序列化

本章关注对象的序列化（object serialization）API，它提供了一个框架，用来将对象编码成字节流，以及从字节流编码中重新构建对象。"将一个对象编码成一个字节流"，这就称作序列化（serializing）该对象；相反的处理过程被称作反序列化（deserializing）。一旦对象被序列化后，它的编码就可以从一台正在运行的虚拟机被传递到另一台虚拟机上，或者被存储到磁盘上，供以后反序列化时用。序列化技术为远程通信提供了标准的线路级（wire-level）对象表示法，也为 JavaBeans 组件结构提供了标准的持久数据格式。

74、谨慎地实现 serializable 接口

要想使一个类的实例可被序列化，很简单，只要在它的声明中加入 implements Serializable 字样即可。正因为太简单了，所以普遍存在一个误解，任务程序员可以毫不费力地实现序列化。但是，实际情况复杂的多，虽然使一个类可被序列化的直接开销比较低，但是为了序列化而付出的长期开销往往是很大的。

(1)实现 Serializable 接口而付出的最大代价是，一旦一个类被发布了，就大大的降低了“改变这个类的实现”的灵活性。

(2)实现 Serializable 的第二个代价是，增加了出现 bug 和安全漏洞的可能性。

(3)实现 Serializable 的第三个代价是，随着类发行新的版本，相关的测试负担也增加了

简而言之，千万不要认为实现 Serializable 接口很容易，除非一个类在用了一段时间后会被抛弃，否则，实现 Serializable 接口是个很严肃的承诺，必须认真对待。

75、考虑使用自定义的序列化形式

(1) 如果没有先认真考虑默认的序列化形式是否适合，则不要贸然接受。一般来将，只有当你自行设计的自定义序列化形式与默认的序列化形式基本相同时，才能接受默认序列化形式。

(2) 如果一个对象的物理表示法等同于它的逻辑内容，可能就适合使用默认的序列化形式。

(3) 及时你确定了默认的序列化形式是合适的，通常必须提供一个 readObject 方法以保障约束关系和安全性。

(4) 当一个对象的物理表示法与它的逻辑数据内容有实质性区别的时候，使用默认的序列化形式存在以下缺点

a.它使这个类的导出 API 永远束缚在该类的内部表示法上

b.它会消耗过多空间

c.它会消耗过多时间

d.它会引起栈溢出（在一个对象的物理表示法使用链表，需要对链表进行遍历的时候）

76、保护性地编写 readObject 方法

(1) 对于序列化形成的字节流，并不都是安全的，里面可能有伪造的有害数据，对它们不加分辨地反序列化，可能会导致程序收到损害。伪造的有害数据一方面可以使不正确的字节流 ；另一方面还可能是在正确的字节流中夹带的“私货”，通过“私货”可以恶意修改反序列化的对象。

(2) 编写 readObject 方法的建议：

a. 对于对象引用域必须保持为私有的类，要保护性地拷贝这些域中的每一个对象。不可变类的可变组件就属于这一类

b. 对于任何约束条件，如果检查失败，则抛出 InvalidObjectException 异常。这些检查动作应该跟在所有的保护性拷贝之后

c. 如果整个对象图在被反序列化之后必须进行检验，就应该使用 ObjectInputValidation 接口（查了一下，这个接口有方法 d. validateObject，就是用来检验一个有“图”的对象是否符合约束的，验证不成功就抛出 2 中提到的异常）

d. 无论是直接方式还是间接方式，都不要调用类中任何可能被覆盖的方法。

77、对于实例控制，枚举类型优于 readResolve

(1) readObject 方法实现的困难：

a. 前文中的背景知识提到了 readResolve 方法，这里再做深化：readResolve 的调用是在 readObject 之后，readResolve 方法会返回一个对象，取代 readObject 反序列化的对象。也就是说存在一种可能，在 readResolve 调用之前，readObject 调用之后，有人恶意地得到反序列化的新的对象，取得它的引用，进而破坏单例。因此需要单例的类的所有实例域都是

transient 的。

b. 对于 readObject，它的可访问性值得考虑，私有意味着这个类失去了被子类化的能力；如果它是受保护或者公有的，而这个类的子类没有覆盖 readObject 方法，反序列化会产生一个这个类（超类）的实例，可能导致 ClassCastException 异常

(2) 鉴于前面提到的诸多困难，建议使用枚举实现单例（实例控制），简单而且不会有差错。

(3) 使用枚举类型实现 Singleton 主要有以下三大优势:

a.无需再考虑可序列化的情况

b.无需再考虑通过反射调用私有构造函数的情况

c.枚举实例创建是线程安全的

78、考虑用可序列化代理代替序列化实例

(1) 序列化代理：

a. 在需要序列化的类的内部创建一个私有的静态内部类，这个静态内部类同样实现 Serializable 接口。静态内部类通过构造函数传入外围类的引用，保留外围类的逻辑状态（比如保留所有数据、约束条件），并且有 readObject 方法

b. 同样实现了 Serializable 接口的外部类需要编写方法 writeReplace，返回一个 new 出来的静态内部类（传入了自己的引用）。wrieReplace 会在序列化的时候对写入的对象进行替换，替换为这个静态内部类。

c. 当反序列化的时候，不是调用外围类，而是调用静态内部类的反序列化的方法 readResolve，返回一个使用当初保留的外部类的全部信息构造的外部类。

(2) 使用序列化代理的好处：

a. 可以像保护性拷贝方法一样阻止伪造字符流的攻击以及内部域的盗用

b. 可以不必像保护性拷贝那样不能把需要把需要拷贝的值设为 final

c. 允许反序列化实例与原始序列化实例得到不同的类

d. 无需花费很多形式

(3) 使用序列化代理的局限性：

a. 不能与可被客户扩展的类兼容：（我理解是静态内部类没有写入文档，而且不能扩展，如果客户代码新加入了域，这个静态内部类不能保存新加入域的任何信息，进而影响序列化/反序列化的能力）

b. 不能与对象图中包含循环的某些类兼容：因为不能从对象的序列化代理的 readResolve 方法中调用这个对象的方法，因为这个对象还不存在

c. 可能增加性能开销