序列化

1. 预备知识
   1. 什么是Java Mark Interface（标记接口）

The tag/ marker interface pattern is a design pattern in computer science, used with languages that provide run-time type information about objects. It provides a means to associate metadata with a class where the language does not have explicit support for such metadata.

* + 1. 一个标记接口，Serializable

下面的代码是我从JDK源代码中摘出来的：

if (obj instanceof String) {

writeString((String) obj, unshared);

} else if (cl.isArray()) {

writeArray(obj, desc, unshared);

} else if (obj instanceof Enum) {

writeEnum((Enum) obj, desc, unshared);

} else if (obj instanceof Serializable) {

writeOrdinaryObject(obj, desc, unshared);

} else {

if (extendedDebugInfo) {

throw new NotSerializableException(cl.getName() + " "

+ debugInfoStack.toString());

} else {

throw new NotSerializableException(cl.getName());

}

}

* + 1. 另一个标记接口，Cloneable

<https://en.wikipedia.org/wiki/Marker_interface_pattern>

1. Serializable接口解释

Serializability of a class is enabled by the class implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

类通过实现 java.io.Serializable 接口以启用其序列化功能。未实现此接口的类将无法使其任何状态序列化或反序列化。可序列化类的所有子类型本身都是可序列化的。序列化接口没有方法或字段，仅用于标识可序列化的语义。

To allow subtypes of non-serializable classes to be serialized, the subtype may assume responsibility for saving and restoring the state of the supertype's public, protected, and (if accessible) package fields. The subtype may assume this responsibility only if the class it extends has an accessible no-arg constructor to initialize the class's state. It is an error to declare a class Serializable if this is not the case. The error will be detected at runtime.

要允许不可序列化类的子类型序列化，可以假定该子类型负责保存和恢复超类型的公用 (public)、受保护的 (protected) 和（如果可访问）包 (package) 字段的状态。仅在子类型扩展的类有一个可访问的无参数构造方法来初始化该类的状态时，才可以假定子类型有此职责。如果不是这种情况，则声明一个类为可序列化类是错误的。该错误将在运行时检测到。

During deserialization, the fields of non-serializable classes will be initialized using the public or protected no-arg constructor of the class. A no-arg constructor must be accessible to the subclass that is serializable. The fields of serializable subclasses will be restored from the stream.

在反序列化过程中，将使用该类的公用或受保护的无参数构造方法初始化不可序列化类的字段。可序列化的子类必须能够访问无参数构造方法。可序列化子类的字段将从该流中恢复。

When traversing a graph, an object may be encountered that does not support the Serializable interface. In this case the NotSerializableException will be thrown and will identify the class of the non-serializable object.

当遍历一个图形时，可能会遇到不支持 Serializable 接口的对象。在此情况下，将抛出 NotSerializableException，并将标识不可序列化对象的类。

Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

在序列化和反序列化过程中需要特殊处理的类必须使用下列准确签名来实现特殊方法：

private void writeObject(java.io.ObjectOutputStream out)

throws IOException

private void readObject(java.io.ObjectInputStream in)

throws IOException, ClassNotFoundException;

private void readObjectNoData()

throws ObjectStreamException;

The writeObject method is responsible for writing the state of the object for its particular class so that the corresponding readObject method can restore it. The default mechanism for saving the Object's fields can be invoked by calling out.defaultWriteObject. The method does not need to concern itself with the state belonging to its superclasses or subclasses. State is saved by writing the individual fields to the ObjectOutputStream using the writeObject method or by using the methods for primitive data types supported by DataOutput.

writeObject 方法负责写入特定类的对象的状态，以便相应的 readObject 方法可以恢复它。通过调用 out.defaultWriteObject 可以调用保存 Object 的字段的默认机制。该方法本身不需要涉及属于其超类或子类的状态。通过使用 writeObject 方法或使用 DataOutput 支持的用于基本数据类型的方法将各个字段写入 ObjectOutputStream，状态可以被保存。

The readObject method is responsible for reading from the stream and restoring the classes fields. It may call in.defaultReadObject to invoke the default mechanism for restoring the object's non-static and non-transient fields. The defaultReadObject method uses information in the stream to assign the fields of the object saved in the stream with the correspondingly named fields in the current object. This handles the case when the class has evolved to add new fields. The method does not need to concern itself with the state belonging to its superclasses or subclasses. State is restored by reading data from the ObjectInputStream for the individual fields and making assignments to the appropriate fields of the object. Reading primitive data types is supported by DataInput.

