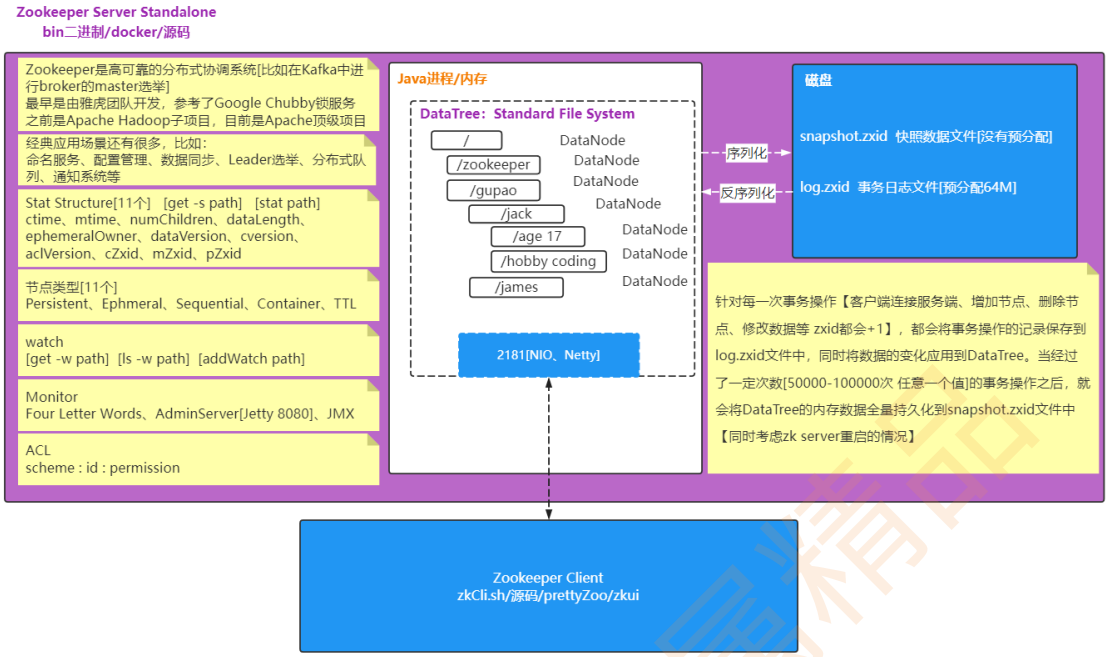
01 前2节课回顾



02 Zookeeper API

2.1 引入Zookeeper依赖

```
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.7.1</version>
</dependency>
```

2.2 日志输出

（1）引入依赖

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.25</version>
</dependency>

<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

（2）定义log4j.properties文件

> resources/log4j.properties

```
###set log levels###
log4j.rootLogger=info, stdout
###output to the console###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=[%d{dd/MM/yy HH:mm:ss:SSS z}] %t
%5p %c{2}: %m%n
```

2.3 连接服务端

```
public class ZkConnUtil {
    private static ZooKeeper zookeeper;
    private static final CountDownLatch countDownLatch = new CountDownLatch(1);

    // 获得zkConn
    public static ZooKeeper getZkConn(String zkServer) throws Exception {
        zookeeper = new ZooKeeper(zkServer, 30000, new Watcher() {
            @Override
            public void process(WatchedEvent event) {
                Event.KeeperState state = event.getState();
                if (Event.KeeperState.SyncConnected == state) {
                    System.out.println("连接zkServer成功.");
                    countDownLatch.countDown();
                }
            }
        });
        countDownLatch.await();
        return zookeeper;
    }

    public static void main(String[] args) throws Exception {
        ZooKeeper zooKeeper = getZkConn("192.168.0.8:2181");
    }
}
```

