时候派上用场。

2.2 获取配置

回到ApolloApplicationContextInitializer类，其中initialize方法中的

```
 Config config = ConfigService.getConfig(namespace);
```

这一行代码是获取配置的逻辑，跟随以下调用链

```
Config config = ConfigService.getConfig(namespace); ->
s_instance.getManager().getConfig(namespace); ->
DefaultConfigManager.getConfig(String namespace) ->
config = factory.create(namespace); ->
DefaultConfigFactory.create ->
DefaultConfig(namespace, createLocalConfigRepository(namespace)); ->
createLocalConfigRepository(namespace) ->
LocalFileConfigRepository(namespace, createRemoteConfigRepository(namespace)); -
>
createRemoteConfigRepository(namespace) ->
RemoteConfigRepository(namespace); ->
```

根据一系列步骤，我们来到了RemoteConfigRepository方法的构造函数

```
public RemoteConfigRepository(String namespace) {
    m_namespace = namespace;
    m_configCache = new AtomicReference<>();
    m_configUtil = ApolloInjector.getInstance(ConfigUtil.class);
    m_httpUtil = ApolloInjector.getInstance(HttpUtil.class);
    m_serviceLocator = ApolloInjector.getInstance(ConfigServiceLocator.class);
    remoteConfigLongPollService =
ApolloInjector.getInstance(RemoteConfigLongPollService.class);
    m_longPollServiceDto = new AtomicReference<>();
    m_remoteMessages = new AtomicReference<>();
    m_loadConfigRateLimiter =
RateLimiter.create(m_configUtil.getLoadConfigQPS());
    m_configNeedForceRefresh = new AtomicBoolean(true);
    m_loadConfigFailSchedulePolicy = new
ExponentialSchedulePolicy(m_configUtil.getOnErrorRetryInterval(),
        m_configUtil.getOnErrorRetryInterval() * 8);
    gson = new Gson();
    this.trySync();
    this.schedulePeriodicRefresh();
    this.scheduleLongPollingRefresh();
  }
```

查看trySync()方法，进入到sync()方法,来到RemoteConfigRepository的实现，进入其loadApolloConfig方法。其中构造了http请求从apollo服务器获取配置，以下摘取部分代码片段,可以见到，apollo-client在这一步中获取了配置。

```
private ApolloConfig loadApolloConfig() {

          HttpRequest request = new HttpRequest(url);

          HttpResponse<ApolloConfig> response = m_httpUtil.doGet(request,
ApolloConfig.class);

    }
```

看完了trySync方法，我们在看接下来的

```
this.schedulePeriodicRefresh();
this.scheduleLongPollingRefresh();
```

这两个方法

```
//这个方法是apollo的一个保底机制，使用一个线程池，每隔一段时间去apollo服务器获取配置信息。
private void schedulePeriodicRefresh() {
    logger.debug("Schedule periodic refresh with interval: {} {}",
        m_configUtil.getRefreshInterval(),
m_configUtil.getRefreshIntervalTimeUnit());
    m_executorService.scheduleAtFixedRate(
        new Runnable() {
          @Override
          public void run() {
            Tracer.logEvent("Apollo.ConfigService",
String.format("periodicRefresh: %s", m_namespace));
            logger.debug("refresh config for namespace: {}", m_namespace);
```

```
      trySync();
      Tracer.logEvent("Apollo.Client.Version", Apollo.VERSION);
     }
    }, m_configUtil.getRefreshInterval(), m_configUtil.getRefreshInterval(),
    m_configUtil.getRefreshIntervalTimeUnit());
 }
```

```
 // 调用栈 scheduleLongPollingRefresh ->
remoteConfigLongPollService.submit(m_namespace, this); -> startLongPolling
// -> m_longPollingService.submit -> stopLongPollingRefresh
private void scheduleLongPollingRefresh() {
  remoteConfigLongPollService.submit(m_namespace, this);
 }
 public boolean submit(String namespace, RemoteConfigRepository
remoteConfigRepository) {
  boolean added = m_longPollNamespaces.put(namespace, remoteConfigRepository);
  m_notifications.putIfAbsent(namespace, INIT_NOTIFICATION_ID);
  if (!m_longPollStarted.get()) {
   startLongPolling();
  }
  return added;
 }
 if (!m_longPollStarted.compareAndSet(false, true)) {
   //already started
   return;
  }
  try {
   final String appId = m_configUtil.getAppId();
   final String cluster = m_configUtil.getCluster();
   final String dataCenter = m_configUtil.getDataCenter();
   final String secret = m_configUtil.getAccessKeySecret();
   final long longPollingInitialDelayInMills =
m_configUtil.getLongPollingInitialDelayInMills();
   m_longPollingService.submit(new Runnable() {
    @Override
    public void run() {
     if (longPollingInitialDelayInMills > 0) {
      try {
       logger.debug("Long polling will start in {} ms.",
longPollingInitialDelayInMills);
       TimeUnit.MILLISECONDS.sleep(longPollingInitialDelayInMills);
      } catch (InterruptedException e) {
       //ignore
      }
     }
     doLongPollingRefresh(appId, cluster, dataCenter, secret);
    }
   });
  } catch (Throwable ex) {
   m_longPollStarted.set(false);
   ApolloConfigException exception =
     new ApolloConfigException("Schedule long polling refresh failed", ex);
   Tracer.logError(exception);
   logger.warn(ExceptionUtil.getDetailMessage(exception));
  }
```

