6. 注册中心优化 - 手写 RPC 框架项目教程 - 编程导航教程

> 仅供 [编程导航](https://www.code-nav.cn/post/1816420035119853569) 内部成员观看，请勿对外分享！一、需求分析上节教程中，我们基于 Etcd 完成了基。

一、需求分析

上节教程中，我们基于 Etcd 完成了基础的注册中心，能够注册和获取服务和节点信息。1747226499385180161_0.8222404767388451

但目前系统仅仅是处于可用的程度，还有很多需要解决的问题和可优化点：

1. 数据一致性：服务提供者如果下线了，注册中心需要即时更新，剔除下线节点。否则消费者可能会调用到已经下线的节点。
2. 性能优化：服务消费者每次都需要从注册中心获取服务，可以使用缓存进行优化。
3. 高可用性：保证注册中心本身不会宕机。1747226499385180161_0.99790318171851
4. 可扩展性：实现更多其他种类的注册中心。

本节教程，鱼皮将带大家实践 4 个注册中心的优化点：1747226499385180161_0.6396524378953539

1. 心跳检测和续期机制
2. 服务节点下线机制
3. 消费端服务缓存 1747226499385180161_0.2939630858451385
4. 基于 ZooKeeper 的注册中心实现

二、注册中心优化

心跳检测和续期机制

心跳检测介绍

心跳检测（俗称 heartBeat）是一种用于监测系统是否正常工作的机制。它通过定期发送 **心跳信号**（请求）来检测目标系统的状态。

如果接收方在一定时间内没有收到心跳信号或者未能正常响应请求，就会认为目标系统故障或不可用，从而触发相应的处理或告警机制。1747226499385180161_0.5275712532747272

心跳检测的应用场景非常广泛，尤其是在分布式、微服务系统中，比如集群管理、服务健康检查等。

之前有同学问，我们怎么检测自己做的 web 后端是否正常运行呢？

一个最简单的方法，就是写一个心跳检测接口，比如：1747226499385180161_0.018049452834527635

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
class HealthCheckController {
    // 健康检查接口
    @GetMapping("/actuator/health")
    public String healthCheck() {
        // 在这里可以添加其他健康检查逻辑，例如检查数据库连接、第三方服务等

        // 返回一个简单的健康状态
        return "OK";
    }
}
```

然后我们只需要执行一个脚本，定期调用这个接口，如果调用失败，就知道系统故障了。

方案设计

1）从心跳检测的概念来看，实现心跳检测一般需要 2 个关键：定时、网络请求。

但是使用 Etcd 实现心跳检测会更简单一些，因为 Etcd 自带了 key 过期机制，我们不妨换个思路：给节点注册信息一个“生命倒计时”，让节点定期 **续期**，重置 **自己的** 倒计时。如果节点已宕机，一直不续期，Etcd 就会对 key 进行过期删除。

一句话总结：到时间还不续期就是寄了。1747226499385180161_0.38513217914498377

在 Etcd 中，我们要实现心跳检测和续期机制，可以遵循如下步骤：

1. 服务提供者向 Etcd 注册自己的服务信息，并在注册时设置 TTL（生存时间）。
2. Etcd 在接收到服务提供者的注册信息后，会自动维护服务信息的 TTL，并在 TTL 过期时删除该服务信息。
3. 服务提供者定期请求 Etcd 续签自己的注册信息，重写 TTL。1747226499385180161_0.5028085591017835

需要注意的是，续期时间一定要小于过期时间，允许一次容错的机会。1747226499385180161_0.5577412158752102

2）每个服务提供者都需要找到自己注册的节点、续期自己的节点，但问题是，怎么找到当前服务提供者项目自己的节点呢?

那就充分利用本地的特性，在服务提供者本地维护一个 **已注册节点集合**，注册时添加节点 key 到集合中，只需要续期集合内的 key 即可。

开发实现

1）给注册中心 `Registry` 接口补充心跳检测方法，代码如下：

```
package com.yupi.yurpc.registry;

import com.yupi.yurpc.config.RegistryConfig;
import com.yupi.yurpc.model.ServiceMetaInfo;

import java.util.List;

/**
 * 注册中心
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public interface Registry {

    ...

