二

13 Curator：如何降低 ZooKeeper 使用的复杂性?

今天我们开始学习 Curator，并了解如何通过其降低 ZooKeeper 使用的复杂性。

作为进阶篇的最后一节课，与前面几节课侧重于会话的知识不同，今天这节课比较轻松易学，我会给你介绍一个日常可以提高开发 ZooKeeper 服务效率和质量的开源框架 Curator。

什么是 Curator

首先我们来介绍一下什么是 Curator，Curator 是一套开源的，Java 语言编程的 ZooKeeper 客户端框架，Curator 把我们平时常用的很多 ZooKeeper 服务开发功能做了封装，例如 Leader 选举、分布式计数器、分布式锁。这就减少了技术人员在使用 ZooKeeper 时的大部分底层细节开发工作。在会话重新连接、Watch 反复注册、多种异常处理等使用场景中，用原生的 ZooKeeper 实现起来就比较复杂。而在使用 Curator 时，由于其对这些功能都做了高度的封装，使用起来更加简单，不但减少了开发时间，而且增强了程序的可靠性。

经过上面的介绍，相信你对 Curator 有了较大的兴趣，接下来我们就来学习一下 Curator 的使用方式。我还是从我们比较熟悉的会话创建入手，来讲解 Curator 在日常工作中经常用到的知识点和使用技巧。

工程准备

如果想在我们开发的工程中使用 Curator 框架，首先要引入相关的开发包。这里我们以 Maven 工程为例，如下面的代码所示，我们通过将 Curator 相关的引用包配置到 Maven 工程的 pom 文件中，将 Curaotr 框架引用到工程项目里，在配置文件中分别引用了两个 Curator 相关的包，第一个是 curator-framework 包，该包是对 ZooKeeper 底层 API 的一些封装。另一个是 curator-recipes 包，该包封装了一些 ZooKeeper 服务的高级特性，如：Cache 事件监听、选举、分布式锁、分布式 Barrier。

```
<dependency>
 <groupId>org.apache.curator</groupId>
```

```
  <artifactId>curator-framework</artifactId>
  <version>2.12.0</version>
 </dependency>
 <dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-recipes</artifactId>
  <version>2.12.0</version>
 </dependency>
```

基础方法

部署好工程的开发环境后，接下来我们就可以利用 Curator 开发 ZooKeeper 服务了。首先，我们先看一下 Curator 框架提供的 API 编码风格。Curator 的编码采用“Fluent Style”流式，而所谓的流式编码风格在 Scala 等编程语言中也广泛被采用。如下面的代码所示，流式编程风格与传统的编码方式有很大不同，传统的代码风格会尽量把功能拆分成不同的函数，并分行书写。而流式编程功能函数则是按照逻辑的先后顺序，采用调用的方式，从而在代码书写上更加连贯，逻辑衔接也更加清晰。

下面这段代码的作用就是创建一个“/my/path”路径的节点到 ZooKeeper 服务端，之后将 myData 变量的数据作为该节点的数据。

```
client.create().forPath("/my/path", myData)
```

会话创建

了解了 Curator 的编码风格和函数的调用规则后。我们来创建一个会话。

通过前几节课的学习，我们知道一个 ZooKeeper 服务中，客户端与服务端要想相互通信，完成特定的功能，首先要做的就是创建一个会话连接。那么如何利用 Curator 来创建一个会话连接呢？我们来看下面的这段代码：

```
RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 3);
CuratorFramework client = CuratorFrameworkFactory.builder()
                .connectString("192.168.128.129:2181")
                .sessionTimeoutMs(5000)  // 会话超时时间
```

```
                .connectionTimeoutMs(5000) // 连接超时时间
                .retryPolicy(retryPolicy)
                .namespace("base") // 包含隔离名称
                .build();
client.start();
```

