四、数据编码与演化

应用程序总是在变化。 修改程序大多数情况下也在修改存储的数据：可能需要捕获新的字段或记录类型，或者需要以新的方式呈现已有数据。

第2章讨论的数据模型有不同的方法来应对这种变化。

- 关系数据库通常假定数据库中的所有数据都遵循一个模式：尽管可以更改该模式（通过模式迁移，即ALTER语句），但是在任何时间点都只有一个有效的模式。
- 读时模式（schema-on-read）（或 无模式（schemaless））数据库不强制执行模式，因此数据库包含了不同时间写入的新旧数据的混合体（第2章文档模型中的模式灵活性）。

当数据格式或模式发生变化时，需要代码更改（向记录添加新字段，然后应用程序代码开始读取和写入该字段）。但是，对于一个大型应用系统，代码更迭并非易事：

- 对于服务端应用程序，会灰度发布。可能需要执行滚动升级（也称分阶段发布），每次将新版本部署到少数几个节点，检查新版本是否正常运行，然后逐步在所有节点上升级新的代码。这样新版本部署无需服务暂停，从而支持更频繁的版本发布和更好的演化。
- 对于客户端应用程序，看用户心情，可能不会马上安装。

这意味着新旧版本的代码和数据，可能同时在系统内共存。 系统需要双向兼容：

- 向后兼容：新代码可以读取由旧代码写入的数据。容易，新代码的作者清楚旧代码所编写的数据格式，因此可以明确地处理这些旧数据（如果需要，只需保留旧的代码来读取旧的数据）。
- 向前兼容：旧代码可以读取由新代码写入的数据。难，它需要旧代码忽略新版本代码所做的添加。

本章，将介绍多种编码数据的格式，包括JSON、XML、Protocol Buffers、Thrift 和 Avro。讨论它们如何处理模式变化，以及如何支持新旧数据和新旧代码共存的系统。之后，还将讨论这些格式如何用于数据存储和通信场景，包括在Web服务中，表现层状态传输（Representational State Transfer，REST）和远程过程调用（remote procedure calls，RPC）以及消息传递系统，如 actors 和消息队列。

数据编码格式

程序通常使用至少两种不同的数据表示形式：

1. 在内存中，数据保存在对象、结构体、列表、数组、哈希表和树等结构中。这些数据结构针对CPU的高效访问和操作进行了优化（通常使用指针）。

2. 将数据写入文件或通过网络发送时，必须将其编码为某种自包含的字节序列（例如JSON文档）。由于指针对其他进程没有意义，所以这个字节序列表示看起来与内存中使用的数据结构不大一样。

因此，在这两种表示之间需要进行类型的转化。从内存中的表示到字节序列的转化称为编码（或序列化），相反的过程称为解码（或解析，反序列化）。

语言特定的格式

许多编程语言都内建了将内存对象编码为字节序列的支持。例如，Java有java.io.Serializable ，Ruby有Marshal，Python有pickle .

这些库很方便，但是有深层次问题：

- 与特定的编程语言绑定
- 为了恢复相同对象类型的数据，解码过程需要实例化任意的类。这经常导致一些安全问题：如果攻击者可以让应用程序解码任意的字节序列，那么它们可以实例化任意的类，这意味着，它们可以远程执行任意代码。
- 这些库主要目标是快速简单地编码数据，多版本数据是次要的，所以经常忽略向前向后兼容性等问题
- 效率不高（编码或解码花费的CPU时间，以及编码结构的大小）

只适合临时使用。

JSON，XML 和二进制变体

由不同编程语言编写和读取的标准化编码，显然JSON和XML是其中的佼佼者。XML 过于冗长和不必要的复杂。JSON 受欢迎主要是由于它在Web浏览器中内置支持（因为是JavaScript的一个子集）以及相对于XML的简单性。CSV是另一种流行的与语言无关的格式，尽管功能较弱。