    /**
     * 心跳检测（服务端）
     */
    void heartBeat();
}
```

2）维护续期节点集合。1747226499385180161_0.33153812339792466

定义一个本机注册的节点 key 集合，用于维护续期：

```
/**
 * 本机注册的节点 key 集合（用于维护续期）
 */
private final Set<String> localRegisterNodeKeySet = new HashSet<>();
```

在服务注册时，需要将节点添加到集合中，代码如下：1747226499385180161_0.43418642375255057

```
public void register(ServiceMetaInfo serviceMetaInfo) throws Exception {
    // 创建 Lease 和 KV 客户端
    Lease leaseClient = client.getLeaseClient();

    // 创建一个 30 秒的租约
    long leaseId = leaseClient.grant(30).get().getID();

    // 设置要存储的键值对
    String registerKey = ETCD_ROOT_PATH + serviceMetaInfo.getServiceNodeKey();
    ByteSequence key = ByteSequence.from(registerKey, StandardCharsets.UTF_8);
    ByteSequence value = ByteSequence.from(JSONUtil.toJsonStr(serviceMetaInfo), StandardCharsets.UTF_8);

    // 将键值对与租约关联起来，并设置过期时间
    PutOption putOption = PutOption.builder().withLeaseId(leaseId).build();
    kvClient.put(key, value, putOption).get();

    // 添加节点信息到本地缓存
    localRegisterNodeKeySet.add(registerKey);
}
```

同理，在服务注销时，也要从集合中移除对应节点：

```
public void unRegister(ServiceMetaInfo serviceMetaInfo) {
    String registerKey = ETCD_ROOT_PATH + serviceMetaInfo.getServiceNodeKey();
    kvClient.delete(ByteSequence.from(registerKey, StandardCharsets.UTF_8));
    // 也要从本地缓存移除
    localRegisterNodeKeySet.remove(registerKey);
}
```

3）在 `EtcdRegistry` 中实现 heartBeat 方法。

可以使用 Hutool 工具类的 CronUtil 实现定时任务，对所有集合中的节点执行 **重新注册** 操作，这是一个小 trick，就相当于续签了。

心跳检测方法的代码如下：1747226499385180161_0.03360381135125823

```
@Override
public void heartBeat() {
    // 10 秒续签一次
    CronUtil.schedule("*/10 * * * * *", new Task() {
        @Override
        public void execute() {
            // 遍历本节点所有的 key
            for (String key : localRegisterNodeKeySet) {
                try {
                    List<KeyValue> keyValues = kvClient.get(ByteSequence.from(key, StandardCharsets.UTF_8))
                            .get()
                            .getKvs();
                    // 该节点已过期（需要重启节点才能重新注册）
                    if (CollUtil.isEmpty(keyValues)) {
                        continue;
                    }
```

```
                    // 节点未过期，重新注册（相当于续签）
                    KeyValue keyValue = keyValues.get(0);
                    String value = keyValue.getValue().toString(StandardCharsets.UTF_8);
                    ServiceMetaInfo serviceMetaInfo = JSONUtil.toBean(value, ServiceMetaInfo.class);
                    register(serviceMetaInfo);
                } catch (Exception e) {
                    throw new RuntimeException(key + "续签失败", e);
                }
            }
        }
    });