可以看到，里面用了一个线程池提交了一个任务，其中调用了stopLongPollingRefresh方法，下面取出该方法部分片段做展示。

```
void stopLongPollingRefresh() {
    this.m_longPollingStopped.compareAndSet(false, true);
  }

  private void doLongPollingRefresh(String appId, String cluster, String
dataCenter, String secret) {


        url =
            assembleLongPollRefreshUrl(lastServiceDto.getHomepageUrl(), appId,
cluster, dataCenter,
                m_notifications);


        HttpRequest request = new HttpRequest(url);
        request.setReadTimeout(LONG_POLLING_READ_TIMEOUT);

        // 次数是长轮询的调用方式
        final HttpResponse<List<ApolloConfigNotification>> response =
            m_httpUtil.doGet(request, m_responseType);

        if (response.getStatusCode() == 200 && response.getBody() != null) {
          //服务端发现有配置更新，返回200，客户端执行配置更新
        }

        //try to load balance
        if (response.getStatusCode() == 304 && random.nextBoolean()) {
          //服务端如果在一个时间段内没查询到配置更新，返回304，客户端继续下一次长轮询
        }

    }
  }
```

可以看到，apollo使用了一种长轮询的方式来实时获取配置，客户端调用服务端的接口，服务端如果配置没有更新，会hand住请求，等待请求过期或者配置更新后再返回，如果配置有更新，客户端去执行配置更新的方法。除了长轮询之外，还有一种托底的方式，每隔一段时间去服务器获取最新的配置。

apollo配置更新可以查看AutoUpdateConfigChangeListener的onChange方法，可以看到，这里使用到了我们上面提到的springValueRegistry中存储的信息，通过反射的方式去更新配置值。

```
//onChange -> updateSpringValue -> update -> injectField
@Override
  public void onChange(ConfigChangeEvent changeEvent) {
    Set<String> keys = changeEvent.changedKeys();
    if (CollectionUtils.isEmpty(keys)) {
      return;
    }
    for (String key : keys) {
      // 1. check whether the changed key is relevant
      Collection<SpringValue> targetValues =
springValueRegistry.get(beanFactory, key);
```

```
      if (targetValues == null || targetValues.isEmpty()) {
        continue;
      }

      // 2. update the value
      for (SpringValue val : targetValues) {
        updateSpringValue(val);
      }
    }
  }

 private void updateSpringValue(SpringValue springValue) {
    try {
      Object value = resolvePropertyValue(springValue);
      springValue.update(value);

      logger.info("Auto update apollo changed value successfully, new value: {},
{}", value,
          springValue);
    } catch (Throwable ex) {
      logger.error("Auto update apollo changed value failed, {}",
springValue.toString(), ex);
    }
  }

  public void update(Object newVal) throws IllegalAccessException,
InvocationTargetException {
    if (isField()) {
      injectField(newVal);
    } else {
      injectMethod(newVal);
    }
  }

    private void injectField(Object newVal) throws IllegalAccessException {
    Object bean = beanRef.get();
    if (bean == null) {
      return;
    }
    boolean accessible = field.isAccessible();
    field.setAccessible(true);
    field.set(bean, newVal);
    field.setAccessible(accessible);
  }
```