JSON，XML和CSV都是文本格式，具有不错的可读性。 除了语法问题外，还有问题：

- 数值编码有歧义：XML 和 CSV 不能区分数字和由数字组成的字符串。JSON区分字符串和数字，但不区分整数和浮点数，并且不指定精度。

  处理大数值困难。大于 2^{53} 的整数在IEEE754双精度浮点数中不能精确表示，所以这些数字在使用浮点数（如JavaScript）的语言中进行分析时，会变得不准确。Twitter有一个大于　的例子，它使用一个64位的数字来标识每条推文。Twitter的API返回的JSON包含两次推特ID，一次是JSON数字，一次是十进制字符串，已解决JavaScript应用程序没有正确解析数字的问题。
- JSON 和 XML 对 unicode（人类可读的文本）有很好的支持，但是不支持二进制字符串（没有字符编码的字节序列）。二进制字符串是一个有用的功能，人们通过 Base64 将二进制数据编码为文本来绕过这个限制。然后，模式可以表明该值应该被解释为 Base64编码。虽然可行，但是有点混乱且数据大小增加了33%。

- XML 和 JSON 都有可选的模式支持。这些模式语言相当强大，学习和使用起来也比较复杂。XML模式使用相当广泛，但许多基于JSON的工具并不局限于使用模式。由于数据（例如数字和二进制字符串）的正确解释取决于模式中的信息，因此不适用 XML/JSON架构的应用程序可能不得不硬编码适当的编码/解码逻辑。
- CSV 没有模式，行列的含义完全由应用程序指定。如果应用程序添加新的行或列，则必须手动处理该更改。格式模糊。

虽然存在这样那样的缺陷，但JSON、XML和CSV已经可用于很多应用。特别是作为数据交换格式（将数据从一个组织发送到另一个组织），它们非常受欢迎。这些情况下，只要人们就格式本身达成一致，格式美观或者高效往往不太重要。让不同的组织达成格式一致的难度通常超过了所有其他问题。

二进制编码

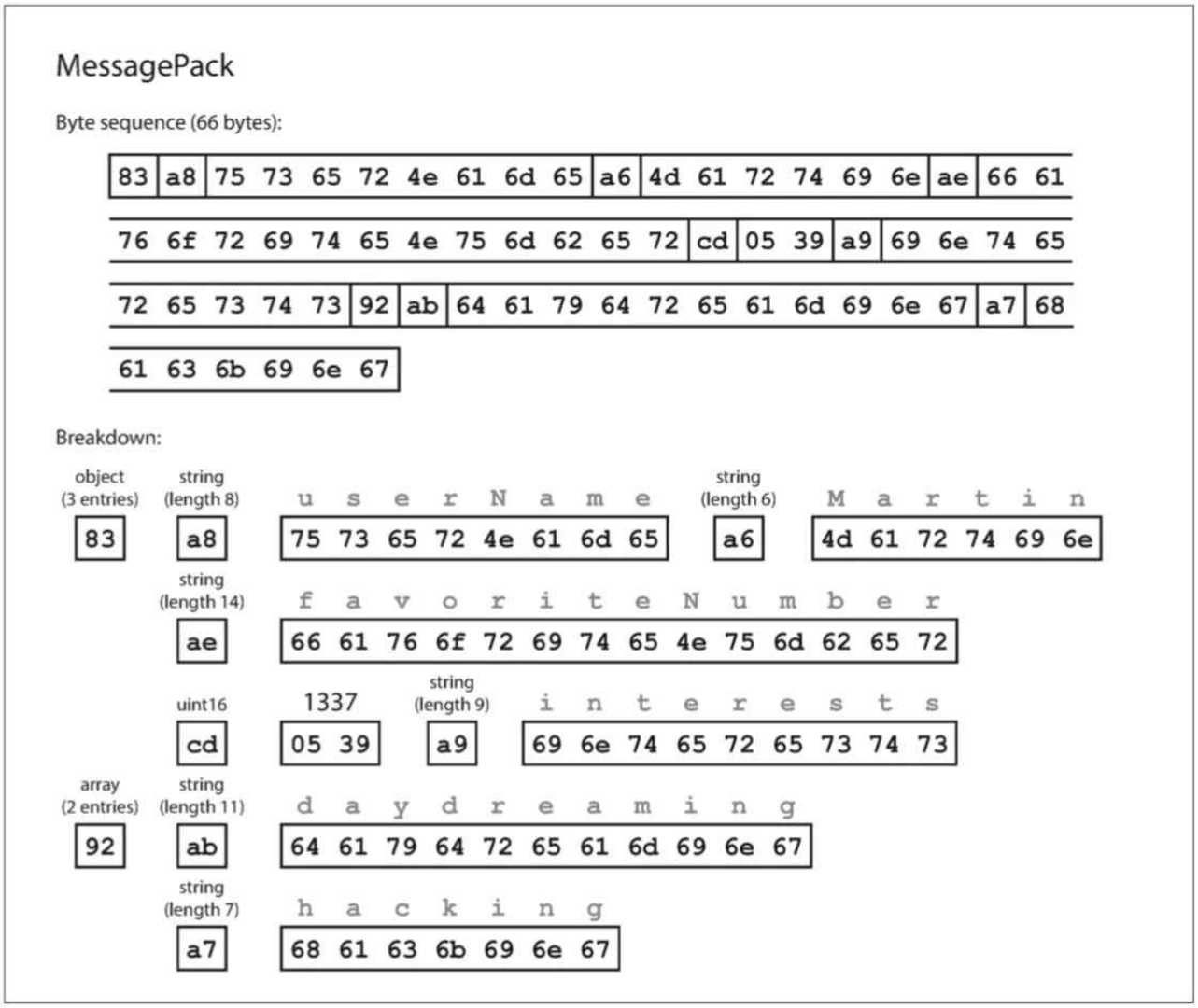
当数据很多的时候，数据格式的选择会有很大影响。 JSON比XML简洁，但与二进制格式相比还是太占空间。现在有很多二进制格式的 JSON（MessagePack，BSON，BJSON，UBJSON，BISON和Smile）和XML（WBXML和Fast Infoset）。这些格式在很多细分领域使用，但是没有一个像JSON和XML那样被广泛采用。

