<http://www.360doc.com/content/18/0511/08/36490684_753005092.shtml>

解决kryo问题的方案:

https://github.com/magro/kryo-serializers

**【RPC 专栏】深入理解RPC之序列化篇 —— Kryo**

2018-05-11  [太极混元...](http://www.360doc.com/userhome/36490684)   阅 435  转 1

[转藏到我的图书馆](javascript:void(0);)

[微信](javascript:void(0);)分享：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 一年前，笔者刚刚接触RPC框架，从单体式应用向分布式应用的变革无疑是让人兴奋的，同时也对RPC背后到底做了哪些工作产生了兴趣，但其底层的设计对新手而言并不是很友好，其涉及的一些常用技术点都有一定的门槛。如传输层常常使用的netty，之前完全没听过，想要学习它，需要掌握前置知识点nio；协议层，包括了很多自定义的协议，而每个RPC框架的实现都有差异；代理层的动态代理技术，如jdk动态代理，虽然实战经验不多，但至少还算会用，而cglib则又有一个盲区；序列化层倒还算是众多层次中相对简单的一环，但RPC为了追求可扩展性，性能等诸多因素，通常会支持多种序列化方式以供使用者插拔使用，一些常用的序列化方案hessian，kryo，Protobuf又得熟知...  这个系列打算就RPC框架涉及到的一些知识点进行探讨，本篇先从序列化层的一种选择--kryo开始进行介绍。  **序列化概述**  大白话介绍下RPC中序列化的概念，可以简单理解为对象-->字节的过程，同理，反序列化则是相反的过程。为什么需要序列化？因为网络传输只认字节。所以互信的过程依赖于序列化。有人会问，FastJson转换成字符串算不算序列化？对象持久化到数据库算不算序列化？没必要较真，广义上理解即可。  **JDK序列化**  可能你没用过kryo，没用过hessian，但你一定用过jdk序列化。我最早接触jdk序列化，是在大二的JAVA大作业中，《XX管理系统》需要把对象保存到文件中（那时还没学数据库），jdk原生支持的序列化方式用起来也很方便。   1. class Student implements Serializable{ 2. private String name; 3. } 4. class Main{ 5. public static void main(String[] args) throws Exception{ 6. // create a Student 7. Student st = new Student('kirito'); 8. // serialize the st to student.db file 9. ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream('student.db')); 10. oos.writeObject(st); 11. oos.close(); 12. // deserialize the object from student.db 13. ObjectInputStream ois = new ObjectInputStream(new FileInputStream('student.db')); 14. Student kirito = (Student) ois.readObject(); 15. ois.close(); 16. // assert 17. assert 'kirito'.equals(kirito.getName()); 18. } 19. }   Student实体类需要实现Serializable接口，以告知其可被序列化。  序列化协议的选择通常有下列一些常用的指标：   1. 通用性。是否只能用于java间序列化/反序列化，是否跨语言，跨平台。 2. 性能。分为空间开销和时间开销。序列化后的数据一般用于存储或网络传输，其大小是很重要的一个参数；解析的时间也影响了序列化协议的选择，如今的系统都在追求极致的性能。 3. 可扩展性。系统升级不可避免，某一实体的属性变更，会不会导致反序列化异常，也应该纳入序列化协议的考量范围。 4. 易用性。API使用是否复杂，会影响开发效率。   容易用的模型通常性能不好，性能好的模型通常用起来都比较麻烦。显然，JDK序列化属于前者。我们不过多介绍它，直接引入今天的主角kryo作为它的替代品。  **Kryo入门**  **引入依赖**    2. com.esotericsoftware 3. kryo 4. 4.0.1   由于其底层依赖于ASM技术，与Spring等框架可能会发生ASM依赖的版本冲突（文档中表示这个冲突还挺容易出现）所以提供了另外一个依赖以供解决此问题    2. com.esotericsoftware 3. kryo-shaded 4. 4.0.1   **快速入门**   1. class Student implements Serializable{ 2. private String name; 3. } 4. public class Main { 5. public static void main(String[] args) throws Exception{ 6. Kryo kryo = new Kryo(); 7. Output output = new Output(new FileOutputStream('student.db')); 8. Student kirito = new Student('kirito'); 9. kryo.writeObject(output, kirito); 10. output.close(); 11. Input input = new Input(new FileInputStream('student.db')); 12. Student kiritoBak = kryo.readObject(input, Student.class); 13. input.close(); 14. assert 'kirito'.equals(kiritoBak.getName()); 15. } 16. }   不需要注释也能理解它的执行流程，和jdk序列化差距并不是很大。  **三种读写方式**  Kryo共支持三种读写方式   1. 如果知道class字节码，并且对象不为空 2. kryo.writeObject(output, someObject); 3. // ... 4. SomeClass someObject = kryo.readObject(input, SomeClass.class);   快速入门中的序列化/反序列化的方式便是这一种。而Kryo考虑到someObject可能为null，也会导致返回的结果为null，所以提供了第二套读写方式。   1. 如果知道class字节码，并且对象可能为空 2. kryo.writeObjectOrNull(output, someObject); 3. // ... 4. SomeClass someObject = kryo.readObjectOrNull(input, SomeClass.class);   但这两种方法似乎都不能满足我们的需求，在RPC调用中，序列化和反序列化分布在不同的端点，对象的类型确定，我们不想依赖于手动指定参数，最好是...emmmmm...将字节码的信息直接存放到序列化结果中，在反序列化时自行读取字节码信息。Kryo考虑到了这一点，于是提供了第三种方式。   1. 如果实现类的字节码未知，并且对象可能为null 2. kryo.writeClassAndObject(output, object); 3. // ... 4. Object object = kryo.readClassAndObject(input); 5. if (object instanceof SomeClass) { 6. // ... 7. }   我们牺牲了一些空间一些性能去存放字节码信息，但这种方式是我们在RPC中应当使用的方式。  **我们关心的问题**  继续介绍Kryo特性之前，不妨让我们先思考一下，一个序列化工具或者一个序列化协议，应当需要考虑哪些问题。比如，支持哪些类型的序列化？循环引用会不会出现问题？在某个类增删字段之后反序列化会报错吗？等等等等....  带着我们考虑到的这些疑惑，以及我们暂时没考虑到的，但Kryo帮我们考虑到的，来看看Kryo到底支持哪些特性。  **支持的序列化类型**   | **boolean** | **Boolean** | **byte** | **Byte** | **char** | | --- | --- | --- | --- | --- | | Character | short | Short | int | Integer | | long | Long | float | Float | double | | Double | byte[] | String | BigInteger | BigDecimal | | Collection | Date | Collections.emptyList | Collections.singleton | Map | | StringBuilder | TreeMap | Collections.emptyMap | Collections.emptySet | KryoSerializable | | StringBuffer | Class | Collections.singletonList | Collections.singletonMap | Currency | | Calendar | TimeZone | Enum | EnumSet |  |   表格中支持的类型一览无余，这都是其默认支持的。   1. Kryo kryo = new Kryo(); 2. kryo.addDefaultSerializer(SomeClass.class, SomeSerializer.class);   这样的方式，也可以为一个Kryo实例扩展序列化器。  总体而言，Kryo支持以下的类型：   * 枚举 * 集合、数组 * 子类/多态 * 循环引用 * 内部类 * 泛型   但需要注意的是，**Kryo不支持Bean中增删字段**。如果使用Kryo序列化了一个类，存入了Redis，对类进行了修改，会导致反序列化的异常。  另外需要注意的一点是使用反射创建的一些类序列化的支持。如使用Arrays.asList();创建的List对象，会引起序列化异常。   1. Exception in thread 'main' com.esotericsoftware.kryo.KryoException: Class cannot be created (missing no-arg constructor): java.util.Arrays$ArrayList   但new ArrayList()创建的List对象则不会，使用时需要注意，可以使用第三方库对Kryo进行序列化类型的扩展。如https://github.com/magro/kryo-serializers所提供的。  **不支持不包含无参构造器类的反序列化**，尝试反序列化一个不包含无参构造器的类将会得到以下的异常：   1. Exception in thread 'main' com.esotericsoftware.kryo.KryoException: Class cannot be created (missing no-arg constructor): moe.cnkirito.Xxx   保证每个类具有无参构造器是应当遵守的编程规范，但实际开发中一些第三库的相关类不包含无参构造，的确是有点麻烦。  **线程安全**  Kryo是线程不安全的，意味着每当需要序列化和反序列化时都需要实例化一次，或者借助ThreadLocal来维护以保证其线程安全。   1. private static final ThreadLocalKryo> kryos = new ThreadLocalKryo>() { 2. protected Kryo initialValue() { 3. Kryo kryo = new Kryo(); 4. // configure kryo instance, customize settings 5. return kryo; 6. }; 7. }; 8. // Somewhere else, use Kryo 9. Kryo k = kryos.get(); 10. ...   **Kryo相关配置参数详解**  每个Kryo实例都可以拥有两个配置参数，这值得被拉出来单独聊一聊。   1. kryo.setRegistrationRequired(false);//关闭注册行为 2. kryo.setReferences(true);//支持循环引用   Kryo支持对注册行为，如 kryo.register(SomeClazz.class);,这会赋予该Class一个从0开始的编号，但Kryo使用注册行为最大的问题在于，其不保证同一个Class每一次注册的号码想用，这与注册的顺序有关，也就意味着在不同的机器、同一个机器重启前后都有可能拥有不同的编号，这会导致序列化产生问题，所以在分布式项目中，一般关闭注册行为。  第二个注意点在于循环引用，Kryo为了追求高性能，可以关闭循环引用的支持。不过我并不认为关闭它是一件好的选择，大多数情况下，请保持 kryo.setReferences(true)。  **常用Kryo工具类**   1. public class KryoSerializer { 2. public byte[] serialize(Object obj) { 3. Kryo kryo = kryoLocal.get(); 4. ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream(); 5. Output output = new Output(byteArrayOutputStream);//<1> 6. kryo.writeClassAndObject(output, obj);//<2> 7. output.close(); 8. return byteArrayOutputStream.toByteArray(); 9. } 10. public T> T deserialize(byte[] bytes) { 11. Kryo kryo = kryoLocal.get(); 12. ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(bytes); 13. Input input = new Input(byteArrayInputStream);// <1> 14. input.close(); 15. return (T) kryo.readClassAndObject(input);//<2> 16. } 17. private static final ThreadLocalKryo> kryoLocal = new ThreadLocalKryo>() {//<3> 18. @Override 19. protected Kryo initialValue() { 20. Kryo kryo = new Kryo(); 21. kryo.setReferences(true);//默认值为true,强调作用 22. kryo.setRegistrationRequired(false);//默认值为false,强调作用 23. return kryo; 24. } 25. }; 26. }   <1> Kryo的Input和Output接收一个InputStream和OutputStream，Kryo通常完成字节数组和对象的转换，所以常用的输入输出流实现为ByteArrayInputStream/ByteArrayOutputStream。  <2> writeClassAndObject和readClassAndObject配对使用在分布式场景下是最常见的，序列化时将字节码存入序列化结果中，便可以在反序列化时不必要传入字节码信息。  <3> 使用ThreadLocal维护Kryo实例，这样减少了每次使用都实例化一次Kryo的开销又可以保证其线程安全。 |