    // 支持秒级别定时任务
    CronUtil.setMatchSecond(true);
    CronUtil.start();
}
```

采用这种实现方案的好处是，即时 Etcd 注册中心的数据出现了丢失，通过心跳检测机制也会重新注册节点信息。

4）开启 heartBeat。1747226499385180161_0.06576591330427717

在注册中心初始化的 init 方法中，调用 heartBeat 方法即可。

代码如下：

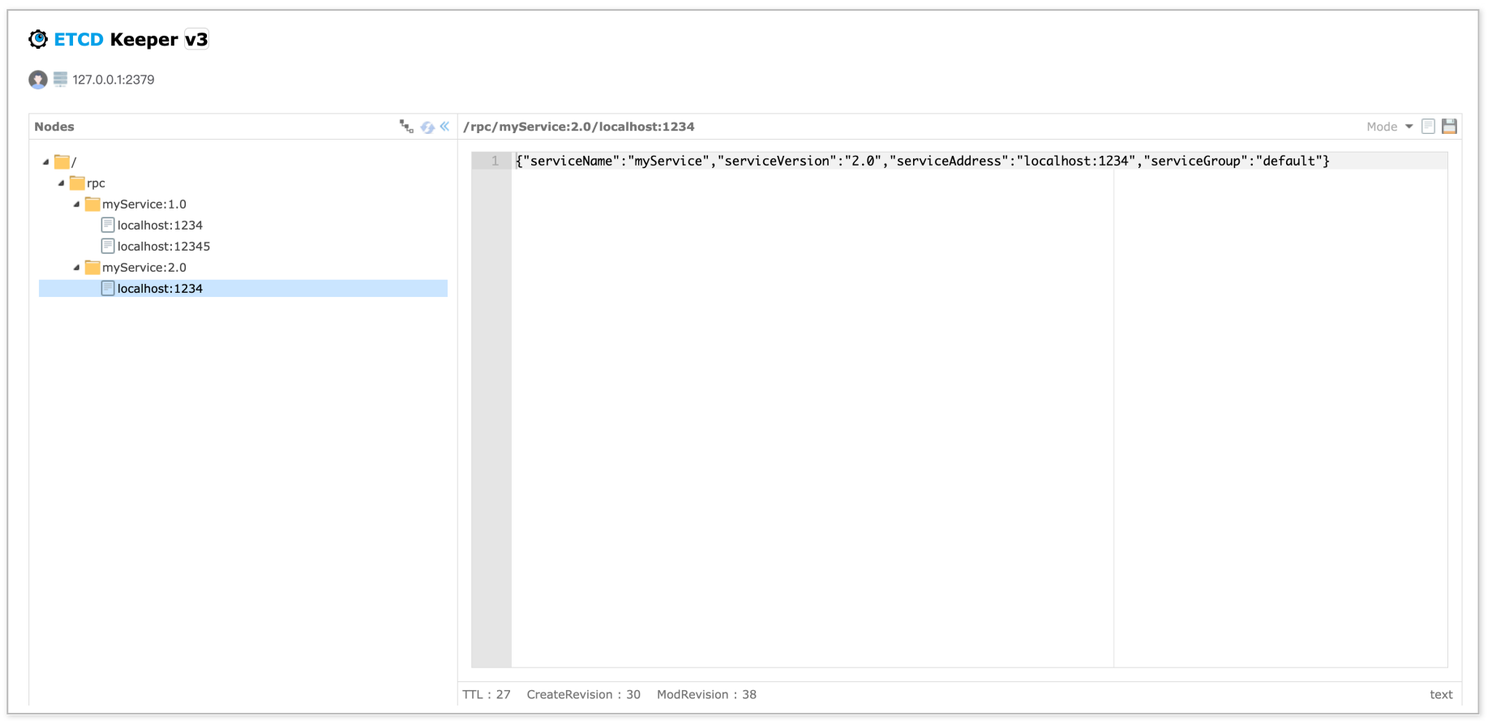
```
@Override
public void init(RegistryConfig registryConfig) {
    client = Client.builder()
            .endpoints(registryConfig.getAddress())
            .connectTimeout(Duration.ofMillis(registryConfig.getTimeout()))
            .build();
    kvClient = client.getKVClient();
    heartBeat();
}
```

测试

完善之前的 `RegistryTest` 单元测试代码：

```
public class RegistryTest {

    final Registry registry = new EtcdRegistry();

    @Before
    public void init() {
        RegistryConfig registryConfig = new RegistryConfig();
        registryConfig.setAddress("http://localhost:2379");
        registry.init(registryConfig);
    }

    ...

    @Test
    public void heartBeat() throws Exception {
        // init 方法中已经执行心跳检测了
        register();
        // 阻塞 1 分钟
        Thread.sleep(60 * 1000L);
    }
}
```

使用可视化工具观察节点底部的过期时间，当 TTL 到 20 左右的时候，又会重置为 30，说明心跳检测和续期机制正常执行。



服务节点下线机制

当服务提供者节点宕机时，应该从注册中心移除掉已注册的节点，否则会影响消费端调用。所以我们需要设计一套服务节点下线机制。

方案设计

服务节点下线又分为：1747226499385180161_0.7005790076607636

- 主动下线：服务提供者项目正常退出时，主动从注册中心移除注册信息。
- 被动下线：服务提供者项目异常推出时，利用 Etcd 的 key 过期机制自动移除。

被动下线已经可以利用 Etcd 的机制实现了，我们主要开发主动下线。

问题是，怎么在 Java 项目正常退出时，执行某个操作呢？1747226499385180161_0.29999553378000265

其实，非常简单，利用 JVM 的 ShutdownHook 就能实现。

JVM 的 ShutdownHook 是 Java 虚拟机提供的一种机制，允许开发者在 JVM 即将关闭之前执行一些清理工作或其他必要的操作，例如关闭数据库连接、释放资源、保存临时数据等。

Spring Boot 也提供了类似的优雅停机能力。1747226499385180161_0.3600655364436567

开发实现

1）完善 Etcd 注册中心的 `destroy` 方法，补充下线节点的逻辑。

代码如下：1747226499385180161_0.45775561305106427

```
public void destroy() {
    System.out.println("当前节点下线");
    // 下线节点
    // 遍历本节点所有的 key
    for (String key : localRegisterNodeKeySet) {
        try {
            kvClient.delete(ByteSequence.from(key, StandardCharsets.UTF_8)).get();
        } catch (Exception e) {
            throw new RuntimeException(key + "节点下线失败");
        }
    }

    // 释放资源
    if (kvClient != null) {
        kvClient.close();
    }
    if (client != null) {
        client.close();
    }
}
```

2）在 `RpcApplication` 的 init 方法中，注册 Shutdown Hook，当程序正常退出时会执行注册中心的 destroy 方法。

代码如下：1747226499385180161_0.1640651240049995