JSON 字符串是：

```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

MessagePack 编码的 Json 举例：

二进制编码长度为66个字节，仅略小于文本JSON编码所取的81个字节（删除了空白)。



Thrift 与 Protocol Buffers

- Protocol Buffers最初是在Google开发的，Thrift最初是在Facebook开发的，并且在2007~2008年都是开源的，都是二进制编码库。
- Thrift和Protocol Buffers都需要一个模式来编码任何数据。

接口定义语言（IDL） 来描述模式。

- Thrift 比如：

```
struct Person {
    1: required string       userName,
    2: optional i64          favoriteNumber,
    3: optional list<string> interests
}
```

- Protocol Buffers的等效模式定义看起来非常相似：

```
message Person {
    required string user_name       = 1;
    optional int64  favorite_number = 2;
    repeated string interests       = 3;
}
```

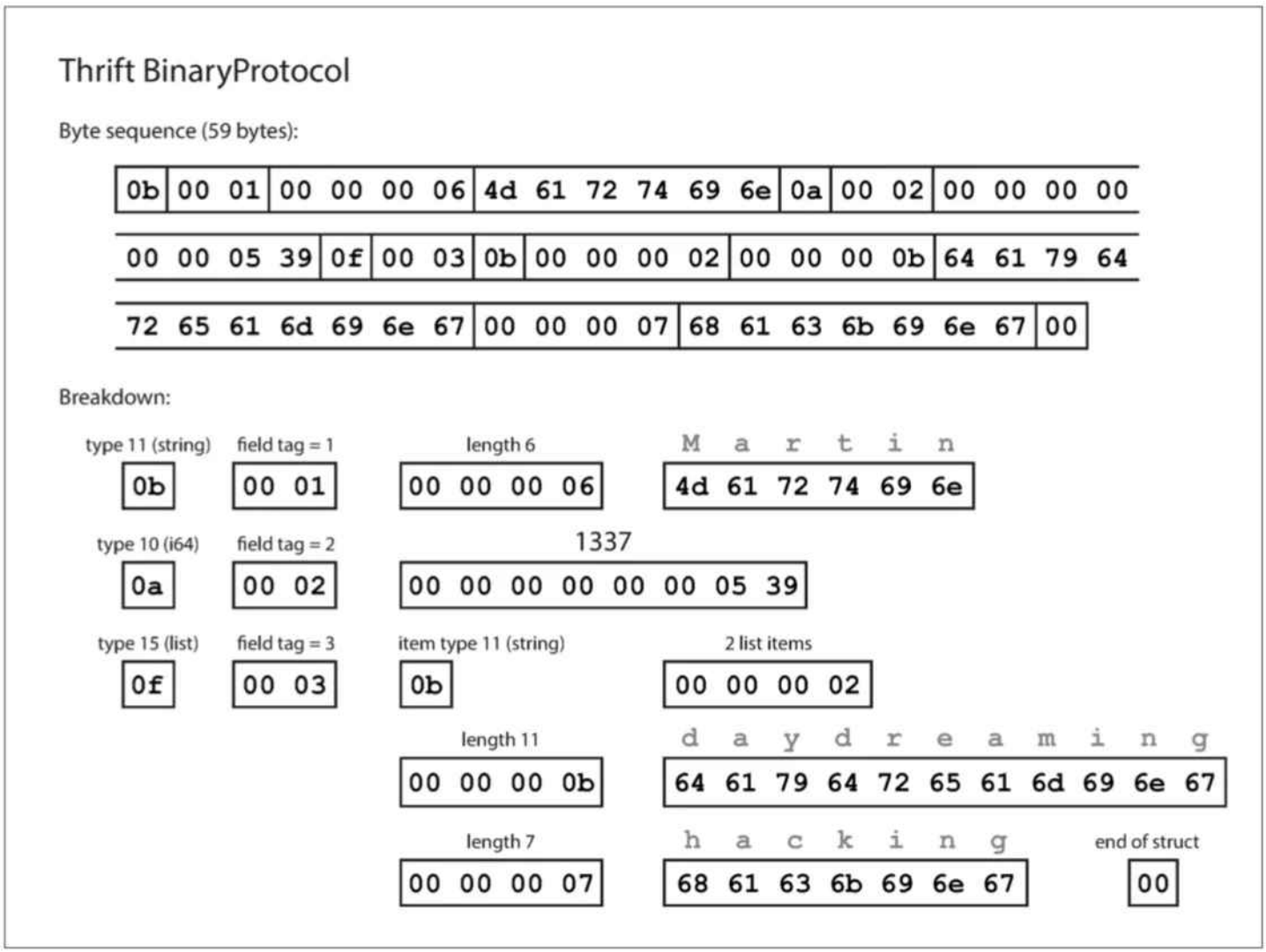
- Thrift和Protocol Buffers每一个都带有一个代码生成工具，可以调用此代码对模式进行编码和解码。