这段代码的编码风格采用了流式方式，最核心的类是 CuratorFramework 类，该类的作用是定义一个 ZooKeeper 客户端对象，并在之后的上下文中使用。在定义 CuratorFramework 对象实例的时候，我们使用了 CuratorFrameworkFactory 工厂方法，并指定了 connectionString 服务器地址列表、retryPolicy 重试策略 、sessionTimeoutMs 会话超时时间、connectionTimeoutMs 会话创建超时时间。下面我们分别对这几个参数进行讲解：

- **connectionString**：服务器地址列表，服务器地址列表参数的格式是 host1:port1,host2:port2。在指定服务器地址列表的时候可以是一个地址，也可以是多个地址。如果是多个地址，那么每个服务器地址列表用逗号分隔。
- **retryPolicy**：重试策略，当客户端异常退出或者与服务端失去连接的时候，可以通过设置客户端重新连接 ZooKeeper 服务端。而 Curator 提供了 一次重试、多次重试等不同种类的实现方式。在 Curator 内部，可以通过判断服务器返回的 keeperException 的状态代码来判断是否进行重试处理，如果返回的是 OK 表示一切操作都没有问题，而 SYSTEMERROR 表示系统或服务端错误。
- **超时时间**：在 Curator 客户端创建过程中，有两个超时时间的设置。这也是平时你容易混淆的地方。一个是 sessionTimeoutMs 会话超时时间，用来设置该条会话在 ZooKeeper 服务端的失效时间。另一个是 connectionTimeoutMs 客户端创建会话的超时时间，用来限制客户端发起一个会话连接到接收 ZooKeeper 服务端应答的时间。**sessionTimeoutMs 作用在服务端，而 connectionTimeoutMs 作用在客户端**，请你在平时的开发中多注意。

在完成了客户端的创建和实例后，接下来我们就来看一看如何使用 Curator 对节点进行创建、删除、更新等基础操作。

创建节点

创建节点的方式如下面的代码所示，回顾我们之前课程中讲到的内容，描述一个节点要包括节点的类型，即临时节点还是持久节点、节点的数据信息、节点是否是有序节点等属性和性质。

```
client.create().withMode(CreateMode.EPHEMERAL).forPath("path","init".getBytes());
```

在 Curator 中，使用 create 函数创建数据节点，并通过 withMode 函数指定节点是临时节点还是持久节点，之后调用 forPath 函数来指定节点的路径和数据信息。

更新节点

说完了节点的创建，接下来我们再看一下节点的更新操作，如下面的代码所示，我们通过客户端实例的 setData() 方法更新 ZooKeeper 服务上的数据节点，在setData 方法的后边，通过 forPath 函数来指定更新的数据节点路径以及要更新的数据。

```
client.setData().forPath("path","data".getBytes());
```

删除节点

如果我们想删除 ZooKeeper 服务器上的节点，可以使用 Curator 中的 client.delete()。如下面的代码所示，在流式调用中有几个功能函数：

```
client.delete().guaranteed().deletingChildrenIfNeeded().withVersion(10086).forPath(
```

下面我将为你分别讲解这几个函数的功能：

- **guaranteed**：该函数的功能如字面意思一样，主要起到一个保障删除成功的作用，其底层工作方式是：**只要该客户端的会话有效，就会在后台持续发起删除请求，直到该数据节点在 ZooKeeper 服务端被删除。**
- **deletingChildrenIfNeeded**：指定了该函数后，系统在删除该数据节点的时候会以递归的方式直接删除其子节点，以及子节点的子节点。
- **withVersion**：该函数的作用是指定要删除的数据节点版本。
- **forPath**：该函数的作用是指定要删除的数据节点的路径，也就是我们要删除的节点。