```
public static void init(RpcConfig newRpcConfig) {
    rpcConfig = newRpcConfig;
    log.info("rpc init, config = {}", newRpcConfig.toString());
    // 注册中心初始化
    RegistryConfig registryConfig = rpcConfig.getRegistryConfig();
    Registry registry = RegistryFactory.getInstance(registryConfig.getRegistry());
    registry.init(registryConfig);
    log.info("registry init, config = {}", registryConfig);

    // 创建并注册 Shutdown Hook，JVM 退出时执行操作
    Runtime.getRuntime().addShutdownHook(new Thread(registry::destroy));
}
```

测试

测试方法很简单：1747226499385180161_0.6957642944523779

1. 启动服务提供者，然后观察服务是否成功被注册

2. 正常停止服务提供者，然后观察服务信息是否被删除

消费端服务缓存

正常情况下，服务节点信息列表的更新频率是不高的，所以在服务消费者从注册中心获取到服务节点信息列表后，完全可以 **缓存在本地**，下次就不用再请求注册中心获取了，能够提高性能。1747226499385180161_0.8433736935161091

1、增加本地缓存

本地缓存的实现很简单，用一个列表来存储服务信息即可，提供操作列表的基本方法，包括：写缓存、读缓存、清空缓存。

⭐ 注意，本教程为了大家循序渐进的学习，暂时先只考虑单服务（相同 serviceKey）的缓存。如果要实现多服务缓存，可以改为使用 Map 接口。参考本次提交的代码： https://github.com/liyupi/yu-rpc/commit/c420222f4673114ee760b7875d68635902625ce9 1747226499385180161_0.6163403951550526

在 `registry` 包下新增缓存类 `RegistryServiceCache` ，代码如下：

```
package com.yupi.yurpc.registry;

import com.yupi.yurpc.model.ServiceMetaInfo;
import java.util.List;

/**
 * 注册中心服务本地缓存
 */
public class RegistryServiceCache {

    /**
     * 服务缓存
     */
    List<ServiceMetaInfo> serviceCache;

    /**
     * 写缓存
     *
     * @param newServiceCache
     * @return
     */
    void writeCache(List<ServiceMetaInfo> newServiceCache) {
        this.serviceCache = newServiceCache;
    }

    /**
     * 读缓存
     *
     * @return
     */
```

```
    List<ServiceMetaInfo> readCache() {
        return this.serviceCache;
    }

    /**
     * 清空缓存
     */
    void clearCache() {
        this.serviceCache = null;
    }
}
```

2、使用本地缓存

1）修改 `EtcdRegisty` 的代码，使用本地缓存对象：

```
/**
 * 注册中心服务缓存
 */
private final RegistryServiceCache registryServiceCache = new RegistryServiceCache();
```

2）修改服务发现逻辑，优先从缓存获取服务；如果没有缓存，再从注册中心获取，并且设置到缓存中。
1747226499385180161_0.7191311056811978

代码如下：

```
@Override
public List<ServiceMetaInfo> serviceDiscovery(String serviceKey) {
    // 优先从缓存获取服务
    List<ServiceMetaInfo> cachedServiceMetaInfoList = registryServiceCache.readCache();
    if (cachedServiceMetaInfoList != null) {
        return cachedServiceMetaInfoList;
    }

    // 前缀搜索，结尾一定要加 '/'
    String searchPrefix = ETCD_ROOT_PATH + serviceKey + "/";