2.4 创建ZNode

```
public class CreateZNode {

    private ZooKeeper zooKeeper;

    public CreateZNode(ZooKeeper zooKeeper) {
        this.zooKeeper = zooKeeper;
    }

    // 同步创建节点
    public void createZNodeWithSync() throws Exception {
        String znode = zooKeeper.create("/zookeeper-api-sync", "111".getBytes(),
ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
        System.out.println("创建节点成功: "+znode);
    }
```

```
    // 异步创建节点
    public void createZNodeWithAsync(){
        zooKeeper.create("/zookeeper-api-
async","111".getBytes(),ZooDefs.Ids.OPEN_ACL_UNSAFE,CreateMode.PERSISTENT,new
AsyncCallback.StringCallback(){
            @Override
            public void processResult(int rc, String path, Object ctx, String
name) {
                System.out.println("rc: "+rc);
                System.out.println("path: "+path);
                System.out.println("ctx: "+ctx);
                System.out.println("name: "+name);
            }
        },"create-asyn");
    }

    public static void main(String[] args) throws Exception {
        CreateZNode createZNode = new
CreateZNode(ZkConnUtil.getZkConn("192.168.0.8:2181"));
//        createZNode.createZNodeWithSync();
        createZNode.createZNodeWithAsync();
        System.in.read();
    }
}
```

2.5 查询ZNode数据并设置监听

```
public class GetZNodeData {
    private ZooKeeper zooKeeper;

    public GetZNodeData(ZooKeeper zooKeeper) {
        this.zooKeeper = zooKeeper;
    }

    // 同步获取数据
    public void getDataSync(){
        Stat stat = new Stat();
        try {
            byte[] data = zooKeeper.getData("/zookeeper-api-sync", new Watcher()
{
                @Override
                public void process(WatchedEvent event) {
                    // 一旦节点发生变化，则会回调该方法
                    System.out.println("event: "+event);
                }
            }, stat);
            String s = new String(data);
            System.out.println("data: "+s);
            System.out.println("stat: "+stat);
        } catch (Exception e){
            e.printStackTrace();
        }
    }
```

```
    // 异步获取数据
    public void getDataAsync(){
        zooKeeper.getData("/zookeeper-api-async",false, new
AsyncCallback.DataCallback() {
            @Override
            public void processResult(int rc, String path, Object ctx, byte[]
data, Stat stat) {
                System.out.println("rc: "+rc);
                System.out.println("path: "+path);
                System.out.println("ctx: "+ctx);
                System.out.println("data: "+new String(data));
                System.out.println("stat: "+stat);
            }
        },"get-data-async");
    }

    public static void main(String[] args) throws Exception {
        GetZNodeData getZNodeData = new
GetZNodeData(ZkConnUtil.getZkConn("192.168.0.8:2181"));
        getZNodeData.getDataSync();
        getZNodeData.getDataAsync();
        System.in.read();
    }
}
```

2.6 修改ZNode数据

```
public class UpdateZNodeData {
    private ZooKeeper zooKeeper;

    public UpdateZNodeData(ZooKeeper zooKeeper) {
        this.zooKeeper = zooKeeper;
    }

    // 同步修改节点数据
    public void setDataSync() throws Exception {
        // 版本号为-1，表示可以直接修改，不用关心版本号
        zooKeeper.setData("/zookeeper-api-sync", "222".getBytes(), -1);
    }

    // 异步修改节点数据
    public void setDataAsync(){
        zooKeeper.setData("/zookeeper-api-async", "222".getBytes(), -1, new
AsyncCallback.StatCallback() {
            @Override
            public void processResult(int rc, String path, Object ctx, Stat
stat) {
                System.out.println("rc: "+rc);
                System.out.println("path: "+path);
                System.out.println("ctx: "+ctx);
                System.out.println("stat: "+stat);
            }
        },"set-data-async");
    }
```

```
    // 根据版本修改同步节点数据
    public void setDataSyncWithVersion() throws Exception {
        Stat stat = new Stat();
        zooKeeper.getData("/zookeeper-api-sync", false, stat);
        zooKeeper.setData("/zookeeper-api-sync", "555".getBytes(),
stat.getVersion());
    }

    public static void main(String[] args) throws Exception {
        UpdateZNodeData updateZNodeData = new
UpdateZNodeData(ZkConnUtil.getZkConn("192.168.0.8:2181"));
//        updateZNodeData.setDataSync();
//        updateZNodeData.setDataAsync();
        updateZNodeData.setDataSyncWithVersion();
        System.in.read();
    }
}
```