Thrift 编码格式

- Thrift 有两种不同的二进制编码格式，分别称为 BinaryProtocol 和 CompactProtocol

BinaryProtocol

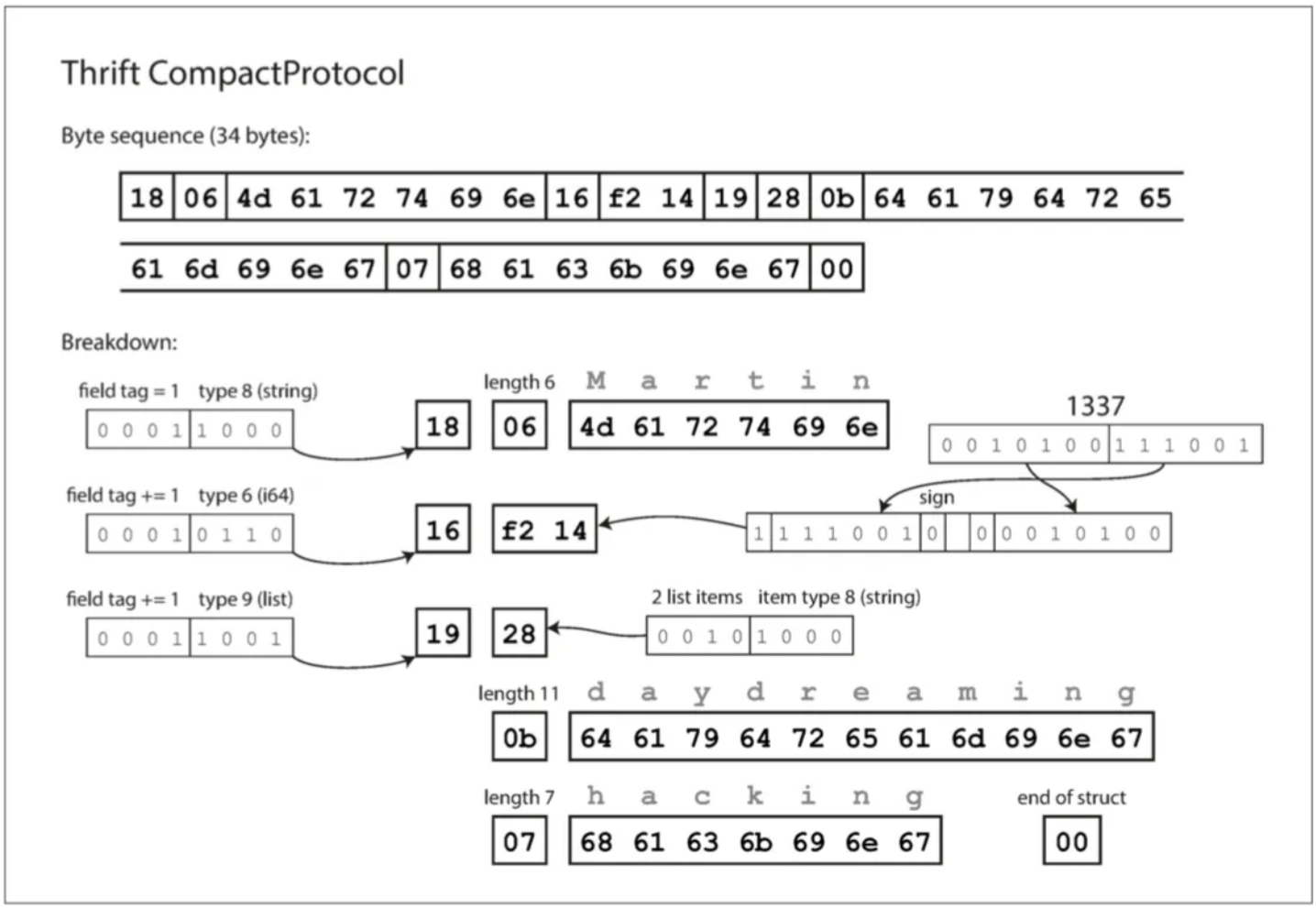
对上面的信息编码只需要59个字节



- 每个字段都有一个类型注释（用于指示它是一个字符串，整数，列表等），还可以根据需要指定长度（字符串的长度，列表中的项目数）。
- 最大的区别是没有字段名，而只有字段标签，即数字 1，2，3，就像别名。

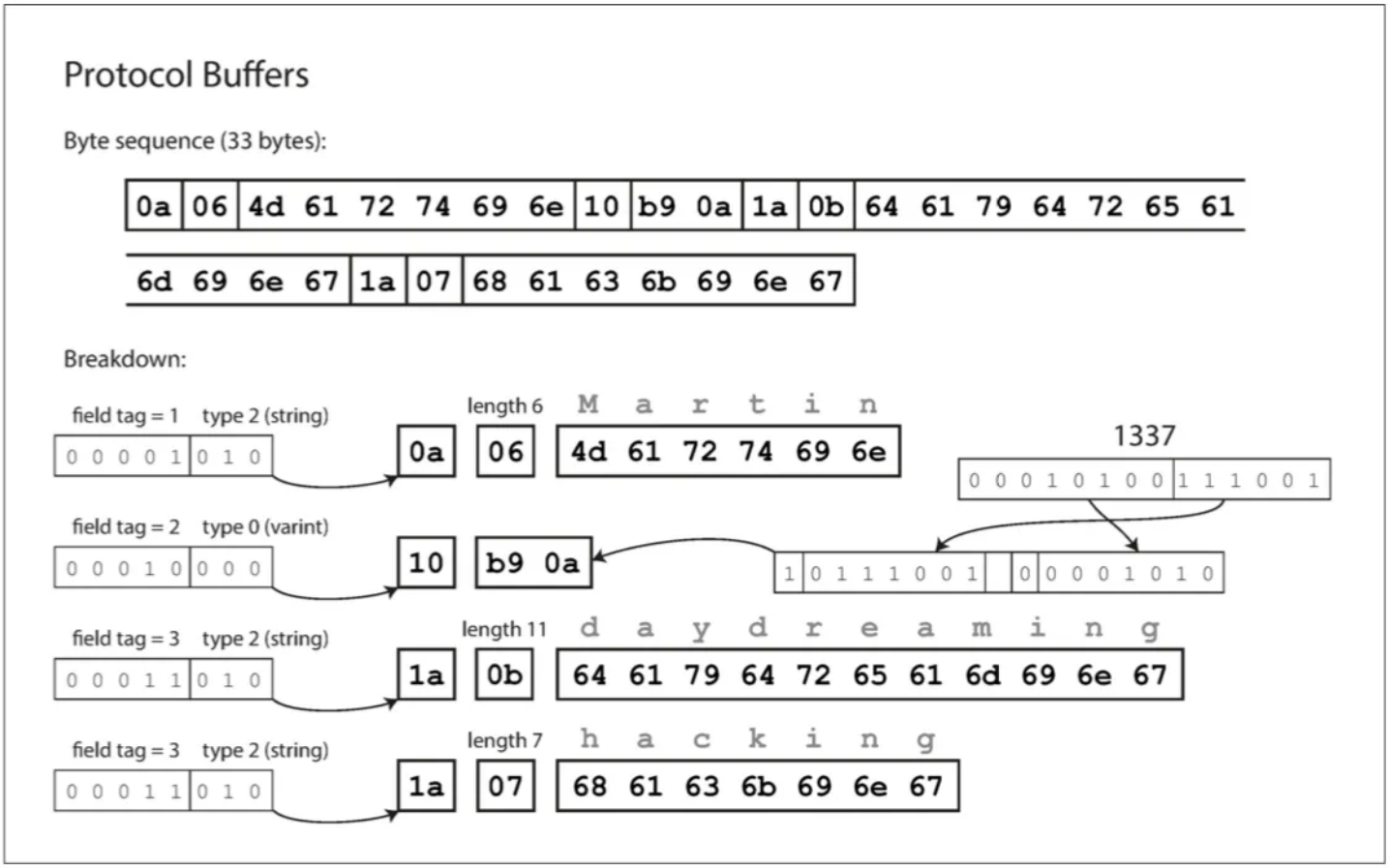
CompactProtocol

- 语义上等同于BinaryProtocol