    try {
        // 前缀查询
        GetOption getOption = GetOption.builder().isPrefix(true).build();
        List<KeyValue> keyValues = kvClient.get(
                        ByteSequence.from(searchPrefix, StandardCharsets.UTF_8),
                        getOption)
                .get()
                .getKvs();
        // 解析服务信息
        List<ServiceMetaInfo> serviceMetaInfoList = keyValues.stream()
                .map(keyValue -> {
                    String key = keyValue.getKey().toString(StandardCharsets.UTF_8);
                    // 监听 key 的变化
                    watch(key);
                    String value = keyValue.getValue().toString(StandardCharsets.UTF_8);
                    return JSONUtil.toBean(value, ServiceMetaInfo.class);
                })
                .collect(Collectors.toList());
        // 写入服务缓存
        registryServiceCache.writeCache(serviceMetaInfoList);
        return serviceMetaInfoList;
    } catch (Exception e) {
        throw new RuntimeException("获取服务列表失败", e);
    }
}
```

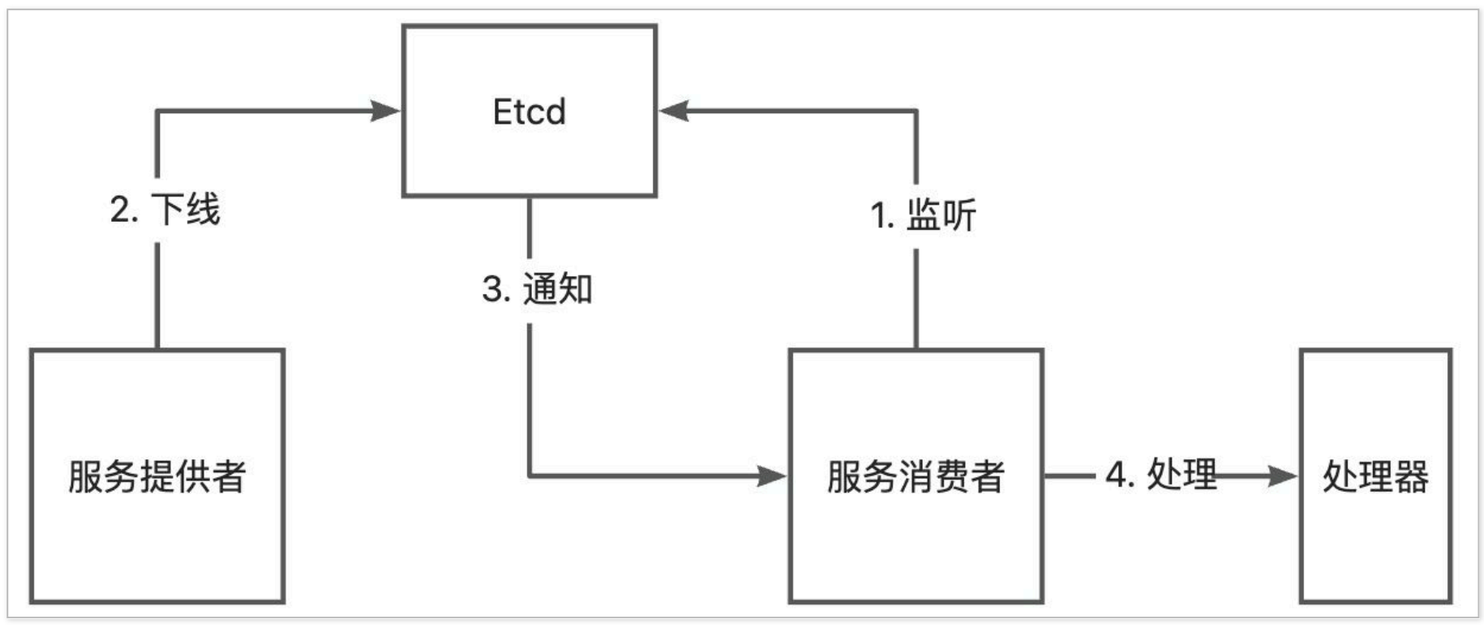
3、服务缓存更新 - 监听机制

当服务注册信息发生变更（比如节点下线）时，需要即时更新消费端缓存。

问题是，怎么知道服务注册信息什么时候发生变更呢?

这就需要我们使用 Etcd 的 watch 监听机制，当监听的某个 key 发生修改或删除时，就会触发事件来通知监听者。
1747226499385180161_0.4731225207471994

如图：



什么时候去创建 watch 监听器呢？1747226499385180161_0.2084213441067828

我们首先要明确 watch 监听是服务消费者还是服务提供者执行的。由于我们的目标是更新缓存，缓存是在服务消费端维护和使用的，所以也应该是服务消费端去 watch。

也就是说，只有服务消费者执行的方法中，可以创建 watch 监听器，那么比较合适的位置就是服务发现方法（serviceDiscovery）。可以对本次获取到的所有服务节点 key 进行监听。

还需要防止重复监听同一个 key，可以通过定义一个已监听 key 的集合来实现。1747226499385180161_0.5996035138271965

下面我们来开发编码。

1）Registry 注册中心接口补充监听 key 的方法，代码如下：

```
package com.yupi.yurpc.registry;

import com.yupi.yurpc.config.RegistryConfig;
import com.yupi.yurpc.model.ServiceMetaInfo;

import java.util.List;

/**
 * 注册中心
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public interface Registry {

    /**
     * 监听（消费端）
     *
     * @param serviceNodeKey
     */
    void watch(String serviceNodeKey);
}
```

2） `EtcdRegistry` 类中，新增监听 key 的集合。

可以使用 `ConcurrentHashSet` 防止并发冲突，代码如下：