2.7 删除ZNode

```
public class DeleteZNode {
    private ZooKeeper zooKeeper;
    public DeleteZNode(ZooKeeper zooKeeper) {
        this.zooKeeper = zooKeeper;
    }
    // 同步删除节点
    public void deleteZNodeSync() throws Exception {
        zooKeeper.delete("/zookeeper-api-sync",-1);
    }
    // 异步删除节点
    public void deleteZNodeAsync(){
        zooKeeper.delete("/zookeeper-api-async", -1, new
AsyncCallback.VoidCallback() {
            @Override
            public void processResult(int rc, String path, Object ctx) {
                System.out.println("rc: "+rc);
                System.out.println("path: "+path);
                System.out.println("ctx: "+ctx);
            }
        },"delete-znode-async");
    }
    public static void main(String[] args) throws Exception {
        DeleteZNode deleteZNode = new
DeleteZNode(ZkConnUtil.getZkConn("192.168.0.8:2181"));
        deleteZNode.deleteZNodeAsync();
        deleteZNode.deleteZNodeSync();
        System.in.read();
    }
}
```

03 Apache Curator

3.1 Curator发音

3.2 What is Curator

官网：https://curator.apache.org/

Apache Curator is a Java/JVM client library for Apache ZooKeeper, a distributed coordination service. It includes a highlevel API framework and utilities to make using Apache ZooKeeper much easier and more reliable. It also includes recipes for common use cases and extensions such as service discovery and a Java 8 asynchronous DSL.

"Guava is to Java what Curator is to ZooKeeper"
Patrick Hunt, ZooKeeper committer

最初是由Netflix团队开发的，后来捐献给了Apache，目前是Apache的顶级项目。

Curator是对Zookeeper客户端的封装，主要目的就是简化Zookeeper客户端的使用，不需要自己手动处理ConnectionLossException、NodeExistsException等异常，提供了连接重连以及watch永久注册等解决方案。

It adds many features that build on ZooKeeper and handles the complexity of managing connections to the ZooKeeper cluster and retrying operations.

3.3 Architecture

3.4 引入curator依赖

```
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-recipes</artifactId>
    <version>5.2.1</version>
</dependency>
```

3.5 配置日志输出

> 参考2.2配置步骤

3.6 Curator对节点的增删改查

```
public class CuratorApi {
    public static void main(String[] args) {
        String connectStr = "192.168.0.8:2181";
        CuratorFramework curatorFramework = CuratorFrameworkFactory   // fluent
                .builder()
                .connectionTimeoutMs(20000)
                .connectString(connectStr)
                .retryPolicy(new ExponentialBackoffRetry(1000, 3))   // 设置客户端
的重试策略，每隔10秒中重试一次，最多3次
                .build();
        curatorFramework.start();
        try {
            // 创建节点 curator-api
            String znode = curatorFramework
                    .create()
                    .withMode(CreateMode.PERSISTENT)
```

```
                .forPath("/curator-api", "666".getBytes());
            System.out.println("创建节点成功: " + znode);

            // 查询节点 curator-api 数据
            byte[] bytes = curatorFramework.getData().forPath(znode);
            System.out.println("节点curator-api 数据查询成功: " + new
String(bytes));

            // 修改节点 curator-api 数据
            curatorFramework.setData().forPath(znode, "888".getBytes());
            System.out.println("节点curator-api 数据修改成功.");
            // 删除节点 curator-api
            curatorFramework.delete().forPath(znode);
            System.out.println("节点curator-api 已被删除.");
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
}
```

