**Kryo 使用指南**

* Author: hanwenyang@jd.com
* Date: 2017-05-26
* Version: 1.1

**1、Kryo 的简介**

Kryo 是一个快速序列化/反序列化工具，其使用了字节码生成机制（底层依赖了 ASM 库），因此具有比较好的运行速度。

Kryo 序列化出来的结果，是其自定义的、独有的一种格式，不再是 JSON 或者其他现有的通用格式；而且，其序列化出来的结果是二进制的（即 byte[]；而 JSON 本质上是字符串 String）；二进制数据显然体积更小，序列化、反序列化时的速度也更快。

Kryo 一般只用来进行序列化（然后作为缓存，或者落地到存储设备之中）、反序列化，而不用于在多个系统、甚至多种语言间进行数据交换 —— 目前 kryo 也只有 java 实现。

像 Redis 这样的存储工具，是可以安全地存储二进制数据的，所以可以直接把 Kryo 序列化出来的数据存进去。

当然，如果你希望用 String 的形式存储、传输 Kryo 序列化之后的数据，也可以通过 Base64 等编码方式来实现。但这会降低程序的运行速度，一定程度上违背了使用 kryo 的初衷。

Kryo 在使用时，需要根据使用场景进行一定的设置；如果设置不当，会导致一些严重的错误。（这些问题的原因参见第 2 节）

附件中提供了我们部门封装的 KryoUtil ，其根据分布式 Web 应用的一般场景，进行了配置及封装；可以在自己的项目里安全地使用此工具类。

**2、Kryo 的特点和配置的选择**

**2.1 支持的范围**

除了常见的 JDK 类型、以及这些类型组合而来的普通 POJO，Kryo 还支持以下特殊情况：

* 枚举；
* 任意 Collention、数组；
* 子类/多态（详见 **2.2** 节）；
* 循环引用（详见 **2.5** 节）；
* 内部类（详见 **2.6** 节）；
* 泛型对象（详见 **3.2** 节）；
* Builder 模式；

其中部分特性的支持，需要使用者手动设定 Kryo 的某些配置（KryoUtil 已经进行了这些配置）。

Kryo 不支持以下情况：

* 增加或删除 Bean 中的字段；

举例来说，某一个 Bean 使用 Kryo 序列化后，结果被放到 Redis 里做了缓存，如果某次上线增加/删除了这个 Bean 中的一个字段，则缓存中的数据进行反序列化时会报错；作为缓存功能的开发者，此时应该 catch 住异常，清除这条缓存，然后返回 “缓存未命中” 信息给上层调用者。

字段顺序的变化不会导致反序列化失败。

**2.2 记录类型/对多态的支持**

Kryo 的一大特点是，支持把对象的类型信息，也放进序列化的结果里。

举例来说：假设我们有一个自己定义的接口 WeightList<T>，有两个实现：ArrayWeightList<T> 和 LinkedWeightList<T>；一般的 JSON 序列化工具，在默认情况下无法记录我们使用的是哪一个实现类；如果不进行特殊的配置，JSON 序列化工具在进行反序列化时会报错。

而 Kryo 将原始对象的类型信息，记录到了序列化的结果里；所以反序列的时候可以精确地找到原始的类型，不会报错。

同时，在反序列化任意对象时，也不再需要再提供 Class 信息或者 Type 信息了，代码也更为简洁、通用。（可以参考第 5 节中的例子）

如果选择记录类型信息，则使用 kryo 中的 writeClassAndObject/readClassAndObject 方法，如果选择不记录类型信息（反序列化时由调用方提供类型信息），则使用 writeObject/readObject 方法。

**2.3 线程安全**

Kryo 对象不是线程安全的，所以需要借用 ThreadLocal 来保证线程安全性。具体实现可以参考附件中的 KryoUtil。

如果对性能有更高要求，也可以使用 KryoPool：<https://github.com/EsotericSoftware/kryo#threading>

**2.4 注册行为**

Kryo 支持对类进行注册。注册行为会给每一个 Class 编一个号码，从 0 开始；但是，Kryo 并不保证同一个 Class 每一次的注册的号码都相同（比如重启 JVM 后，用户访问资源的顺序不同，就会导致类注册的先后顺序不同）。