高级应用

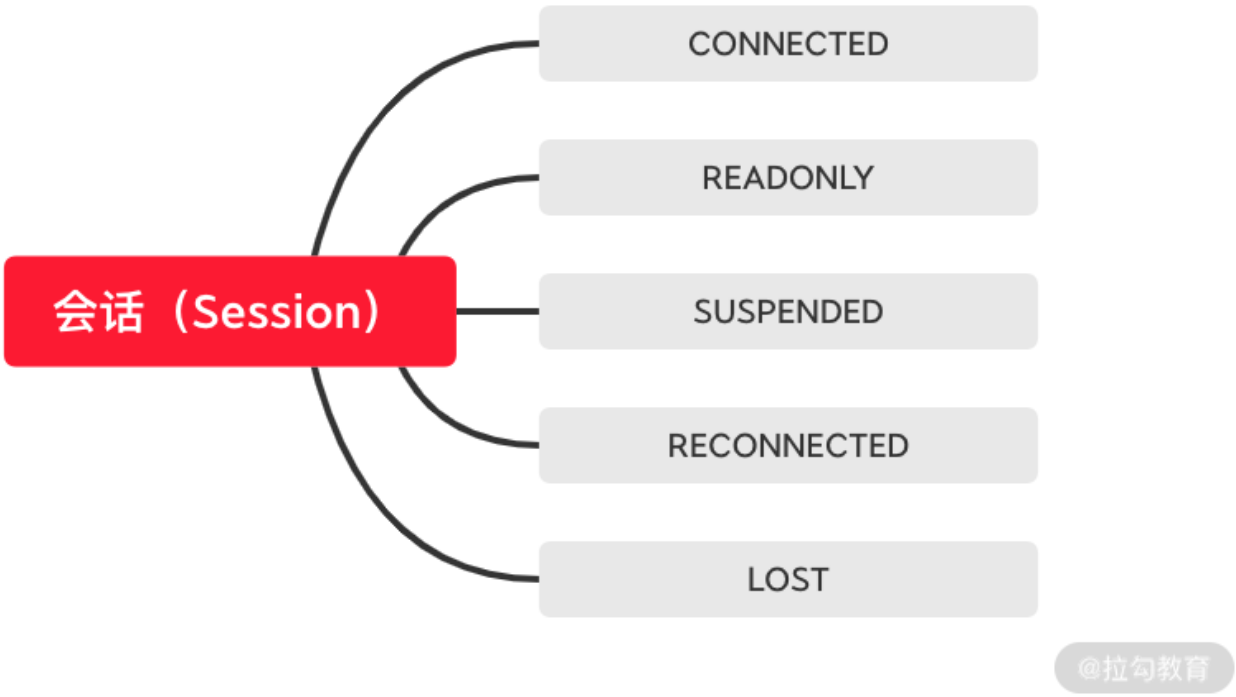
在上面的工程部署中，我们提到使用 Curator 需要引用两个包，而其中的 curator-recipes 包如它的名字一样，即“Curator 菜谱”，它提供了很多丰富的功能，很多高级特性都是在这个包中实现的。

异常处理

无论使用什么开发语言和框架，程序都会出现错误与异常等情况，而 ZooKeeper 服务使客户端与服务端通过网络进行通信的方式协同工作，由于网络不稳定和不可靠等原因，使得网络中断或高延迟异常情况出现得更加频繁，所以处理好异常是决定 ZooKeeper 服务质量的关键。

在 Curator 中，客户端与服务端建立会话连接后会一直监控该连接的运行情况。在底层实现中通过

ConnectionStateListener 来监控会话的连接状态，当连接状态改变的时候，根据参数设置 ZooKeeper 服务会采取不同的处理方式，而一个会话基本有六种状态，如下图所示：



下面我来为你详细讲解这六种状态的作用：

- **CONNECTED（已连接状态）**：当客户端发起的会话成功连接到服务端后，该条会话的状态变为 CONNECTED 已连接状态。
- **READONLY（只读状态）**：当一个客户端会话调用 CuratorFrameworkFactory.Builder.canBeReadOnly() 的时候，该会话会一直处于只读模式，直到重新设置该条会话的状态类型。
- **SUSPENDED（会话连接挂起状态）**：当进行 Leader 选举和 lock 锁等操作时，需要先挂起客户端的连接。**注意这里的会话挂起并不等于关闭会话，也不会触发诸如删除临时节点等操作。**
- **RECONNECTED（重新连接状态）**：当已经与服务端成功连接的客户端断开后，尝试