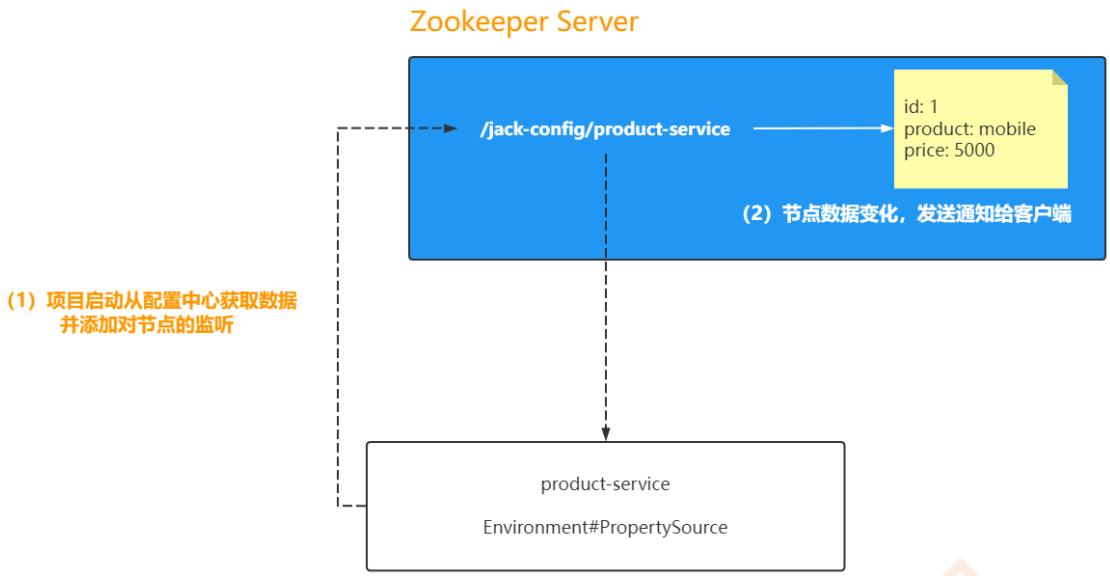
3.7 Curator设置监听

```
public class CuratorWatch {

    public static void main(String[] args) {
        curatorWatchPersistent();
    }

    // 一次性监听
    private static void curatorWatchOnce() {
        String connectStr = "192.168.0.8:2181";
        CuratorFramework curatorFramework = CuratorFrameworkFactory
                .builder()
                .connectionTimeoutMs(20000)
                .connectString(connectStr)
                .retryPolicy(new ExponentialBackoffRetry(1000, 3))
                .build();
        curatorFramework.start();
        try {
            // 创建节点 curator-watch-once
            String znode =
curatorFramework.create().withMode(CreateMode.PERSISTENT).forPath("/curator-
watch-once", "".getBytes());
            System.out.println("节点创建成功: " + znode);

            // 给节点 curator-watch-once 添加一次性watch
            curatorFramework.getData().usingWatcher(new CuratorWatcher() {
                @Override
                public void process(WatchedEvent event) throws Exception {
                    System.out.println("节点发生变化: " + event);
                }
            }).forPath(znode);
            System.out.println("给节点curator-watch-once 添加watch成功.");
```

```
            // 让当前进程不结束
            System.in.read();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    // 永久监听
    private static void curatorWatchPersistent() {
        String connectStr = "192.168.0.8:2181";
        CuratorFramework curatorFramework = CuratorFrameworkFactory
                .builder()
                .connectionTimeoutMs(20000)
                .connectString(connectStr)
                .retryPolicy(new ExponentialBackoffRetry(1000, 3))
                .build();
        curatorFramework.start();
        try {
            // 创建节点 curator-watch-persistent
            String znode = curatorFramework.create().forPath("/curator-watch-
persistent", "".getBytes());
            System.out.println("节点创建成功: " + znode);

            // 永久的监听
            CuratorCache curatorCache = CuratorCache.build(curatorFramework,
znode, CuratorCache.Options.SINGLE_NODE_CACHE);
            CuratorCacheListener listener =
CuratorCacheListener.builder().forAll(new CuratorCacheListener() {
                @Override
                public void event(Type type, ChildData oldData, ChildData data)
{
                    // 等同于Watch#process回调
                    System.out.println("节点 "+data.getPath()+" 发生改变, 事件类型
为: " + type);
                }
            }).build();
            curatorCache.listenable().addListener(listener);
            curatorCache.start();

            System.out.println("给节点curator-watch-persistent 添加watch成功.");
            // 让当前进程不结束
            System.in.read();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

04 基于Zookeeper实现配置中心

4.1 什么是配置中心



4.2 本地配置

（1）创建Spring Boot项目，名称为handwritten-zookeeper-config

（2）application.properties文件

```
age=17
hobby=coding
```

（3）controller/UserController

```
@RestController
@RequestMapping("/user")
public class UserController {


    @Value("${age}")
    private Integer age;

    @Value("${hobby}")
    private String hobby;
    @RequestMapping("/config-value")
    public String config(){
        return "age: "+this.age+", hobby: "+this.hobby;
    }

    @Resource    // 依赖注入Environment对象实例
    private Environment environment;