```
/**
 * 正在监听的 key 集合
 */
private final Set<String> watchingKeySet = new ConcurrentHashSet<>();
```

3）在 `EtcdRegistry` 类中实现监听 key 的方法。

通过调用 Etcd 的 `WatchClient` 实现监听，如果出现了 `DELETE` key 删除事件，则清理服务注册缓存。

注意，即使 key 在注册中心被删除后再重新设置，之前的监听依旧生效。所以我们只监听首次加入到监听集合的 key，防止重复。1747226499385180161_0.9238299432414361

代码如下：

```
/**
 * 监听（消费端）
 *
 * @param serviceNodeKey
 */
@Override
public void watch(String serviceNodeKey) {
    Watch watchClient = client.getWatchClient();
    // 之前未被监听，开启监听
    boolean newWatch = watchingKeySet.add(serviceNodeKey);
    if (newWatch) {
        watchClient.watch(ByteSequence.from(serviceNodeKey, StandardCharsets.UTF_8), response -> {
            for (WatchEvent event : response.getEvents()) {
                switch (event.getEventType()) {
                    // key 删除时触发
                    case DELETE:
                        // 清理注册服务缓存
                        registryServiceCache.clearCache();
                        break;
                    case PUT:
                    default:
                        break;
                }
            }
        });
    }
}
```

4）在消费端获取服务时调用 watch 方法，对获取到的服务节点 key 进行监听。1747226499385180161_0.514085884388976

修改服务发现方法的代码如下：

```
public List<ServiceMetaInfo> serviceDiscovery(String serviceKey) {
    // 优先从缓存获取服务
    List<ServiceMetaInfo> cachedServiceMetaInfoList = registryServiceCache.readCache();
    if (cachedServiceMetaInfoList != null) {
        return cachedServiceMetaInfoList;
    }

    // 前缀搜索，结尾一定要加 '/'
    String searchPrefix = ETCD_ROOT_PATH + serviceKey + "/";

    try {
        // 前缀查询
        GetOption getOption = GetOption.builder().isPrefix(true).build();
        List<KeyValue> keyValues = kvClient.get(
                        ByteSequence.from(searchPrefix, StandardCharsets.UTF_8),
                        getOption)
                .get()
                .getKvs();
        // 解析服务信息
        List<ServiceMetaInfo> serviceMetaInfoList = keyValues.stream()
                .map(keyValue -> {
                    String key = keyValue.getKey().toString(StandardCharsets.UTF_8);
                    // 监听 key 的变化
                    watch(key);
                    String value = keyValue.getValue().toString(StandardCharsets.UTF_8);
                    return JSONUtil.toBean(value, ServiceMetaInfo.class);
                })
```

```
                .collect(Collectors.toList());
        // 写入服务缓存
        registryServiceCache.writeCache(serviceMetaInfoList);
        return serviceMetaInfoList;
    } catch (Exception e) {
        throw new RuntimeException("获取服务列表失败", e);
    }
}
```

5）测试。1747226499385180161_0.46768237307151805

可以使用如下步骤，通过 debug 进行测试：

1. 先启动服务提供者
2. 修改服务消费者项目，连续调用服务 3 次，通过 debug 可以发现，第一次查注册中心、第二次查询缓存。
3. 在第三次要调用服务时，下线服务提供者，可以在注册中心看到节点的注册 key 已被删除。1747226499385180161_0.9982336813944652
4. 继续向下执行，发现第三次调用服务时，又重新从注册中心查询，说明缓存已经被更新。

至此，消费端服务缓存功能已经完成。1747226499385180161_0.6694846317789871

ZooKeeper 注册中心实现

> 这部分不作为学习重点，理解了一种注册中心的实现方式，再用其他技术实现注册中心就很简单了。1747226499385180161_0.5271932865024358

其实和 Etcd 注册中心的实现方式极其相似，步骤如下：1747226499385180161_0.3399266575288775

1. 安装 ZooKeeper
2. 引入客户端依赖
3. 实现接口 1747226499385180161_0.3835074547072199
4. SPI 补充 ZooKeeper 注册中心

1）本地下载并启动 ZooKeeper，教程使用的版本是 `3.8.4` ，大家最好跟教程保持一致。

下载链接： https://dlcdn.apache.org/zookeeper/zookeeper-3.8.4/apache-zookeeper-3.8.4-bin.tar.gz 1747226499385180161_0.322126675102679

如果发现该版本不存在，换一个最接近的版本即可。

正常启动 ZooKeeper 后，默认会占用几个端口号，比如 2181（客户端）、8080（管理端）等。

2）引入客户端依赖。1747226499385180161_0.20832407710223033

一般我们会使用 Apache Curator 来操作 ZooKeeper，可以参考官方文档： https://curator.apache.org/docs/getting-started 。

引入的依赖代码如下：

```
<!-- zookeeper -->
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-x-discovery</artifactId>
    <version>5.6.0</version>
</dependency>
```

3）ZooKeeper 注册中心实现，这里不再赘述：