也就是说，同样的代码、同一个 Class ，在两台机器上的注册编号可能不一致；那么，一台机器序列化之后的结果，可能就无法在另一台机器上反序列化。

因此，对于多机器部署的情况，建议关闭注册，让 Kryo 记录每个类型的真实的名称。

而且，注册行为需要用户对每一个类进行手动注册：即便使用者注册了 A 类型，而 A 类型内部使用了 B 类型，使用者也必须手动注册 B 类型；（甚至，即便某一个类型是 JDK 内部的类型，比如 ArrayList ，也是需要手动注册的）一个普通的业务对象，往往需要注册十几个 Class，这是十分麻烦、甚至是寸步难行的。

关闭注册行为，需要保证**没有**进行过这样的设置：

kryo.setRegistrationRequired(true);

并且要保证**没有**显式地注册任何一个类，例如：

kryo.register(ArrayList.class);

同时保证以上二者，才真正地关闭了注册行为。

**2.5 对循环引用的支持**

举例而言，“循环引用” 是指，假设有一个 “账单” 的 Bean（比如：BillDomain），这个账单下面有很多明细（比如：private List<ItemDomain> items;），而明细类中又有一个字段引用了所属的账单（比如：private BillDomain superior;），那么这就构成了“循环引用”。

Kryo 是支持循环引用的，只需要保证**没有**进行过这样的设置就可以了：

kryo.setReferences(false);

配置成 false 的话，序列化速度更快，但是遇到循环引用，就会报 “栈内存溢出” 错误。这有很大的风险：等你不得不支持循环引用的那一天你就会发现，你必须在代码上线的同时，清除 Redis 里已有的大量缓存（详见 **2.8** 节）。

**2.6 内部类**

Kryo 支持静态内部类，既可以是私有/包级私有的，也可以是 public 的；但是对非静态内部类的支持不够好（一般不会报错，但在有些情况下会产生错误的数据），这和不同的编译器对内部类的处理有关（可参阅 Java 内部类的语法糖机制）。同样地，Kryo 支持 Builder 模式。

Kryo **不支持匿名类**，反序列化时往往会产生错误的数据（这比报错更加危险），请尽量不要使用匿名类传递数据。

**2.7 序列化格式**

Kryo 实际上支持任意的序列化格式，并不一定使用 Kryo 自己定义的那种特殊的格式（甚至可以为不同的 class 指定不同的序列化格式），比如使用 Java 语言自己的序列化格式（在 Kryo 中注册 JavaSerializer 即可） —— 但我们强烈建议不要这么使用，Java 语言本身的序列化方式有很多限制，比如必须要保证每一个 Bean 都实现 Serializable 接口；而系统中可能有很多 Bean 都忘了实现这个接口；这些类在编译时并不会报错，只有在运行期间、进行序列化时才会报错，这是危险的。

Kryo 默认的序列化格式没有任何限制，显然方便的多。

**2.8 配置的修改**

Kryo 可以通过修改配置来达到更快的速度，或者支持更多的特殊形式；但是必须注意的是，一旦改变某一个个配置，**序列化出来的格式和之前的格式是完全不一样的**； 也就是说，你必须在上线代码的同时，清除 Redis 里所有已有的缓存，否则那些缓存里的数据再回来进行反序列化的时候，就会报错。

**3、常见问题**

**3.1 使用的时候报 asm 相关类的错误**

Kryo 底层用了 asm 库（一个字节码生成库），Spring 底层也用了这个库 ；但是，Kryo 使用的版本比较高；而 Spring 用的版本较低； 如果 pom 里的 Kryo 和 Spring 的顺序不对的话，Kryo 就会读到低版本的 asm，就会出错。