    @RequestMapping("/conf-env")
    public String confEnv(){
        // this.environment.getProperty("age")其实是通过属性配置源获取到对应的数据内容
        return "age: "+this.environment.getProperty("age")
                +", hobby: "+this.environment.getProperty("hobby");
    }
}
```

（4）访问/user/config-value和/user/config-env

4.3 本地配置实现原理

Externalized Configuration：https://docs.spring.io/spring-boot/docs/2.7.2/reference/htmlsingle/#features.external-config

（1）加载application.properties文件

```
PropertySourceLoader#load()
    ->PropertiesPropertySourceLoader#load()
        ->OriginTrackedMapPropertySource
```

（2）查看Environment的属性配置源

```
SpringApplication#run()
    ->debug: configureIgnoreBeanInfo(environment)
```

（3）比如在Program arguments中配置hello=hi，观察environment中的SimpleCommandLinePropertySource

（4）@Value实现原理

> 推测

```
（1）使用BeanPostProcessor解析类上的@Value字段
（2）获取到字段上的@Value字段
（3）解析@Value字段的value属性值，比如age
（4）从environment中的属性配置源OriginTrackedMapPropertySource中寻找age的key
（5）根据key获取到对应的value值
（6）通过field反射的方式设置value值
```

> 源码验证

```
AutowiredAnnotationBeanPostProcessor#inject()
    ->resolveFieldValue(field, bean, beanName)    # debug  设置条件:
beanName.equals("orderController")
        -
>AutowiredAnnotationBeanPostProcessor.this.beanFactory.resolveDependency(
            ->DefaultListableBeanFactory#resolveDependency
                ->this.doResolveDependency(descriptor...)
                    ->Object value =
getAutowireCandidateResolver().getSuggestedValue(descriptor)     # debug   获取到
@Value属性上的value，比如age
                        -> String strVal = resolveEmbeddedValue((String) value)
  # 根据age从env中寻找与之对应的值
                            -> field#set(bean,value)     #
AutowiredAnnotationBeanPostProcessor#inject最后一段逻辑  通过反射给目标字段赋值
```

4.4 本地配置存在的问题

（1）修改application.properties中的属性值，在不重启项目的情况下不会自动更新

（2）如果有多个微服务项目需要用到该属性值，就只能在各自项目中维护一份，不利于管理

4.5 Spring生态中的扩展机制

4.5.1 常见扩展机制

> 所谓的扩展机制就是不修改Spring生态源码，也能够把一些想要的代码放到启动流程中

```
ApplicationContextInitializer
事件监听机制
BeanPostProcessor
BeanFactoryPostProcessor
ApplicationRunner
...
```

4.5.2 技术选型ApplicationContextInitializer

（1）设置ApplicationContextInitializer

```
// 构造函数
public SpringApplication(ResourceLoader resourceLoader, Class<?>...
primarySources) {
   setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));   # SPI:读取所
有spring.factories文件中的ApplicationContextInitializer类型，并实例化存放到list集合中
}

// 存放到list集合中
private List<ApplicationContextInitializer<?>> initializers;
```

（2）自定义ApplicationContextInitializer

```
public class ZkConfigApplicationContextInitializer implements
ApplicationContextInitializer {
    @Override
    public void initialize(ConfigurableApplicationContext context) {

    }
}
```

> resources/META-INF/spring.factories

```
# Initializers
org.springframework.context.ApplicationContextInitializer=\
com.jack.handwrittenzookeeperconfig.initializer.ZkConfigApplicationContextInitia
lizer
```

> 再次查看initializers数量

```
initializers = {ArrayList@1714} size = 8
    0 = {DelegatingApplicationContextInitializer@1790}
    1 = {SharedMetadataReaderFactoryContextInitializer@1791}
    2 = {ContextIdApplicationContextInitializer@1792}
    3 = {ZkConfigApplicationContextInitializer@1793}
    4 = {ConfigurationWarningsApplicationContextInitializer@1794}
    5 = {RSocketPortInfoApplicationContextInitializer@1795}
    6 = {ServerPortInfoApplicationContextInitializer@1796}
    7 = {ConditionEvaluationReportLoggingListener@1797}
```

（3）回调

> SpringApplication#run()
>
> ->prepareContext(bootstrapCont...)
>
> ->applyInitializers(context)