```
package com.yupi.yurpc.registry;

import cn.hutool.core.collection.ConcurrentHashSet;
import com.yupi.yurpc.config.RegistryConfig;
import com.yupi.yurpc.model.ServiceMetaInfo;
import lombok.extern.slf4j.Slf4j;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.CuratorFrameworkFactory;
import org.apache.curator.framework.recipes.cache.CuratorCache;
import org.apache.curator.framework.recipes.cache.CuratorCacheListener;
import org.apache.curator.retry.ExponentialBackoffRetry;
import org.apache.curator.x.discovery.ServiceDiscovery;
import org.apache.curator.x.discovery.ServiceDiscoveryBuilder;
import org.apache.curator.x.discovery.ServiceInstance;
import org.apache.curator.x.discovery.details.JsonInstanceSerializer;

import java.util.Collection;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

/**
 * zookeeper 注册中心
 * 操作文档：<a href="https://curator.apache.org/docs/getting-started">Apache Curator</a>
 * 代码示例：<a href="https://github.com/apache/curator/blob/master/curator-examples/src/main/java/discovery/DiscoveryExample.java">Discov
 * 监听 key 示例：<a href="https://github.com/apache/curator/blob/master/curator-examples/src/main/java/cache/CuratorCacheExample.java">Cu
 *
 * @author <a href="https://github.com/liyupi">coder_yupi</a>
 * @from <a href="https://yupi.icu">编程导航学习圈</a>
 * @learn <a href="https://codefather.cn">yupi 的编程宝典</a>
 */
@Slf4j
```

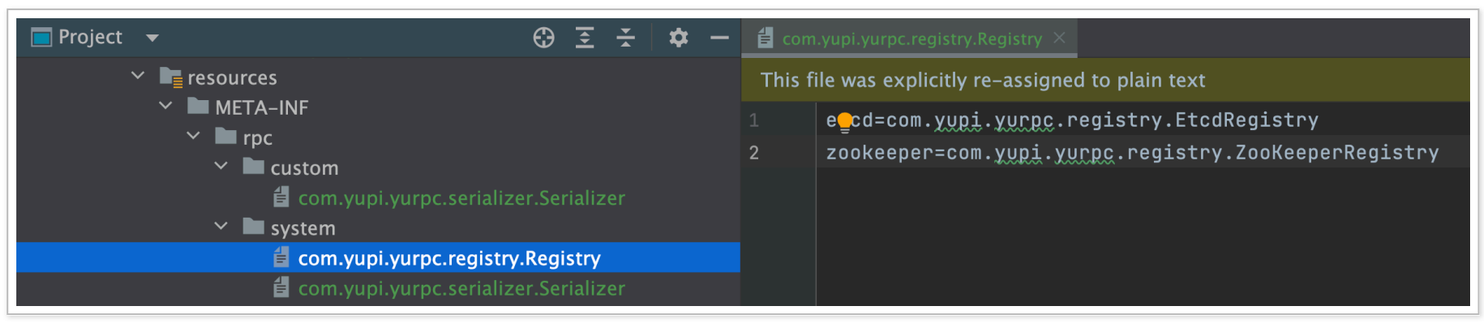
```java
public class ZooKeeperRegistry implements Registry {

    private CuratorFramework client;

    private ServiceDiscovery<ServiceMetaInfo> serviceDiscovery;

    /**
     * 本机注册的节点 key 集合（用于维护续期）
     */
    private final Set<String> localRegisterNodeKeySet = new HashSet<>();

    /**
     * 注册中心服务缓存
     */
    private final RegistryServiceCache registryServiceCache = new RegistryServiceCache();

    /**
     * 正在监听的 key 集合
     */
    private final Set<String> watchingKeySet = new ConcurrentHashSet<>();

    /**
     * 根节点
     */
    private static final String ZK_ROOT_PATH = "/rpc/zk";


    @Override
    public void init(RegistryConfig registryConfig) {
        // 构建 client 实例
        client = CuratorFrameworkFactory
                .builder()
                .connectString(registryConfig.getAddress())
                .retryPolicy(new ExponentialBackoffRetry(Math.toIntExact(registryConfig.getTimeout()), 3))
                .build();

        // 构建 serviceDiscovery 实例
        serviceDiscovery = ServiceDiscoveryBuilder.builder(ServiceMetaInfo.class)
                .client(client)
                .basePath(ZK_ROOT_PATH)
                .serializer(new JsonInstanceSerializer<>(ServiceMetaInfo.class))
                .build();

        try {
            // 启动 client 和 serviceDiscovery
            client.start();
            serviceDiscovery.start();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public void register(ServiceMetaInfo serviceMetaInfo) throws Exception {
        // 注册到 zk 里
        serviceDiscovery.registerService(buildServiceInstance(serviceMetaInfo));