- 将字段类型和标签号打包到单个字节中，并使用可变长度整数来实现
- 相同的信息打包成只有 34 个字节
- 将数字 1337 编码成为 2 个字节，每个字节的最高位标识是否还有更多的字节。



Protocol Buffers

- 只有一种二进制编码格式，与Thrift的CompactProtocol非常相似。
- 同样的记录塞进了33个字节中。



字段是否为必须?

- 如果字段没有设置值，则从编码记录中省略。
- 模式中每个字段标记为是否为必须，但对编码无影响。
- 区别在于，如果字段设置为必须，但是未设置，那么运行时检查会失败

字段标签和模式演变

- 字段标记很重要！可以改字段的名字，但是不能改字段标记。
- 向前兼容：可以添加新字段，只要有一个新的标记号码。
- 向后兼容：在模式初始部署之后，添加的每个字段必须是可选的或具有默认值。否则之前的代码会检查失败。
- 删除字段：只能删除可选字段；不能再次使用相同的标签号码。

数据类型和模式演变

- 数据类型可以被改变：int32 升级 int64，新代码可以读取旧代码写入的数据（补0）；但是旧代码不能解析新数据（int32 读取 int64 会被截断）
- Protobuf 一个细节：没有列表类型，只有 `repeated`，因此可以把可选字段改为重复字段。
- Thrift 不能把更改为列表参数，但优点是可以嵌套列表。

Avro

- Apache Avro 是另一种二进制编码格式。
- Avro 有两种模式语言：一种（Avro IDL）用于人工编辑，一种（基于JSON）更易于机器读取。

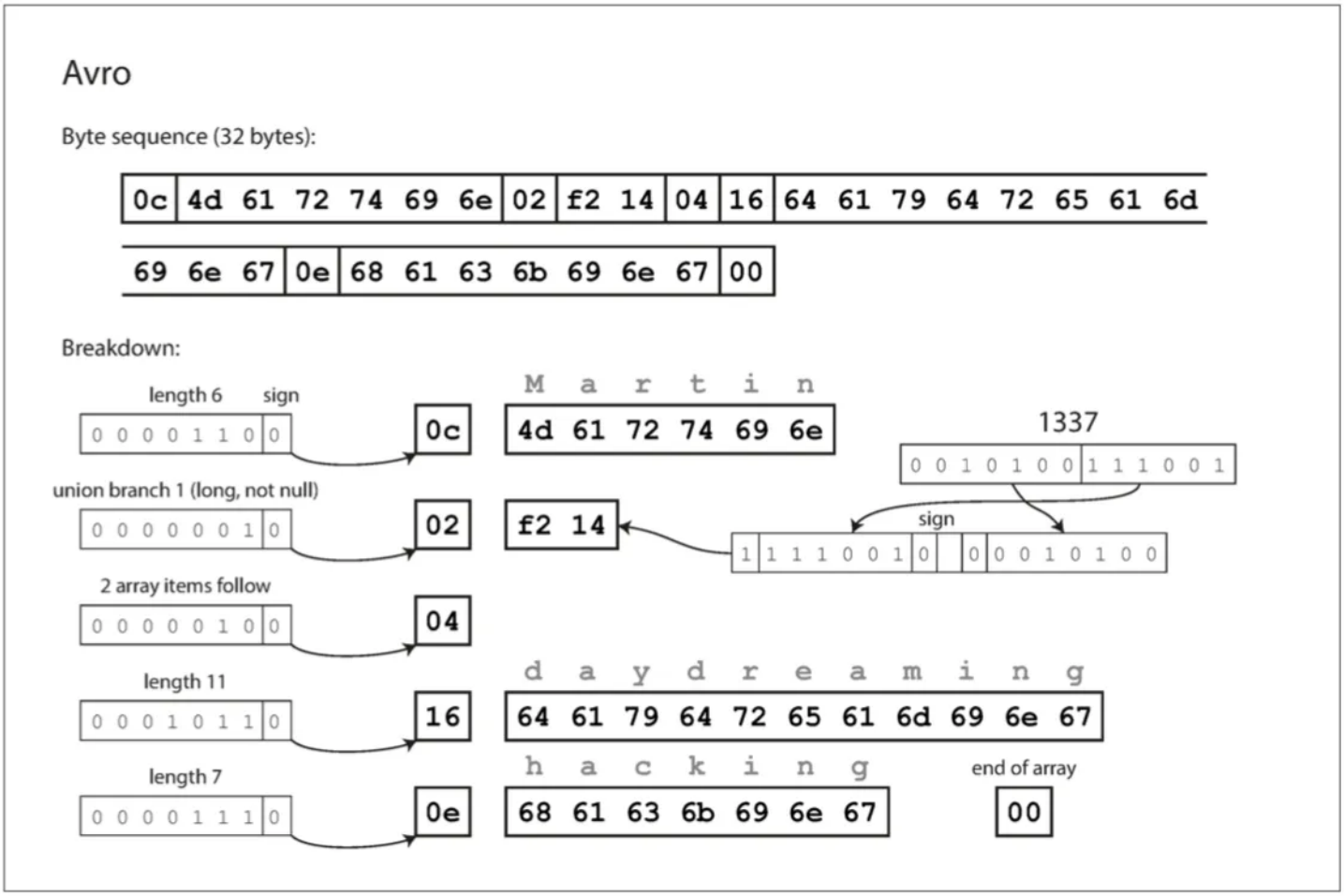
举例：