```
for (ApplicationContextInitializer initializer : getInitializers()) {
    Class<?> requiredType =
GenericTypeResolver.resolveTypeArgument(initializer.getClass(),
            ApplicationContextInitializer.class);
    Assert.isInstanceOf(requiredType, context, "Unable to call initializer.");
    initializer.initialize(context);
}
```

4.6 启动Spring Boot拉取Zookeeper Server数据

4.6.1 Zookeeper Server数据准备

> /jack-config/product-service

```
{
    "id": "1",
    "product": "mobile",
    "price": "3000"
}
```

4.6.2 Curator连接ZK并获取指定节点数据

```
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-recipes</artifactId>
    <version>5.2.1</version>
</dependency>
```

```
public class ZkConfigApplicationContextInitializer implements
ApplicationContextInitializer {
    @Override
    public void initialize(ConfigurableApplicationContext context) {
        System.out.println("我被调用了...");
```

```
        String connectStr="192.168.0.8:2181";
        CuratorFramework curatorFramework = CuratorFrameworkFactory
                .builder()
                .connectionTimeoutMs(20000)
                .connectString(connectStr)
                .retryPolicy(new ExponentialBackoffRetry(1000, 3))
                .build();
        curatorFramework.start();
        try {
            byte[] bytes = curatorFramework.getData().forPath("/jack-
config/product-service");
            // String字符串的Json转成Map
            Map<String,Object> map = new ObjectMapper().readValue(new
String(bytes), Map.class);
            System.out.println("从zookeeper server获取到的值为: "+map);
        }catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

4.7 将map以属性源的形式保存到env中

```
// 将map转换成MapPropertySource
MapPropertySource mapPropertySource = new MapPropertySource("product-service-
remote-env", map);
ConfigurableEnvironment environment = context.getEnvironment();
environment.getPropertySources().addFirst(mapPropertySource);
System.out.println("env新增MapPropertySource成功.");
```

4.8 业务代码使用配置属性值

4.8.1 通过environment api获取

```
@RestController
@RequestMapping("/product")
public class ProductController {

    @Resource
    private Environment environment;

    @RequestMapping("/remote-config-env")
    public String remoteConfig(){
        return "[env] id: "+this.environment.getProperty("id")+
                ", product: "+this.environment.getProperty("product")
                +", price: "+this.environment.getProperty("price");
    }
}
```

4.8.2 @Value绑定

> 这个值也是从environment中的属性源获取到的，也就是字段绑定，底层是通过反射实现的

```
@Value("${id}")
private String Id;

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

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

@RequestMapping("/remote-config-value")
public String remoteConfigAnno(){
    return "[@Value] id: "+this.Id+
            ", product: "+this.product
            +", price: "+this.price;
}
```

4.9 添加对节点/jack-config/product-service的监听

```
// 永久的监听
CuratorCache curatorCache = CuratorCache.build(curatorFramework, "/jack-
config/product-service", CuratorCache.Options.SINGLE_NODE_CACHE);
CuratorCacheListener listener = CuratorCacheListener.builder().forAll(new
CuratorCacheListener() {
    // 一旦对应 /jack-config/product-service ZNode发生变化，就会回调这个方法
    @Override
    public void event(Type type, ChildData oldData, ChildData data) {
        if(type.equals(Type.NODE_CHANGED)){
            System.out.println("ZNode数据更新了，事件类型为: " + type);
        }
    }
}).build();
curatorCache.listenable().addListener(listener);
curatorCache.start();
```

4.10 获取到更新后的数据并重新赋值给environment

```
CuratorCacheListener listener = CuratorCacheListener.builder().forAll(new
CuratorCacheListener() {
    // 一旦对应 /jack-config/product-service ZNode发生变化，就会回调这个方法
    @Override
    public void event(Type type, ChildData oldData, ChildData data) {
        if(type.equals(Type.NODE_CHANGED)){
            System.out.println("ZNode数据更新了，事件类型为: " + type);

            try {
                Map<String, Object> updateMap = new ObjectMapper().readValue(new
String(data.getData()), Map.class);
                System.out.println("更新后的数据map为: "+updateMap);
```

```
                environment.getPropertySources().replace("product-service-remote-env", new MapPropertySource("product-service-remote-enve", updateMap));
            }catch (Exception e){
                e.printStackTrace();
            }
        }
    }
}).build();
```