        // 添加节点信息到本地缓存
        String registerKey = ZK_ROOT_PATH + "/" + serviceMetaInfo.getServiceNodeKey();
        localRegisterNodeKeySet.add(registerKey);
    }

    @Override
    public void unRegister(ServiceMetaInfo serviceMetaInfo) {
        try {
            serviceDiscovery.unregisterService(buildServiceInstance(serviceMetaInfo));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        // 从本地缓存移除
        String registerKey = ZK_ROOT_PATH + "/" + serviceMetaInfo.getServiceNodeKey();
        localRegisterNodeKeySet.remove(registerKey);
    }

    @Override
    public List<ServiceMetaInfo> serviceDiscovery(String serviceKey) {
        // 优先从缓存获取服务
        List<ServiceMetaInfo> cachedServiceMetaInfoList = registryServiceCache.readCache();
        if (cachedServiceMetaInfoList != null) {
            return cachedServiceMetaInfoList;
        }

        try {
            // 查询服务信息
            Collection<ServiceInstance<ServiceMetaInfo>> serviceInstanceList = serviceDiscovery.queryForInstances(serviceKey);

            // 解析服务信息
            List<ServiceMetaInfo> serviceMetaInfoList = serviceInstanceList.stream()
                    .map(ServiceInstance::getPayload)
                    .collect(Collectors.toList());
            // 写入服务缓存
            registryServiceCache.writeCache(serviceMetaInfoList);
            return serviceMetaInfoList;
        } catch (Exception e) {
            throw new RuntimeException("获取服务列表失败", e);
        }
    }

    @Override
    public void heartBeat() {
        // 不需要心跳机制，建立了临时节点，如果服务器故障，则临时节点直接丢失
    }

    /**
     * 监听（消费端）
     *
     * @param serviceNodeKey 服务节点 key
     */
    @Override
    public void watch(String serviceNodeKey) {
        String watchKey = ZK_ROOT_PATH + "/" + serviceNodeKey;
        boolean newWatch = watchingKeySet.add(watchKey);
        if (newWatch) {
            CuratorCache curatorCache = CuratorCache.build(client, watchKey);
            curatorCache.start();
            curatorCache.listenable().addListener(
                    CuratorCacheListener
```

```
                        .builder()
                        .forDeletes(childData -> registryServiceCache.clearCache())
                        .forChanges(((oldNode, node) -> registryServiceCache.clearCache()))
                        .build()
        );
    }
}

@Override
public void destroy() {
    log.info("当前节点下线");
    // 下线节点（这一步可以不做，因为都是临时节点，服务下线，自然就被删掉了）
    for (String key : localRegisterNodeKeySet) {
        try {
            client.delete().guaranteed().forPath(key);
        } catch (Exception e) {
            throw new RuntimeException(key + "节点下线失败");
        }
    }

    // 释放资源
    if (client != null) {
        client.close();
    }
}

private ServiceInstance<ServiceMetaInfo> buildServiceInstance(ServiceMetaInfo serviceMetaInfo) {
    String serviceAddress = serviceMetaInfo.getServiceHost() + ":" + serviceMetaInfo.getServicePort();
    try {
        return ServiceInstance
                .<ServiceMetaInfo>builder()
                .id(serviceAddress)
                .name(serviceMetaInfo.getServiceKey())
                .address(serviceAddress)
                .payload(serviceMetaInfo)
                .build();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
  }
}
```

4）SPI 增加对 ZooKeeper 的支持：1747226499385180161_0.3381496899284515

```
etcd=com.yupi.yurpc.registry.EtcdRegistry
zookeeper=com.yupi.yurpc.registry.ZooKeeperRegistry
```

如图：



5）最后，可以更改服务提供者和消费者的注册中心配置来测试。

更改的配置如下：

```
rpc.registryConfig.registry=zookeeper
rpc.registryConfig.address=localhost:2181
```

三、扩展

1）完善服务注册信息。

参考思路：比如增加节点注册时间。1747226499385180161_0.52550243086286

2）实现更多注册中心。（较难）

参考思路：使用 Redis 实现注册中心。

3）保证注册中心的高可用。1747226499385180161_0.9326781327040405

参考思路：了解 Etcd 的集群机制。

4）服务注册信息失效的兜底策略。（较难）

参考思路：如果消费端调用节点时发现节点失效，也可以考虑是否需要从注册中心更新服务注册信息、或者强制更新本地缓存。
1747226499385180161_0.341801673066199

5）注册中心 key 监听时，采用观察者模式实现处理。

参考思路：可以定义一个 Listener 接口，根据 watch key 的变更类型去调用 Listener 的不同方法。

1747226499385180161_0.409986817463339