readObject 方法负责从流中读取并恢复类字段。它可以调用 in.defaultReadObject 来调用默认机制，以恢复对象的非静态和非瞬态字段。defaultReadObject 方法使用流中的信息来分配流中通过当前对象中相应指定字段保存的对象的字段。这用于处理类演化后需要添加新字段的情形。该方法本身不需要涉及属于其超类或子类的状态。通过使用 writeObject 方法或使用 DataOutput 支持的用于基本数据类型的方法将各个字段写入 ObjectOutputStream，状态可以被保存。

The readObjectNoData method is responsible for initializing the state of the object for its particular class in the event that the serialization stream does not list the given class as a superclass of the object being deserialized. This may occur in cases where the receiving party uses a different version of the deserialized instance's class than the sending party, and the receiver's version extends classes that are not extended by the sender's version. This may also occur if the serialization stream has been tampered; hence, readObjectNoData is useful for initializing deserialized objects properly despite a "hostile" or incomplete source stream.

在序列化流不列出给定类作为将被反序列化对象的超类的情况下，readObjectNoData 方法负责初始化特定类的对象状态。这在接收方使用的反序列化实例类的版本不同于发送方，并且接收者版本扩展的类不是发送者版本扩展的类时发生。在序列化流已经被篡改时也将发生；因此，不管源流是“敌意的”还是不完整的，readObjectNoData 方法都可以用来正确地初始化反序列化的对象。

Serializable classes that need to designate an alternative object to be used when writing an object to the stream should implement this special method with the exact signature:

将对象写入流时需要指定要使用的替代对象的可序列化类，应使用准确的签名来实现此特殊方法：

ANY-ACCESS-MODIFIER Object writeReplace() throws ObjectStreamException;

This writeReplace method is invoked by serialization if the method exists and it would be accessible from a method defined within the class of the object being serialized. Thus, the method can have private, protected and package-private access. Subclass access to this method follows java accessibility rules.

Classes that need to designate a replacement when an instance of it is read from the stream should implement this special method with the exact signature.

此 writeReplace 方法将由序列化调用，前提是如果此方法存在，而且它可以通过被序列化对象的类中定义的一个方法访问。因此，该方法可以拥有私有 (private)、受保护的 (protected) 和包私有 (package-private) 访问。子类对此方法的访问遵循 java 访问规则。

在从流中读取类的一个实例时需要指定替代的类应使用的准确签名来实现此特殊方法。

ANY-ACCESS-MODIFIER Object readResolve() throws ObjectStreamException;

This readResolve method follows the same invocation rules and accessibility rules as writeReplace.

The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an [InvalidClassException](https://docs.oracle.com/javase/10/docs/api/java/io/InvalidClassException.html" \o "class in java.io). A serializable class can declare its own serialVersionUID explicitly by declaring a field named "serialVersionUID" that must be static, final, and of type long:

此 readResolve 方法遵循与 writeReplace 相同的调用规则和访问规则。

序列化运行时使用一个称为 serialVersionUID 的版本号与每个可序列化类相关联，该序列号在反序列化过程中用于验证序列化对象的发送者和接收者是否为该对象加载了与序列化兼容的类。如果接收者加载的该对象的类的 serialVersionUID 与对应的发送者的类的版本号不同，则反序列化将会导致 InvalidClassException。可序列化类可以通过声明名为 "serialVersionUID" 的字段（该字段必须是静态 (static)、最终 (final) 的 long 型字段）显式声明其自己的 serialVersionUID：

ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;

If a serializable class does not explicitly declare a serialVersionUID, then the serialization runtime will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is *strongly recommended* that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization. Therefore, to guarantee a consistent serialVersionUID value across different java compiler implementations, a serializable class must declare an explicit serialVersionUID value. It is also strongly advised that explicit serialVersionUID declarations use the private modifier where possible, since such declarations apply only to the immediately declaring class--serialVersionUID fields are not useful as inherited members. Array classes cannot declare an explicit serialVersionUID, so they always have the default computed value, but the requirement for matching serialVersionUID values is waived for array classes.

如果可序列化类未显式声明 serialVersionUID，则序列化运行时将基于该类的各个方面计算该类的默认 serialVersionUID 值，如“Java(TM) 对象序列化规范”中所述。不过， 强烈建议 所有可序列化类都显式声明 serialVersionUID 值，原因是计算默认的 serialVersionUID 对类的详细信息具有较高的敏感性，根据编译器实现的不同可能千差万别，这样在反序列化过程中可能会导致意外的 InvalidClassException。因此，为保证 serialVersionUID 值跨不同 java 编译器实现的一致性，序列化类必须声明一个明确的 serialVersionUID 值。还强烈建议使用 private 修饰符显示声明 serialVersionUID（如果可能），原因是这种声明仅应用于直接声明类 -- serialVersionUID 字段作为继承成员没有用处。数组类不能声明一个明确的 serialVersionUID，因此它们总是具有默认的计算值，但是数组类没有匹配 serialVersionUID 值的要求。