4.11 解决@Value字段没有更新

（1）FieldDetail

```
public class FieldDetail {
    private Field field;   // 具体哪个字段
    private Object instance;   // 属于哪个实例
    public FieldDetail(Field field, Object instance) {
        this.field = field;
        this.instance = instance;
    }
    public Field getField() {
        return field;
    }

    public void setField(Field field) {
        this.field = field;
    }

    public Object getInstance() {
        return instance;
    }

    public void setInstance(Object instance) {
        this.instance = instance;
    }
}
```

（2）定义需要保存下来类的注解

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface JackRefreshScope {
}
```

（3）添加到目标类上，比如ProductController

```
@RestController
@RequestMapping("/product")
@JackRefreshScope
public class ProductController {
```

（4）使用后置处理器进行逻辑判断并保存

```
@Component
public class ParseJackRefreshScopeBeanPostProcessor implements BeanPostProcessor
{
    private Map<String, FieldDetail> fieldDetailMap=new HashMap<>();
    public Map<String, FieldDetail> getFieldDetailMap() {
        return fieldDetailMap;
    }
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName)
throws BeansException {
        Class<?> clazz = bean.getClass();
        if(clazz.isAnnotationPresent(JackRefreshScope.class)){
            System.out.println(clazz);   // class
com.jack.controller.ProductController
            for (Field field : clazz.getDeclaredFields()) {
                if(field.isAnnotationPresent(Value.class)){
                    Value value = field.getAnnotation(Value.class);
                    String val=value.value();  // 获取到了对应的value值   ${id}   -
-->  id
                    val=val.substring(2,val.indexOf("}"));   // id
                    /**
                     * val: orderid
                     * val: product
                     * val: orderprice
                     */
                    System.out.println("val: "+val);
                    // 保存 val 对应的Field和Field所在的clazz
                    /*
                        orderid, FieldDetail
                        product,FieldDetail
                        orderprice,FieldDetail
                     */

                    this.fieldDetailMap.put(val,new FieldDetail(field,bean));
                }
            }
            System.out.println("");
        }
        return BeanPostProcessor.super.postProcessAfterInitialization(bean,
beanName);
    }
}
```

（5）完善ZkConfigApplicationContextInitializer最后的逻辑

```
// 获取到有哪些字段可能需要更新
ParseJackRefreshScopeBeanPostProcessor parseJackRefreshScopeBeanPostProcessor =
        context.getBean("parseJackRefreshScopeBeanPostProcessor",
ParseJackRefreshScopeBeanPostProcessor.class);
Map<String, FieldDetail> fieldDetailMap =
parseJackRefreshScopeBeanPostProcessor.getFieldDetailMap();
for (String key : fieldDetailMap.keySet()) {
    if(updateMap.containsKey(key)){   // 判断远端发送过来的map数据中的key
        String value = environment.getProperty(key);
        Field field = fieldDetailMap.get(key).getField();
        field.setAccessible(true);
        // 反射更新字段的值
        field.set(fieldDetailMap.get(key).getInstance(),value);
    }
}
```

05 Spring Cloud Zookeeper实现配置中心

Spring Cloud Zookeeper：https://spring.io/projects/spring-cloud-zookeeper

（1）创建spring-cloud-zookeeper的spring boot项目，Spring Boot版本为2.7.2

（2）定义Spring Cloud的版本管理

```
<!--定义版本的管理-->
<dependencyManagement>
    <dependencies>
        <!--定义sc的版本-->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>2021.0.3</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>
```

（3）引入spring cloud zookeeper配置中心的依赖