请检查对 Kryo 的 Maven 依赖，如果 artifactId 是这样的：

<artifactId>kryo</artifactId>：

就改为：

<artifactId>kryo-shaded</artifactId>

加了个 shaded 就能解决了。在这个 shaded 的版本里，Kryo 的作者复制了一份高版本的 asm，集成到了 Kryo 内部（作者修改了 asm 类的包名，所以和原来的 asm 就不会再冲突了）。

**3.2 泛型对象的反序列化**

在使用常见的 JSON 库时，泛型对象不能使用 \*.class 进行反序列化；比如 Gson 在反序列化 List<SomeDomain> 的时候，除了传入 JSON 字符串，还需要传入第二个参数：

new TypeToken<List<SomeDomain>>(){}.getType()

直接使用 List.class 是不行的（而 List<SomeDomain>.class 则是语法错误），这是 Java 泛型的 “擦除” 机制导致的。

而 Kryo 的 readObject 方法则没有这个问题。在上例中，向 Kryo 的 readObject 方法传入 List.class 即可；Kryo 实际上在序列化结果里记录了泛型参数的实际类型的信息，反序列化时会根据这些信息来实例化对象。

直觉上我们会觉得，不在序列化结果中包含类型信息，能减小空间的占用、提高速度；但实际上，我们发现，所谓的 “不包含类型信息”，在 Kryo 内部的实现里，仅仅是 “不包含最外层对象的类型信息” ，对象内部的子对象的类型信息依然是包含的（可能是为了支持多态问题）；也就是说，“不包含类型信息” 能带来的空间节省非常有限。

如果对速度、序列化之后的数据大小没有特别极端的要求，推荐在序列化结果中包含类型信息，这样的话，反序列化时能少些一个参数，也更为通用。

**4、使用 Kryo 需要添加的 Maven 依赖**

<!-- Kryo -->

<dependency>

<groupId>com.esotericsoftware</groupId>

<artifactId>kryo-shaded</artifactId>

<version>4.0.0</version>

</dependency>

如果使用 KryoUtil 的话，还需要以下依赖：

<!-- commons-codec -->

<dependency>

<groupId>commons-codec</groupId>

<artifactId>commons-codec</artifactId>

<version>1.10</version>

</dependency>

**5、KryoUtil 使用示例**

KryoUtil 是我们部门编写的工具类，其对 Kryo 进行了一定的封装，能够满足分布式系统的一般需求，而无需进行任何额外的配置。

除了用于获得当前线程的 kryo 实例的 getInstance() 方法之外，KryoUtil 内共有 8 个 public 方法，分为两组：

<T> byte[] writeToByteArray(T obj);

<T> String writeToString(T obj);

<T> T readFromByteArray(byte[] byteArray);

<T> T readFromString(String str);

及：

<T> byte[] writeObjectToByteArray(T obj)

<T> String writeObjectToString(T obj)

<T> T readObjectFromByteArray(byte[] byteArray, Class<T> clazz)

<T> T readObjectFromString(String str, Class<T> clazz)

其中第一组序列化的结果里包含了类型信息，第二组不包含 —— 因此，可以看到，在使用第二组方法进行反序列化的时候，需要提供原始对象的 Class 。但我们建议使用第一组方法，原因见第 3.2 节。

另外，必须注意，第一组方法和第二组方法不能混用，第一组序列化出来的结果，只能由第一组的方法进行反序列化；第二组亦然。

每组方法内，序列化的结果格式都可以选择二进制格式，或者字符串格式。具体的使用示例代码如下：

将任意对象序列化成 byte[]：

byte[] tempByteArray = KryoUtil.writeToByteArray(domainA);

//tempByteArray 就是序列化的结果，直接放到 Redis 里面即可

DomainA domainA1 = KryoUtil.readFromByteArray(tempByteArray);

//domainA1 就是反序列化之后的对象

如果你们的存储服务不支持二进制数据（或者说不是 “二进制安全” 的），那么也可以序列化成 String：

String tempStr = KryoUtil.writeToString(domainA);

//tempStr 就是序列化的结果

DomainA domainA1 = KryoUtil.readFromString(tempStr);

//domainA1 就是反序列化之后的对象

**附件：**