再次连接服务端后，该条会话的状态为 RECONNECTED，也就是重新连接。重新连接的会话会作为一条新会话在服务端运行，之前的临时节点等信息不会被保留。

- **LOST（会话丢失状态）**：这个比较好理解，当客户端与服务器端因为异常或超时，导致会话关闭时，该条会话的状态就变为 LOST。

在这里，我们以 Curator 捕捉到的会话关闭后重新发起的与服务器端的连接为例，介绍 Curator 是如何进行处理的。

首先，要想处理一个会话关闭的异常，必须要捕获这条异常事件。如下面的代码所示，我们创建了一个 sessionConnectionListener 类，用来监听事件的异常关闭操作。该类实现了 ConnectionStateListener 接口，这个接口是 Curator 的内部接口，负责监控会话的状态变化。当会话状态发生改变时，系统就会调用 stateChanged 函数，我们可以在 stateChanged 函数中编写会话状态的处理逻辑，这里我设置了当会话改变的时候，客户端会一直尝试连接服务端直到接连成功。

```
class sessionConnectionListener implements ConnectionStateListener {

  public void stateChanged(CuratorFramework curatorFramework, ConnectionState conne

  while(true){

  ...

    reConnectZKServer(..);

  }

  }

}
```

总体来说，使用 Curator 处理会话异常非常简单，只要实现 ConnectionStateListener 接口并在对应的状态变更函数中实现自己的处理逻辑即可。

Leader 选举

除了异常处理，接下来我们再介绍一个在日常工作中经常要解决的问题，即开发 ZooKeeper 集群的相关功能。

在分布式环境中，ZooKeeper 集群起到了关键作用。在之前的课程中我们讲过，Leader 选举是保证 ZooKeeper 集群可用性的解决方案，可以避免在集群使用中出现单点失效等问题。在 ZooKeeper 服务开始运行的时候，首先会选举出 Leader 节点服务器，之后在服务运行过程中，Leader 节点服务器失效时，又会重新在集群中进行 Leader 节点的选举操

作。

而在日常开发中，使用 ZooKeeper 原生的 API 开发 Leader 选举相关的功能相对比较复杂。Curator 框架中的 recipe 包为我们提供了高效的，方便易用的工具函数，分别是 LeaderSelector 和 LeaderLatch。

LeaderSelector 函数的功能是选举 ZooKeeper 中的 Leader 服务器，其具体实现如下面的代码所示，通过构造函数的方式实例化。在一个构造方法中，第一个参数为会话的客户端实例 CuratorFramework，第二个参数是 LeaderSelectorListener 对象，它是在被选举成 Leader 服务器的事件触发时执行的回调函数。

```
public LeaderSelector(CuratorFramework client, String mutexPath,LeaderSelectorListe
```

第二个工具函数是 LeaderLatch，它也是实现 Leader 服务器功能的函数。与 LeaderSelector 函数不同的是，LeaderLatch 函数将 ZooKeeper 服务中的所有机器都作为 Leader 候选机器，在服务运行期间，集群中的机器轮流作为 Leader 机器参与集群工作。具体方式如下面的代码所示：

```
public LeaderLatch(CuratorFramework client, String latchPath)
```

LeaderLatch 方法也是通过构造函数的方式实现 LeaderLatch 实例对象，并传入会话客户端对象 CuratorFramework，而构造函数的第二个参数 latchPath 是一个字符类型字段。在服务启动后， 调用了 LeaderLatch 的客户端会和其他使用相同 latch path 的服务器交涉，然后选择其中一个作为 Leader 服务器。

结束

通过本课时的学习，我们掌握了 Curator 框架的主要使用方法，在日常工作中能够运用 Curator 提高我们的开发效率和程序的可靠性。

Curator 是一个功能丰富的框架，除了上述功能外，还可以实现诸如分布式锁、分布式计数器、分布式队列等功能。本课时主要起到引导作用，希望你在这节课的基础上进一步学习相关的技术知识，在日常开发中多使用 Curator 提高代码的开发效率和工程服务的质量。

上一页 下一页