```
<!-- spring cloud zookeeper config 配置中心-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zookeeper-config</artifactId>
</dependency>
```

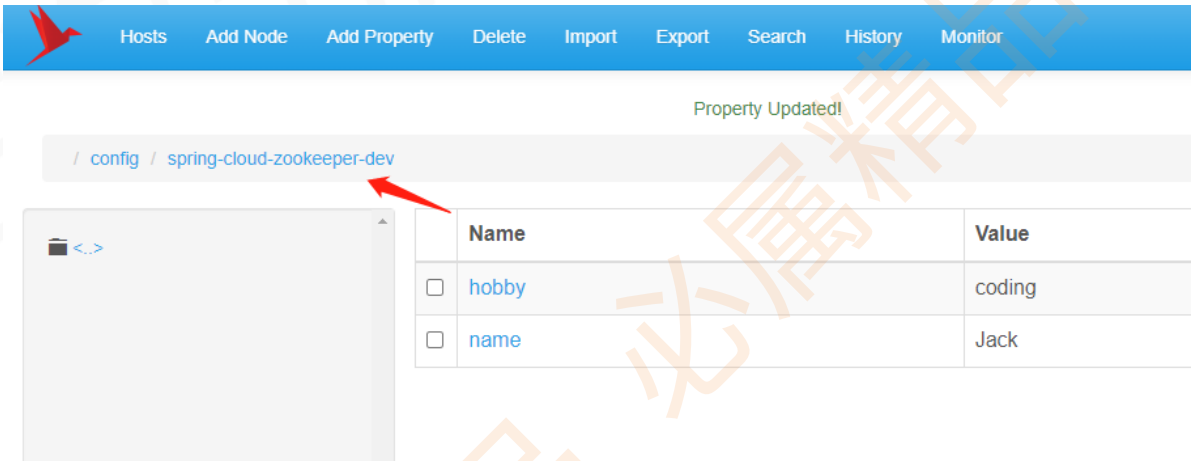
（4）引入bootstrap.yaml文件需要的依赖

```
<!-- bootstrap.yaml文件所需依赖 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bootstrap</artifactId>
</dependency>
```

（5）在bootstrap.yaml文件中编写配置中心相关的配置

```yaml
spring:
  profiles:
    active: dev
  application:
    name: spring-cloud-zookeeper
  cloud:
    zookeeper:
      config:
        root: config
        profile-separator: "-"
        enabled: true
      connect-string: 192.168.0.8:2181
```

（6）在Zookeeper Server上创建指定的节点配置

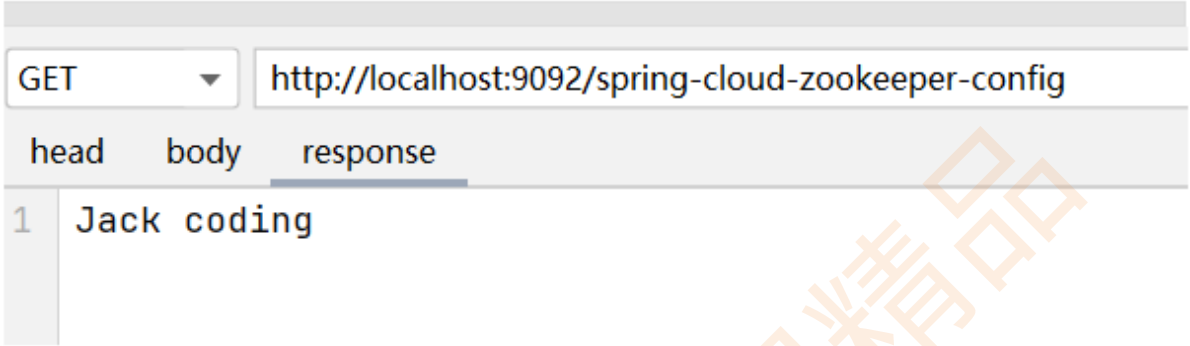


（7）编写测试代码

```java
@SpringBootApplication
@RestController
public class SpringCloudZookeeperApplication {

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

    @Resource
    private Environment environment;

    @RequestMapping("/spring-cloud-zookeeper-config-env")
    public String configEnv() {
        return "[zk server env] " + this.environment.getProperty("name")
                + " " + this.environment.getProperty("hobby");
    }

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

    @Value("${hobby}")
    private String hobby;
```

```
    @RequestMapping("/spring-cloud-zookeeper-config-value")
    public String configValue() {
        return "[zk server remote] " + this.name + " " + this.hobby;
    }
}
```

（8）访问测试



（9）在启动类上添加@RefreshScope注解，然后修改节点数据，在不重启本地项目的情况下访问测试

（10）实现原理

拉取Zookeeper Server的配置信息并保存到env的数据源中：PropertySourceBootstrapConfiguration#initialize(context)

销毁原有的Bean，重新创建Bean：ConfigurationPropertiesRebinder#onApplicationEvent