```
record Person {
    string                userName;
    union { null, long }  favoriteNumber = null;
    array<string>         interests;
}
```

等价JSON表示

```
{
    "type": "record",
    "name": "Person",
    "fields": [
        {"name": "userName", "type": "string"},
        {"name": "favoriteNumber", "type": ["null", "long"], "default": null},
        {"name": "interests", "type": {"type": "array", "items": "string"}
    ]
}
```

- 注意：没有标签号码。
- Avro二进制编码只有32个字节长，最紧凑的。
- 编码知识连在一起的值，不能识别字段和数据类型。



- 必须按照顺序遍历字段才能解码。
- 编码和解码必须使用完全相同的模式。

Avro 如何支持模式演化?

Writer模式与Reader模式

- Avro的关键思想是Writer模式和Reader模式不必是相同的 - 他们只需要兼容。
- 当数据解码（读取）时，Avro库通过并排查看Writer模式和Reader模式并将数据从Writer模式转换到Reader模式来解决差异。（即数据读取的时候，会对比 Writer模式 和 Reader模式 的字段，然后就知道怎么读了）

模式演变规则

- 为了保持兼容性，您只能添加或删除具有默认值的字段。

但Writer模式到底是什么?

对于一段特定的编码数据，Reader如何知道其Writer模式? 答案取决于Avro使用的上下文。举几个例子:

- 有很多记录的大文件
  - Avro的一个常见用途 - 尤其是在Hadoop环境中 - 用于存储包含数百万条记录的大文件，所有记录都使用相同的模式进行编码。可以在文件的开头只包含一次Writer模式。

- 支持独立写入的记录的数据库
  - 最简单的解决方案是在每个编码记录的开始处包含一个版本号，并在数据库中保留一个模式版本列表。
- 通过网络连接发送记录
  - 他们可以在连接设置上协商模式版本，然后在连接的生命周期中使用该模式。

动态生成的模式

- Avro方法的一个优点是架构不包含任何标签号码。

但为什么这很重要？在模式中保留一些数字有什么问题？

- 不同之处在于Avro对动态生成的模式更友善。
  - 方便从数据库生成 Avro 模式，导出数据
  - 当数据库模式发生变化，直接生成新的 Avro 模式，导出数据。自动兼容。
  - 而用 Thrift 或者 PB，需要手动写字段标签。

代码生成和动态类型的语言

- Thrift 和 Protobuf 依赖于代码生成
  - 在定义了模式之后，可以使用您选择的编程语言生成实现此模式的代码。
  - 这在Java，C ++或C＃等静态类型语言中很有用，因为它允许将高效的内存中结构用于解码的数据，并且在编写访问数据结构的程序时允许在IDE中进行类型检查和自动完成。
  - 在动态类型编程语言（如JavaScript，Ruby或Python）中，生成代码没有太多意义，因为没有编译时类型检查器来满足。
  - Avro为静态类型编程语言提供了可选的代码生成功能，但是它也可以在不生成任何代码的情况下使用。

模式的优点

- Protocol Buffers，Thrift和Avro都使用模式来描述二进制编码格式。
  - 他们的模式语言比XML模式或者JSON模式简单得多，也支持更详细的验证规则。
- 许多数据系统（如关系型数据库）也为其数据实现了某种专有的二进制编码。
- 基于模式的二进制编码相对于JSON，XML和CSV等文本数据格式的优点：
  - 它们可以比各种“二进制JSON”变体更紧凑，因为它们可以省略编码数据中的字段名称。
  - 模式是一种有价值的文档形式，因为模式是解码所必需的，所以可以确定它是最新的（而手动维护的文档可能很容易偏离现实）。
  - 维护一个模式的数据库允许您在部署任何内容之前检查模式更改的向前和向后兼容性。

- 对于静态类型编程语言的用户来说，从模式生成代码的能力是有用的，因为它可以在编译时进行类型检查

数据流模式

每当将一些数据发送到非共享内存的另一个进程时，例如，当通过网络发送数据或者把它写入文件时，都需要将数据编码为字节序列。讨论用于执行此操作的各种不同编码技术。

而向前和向后的兼容性对于可演化性来说非常重要，通过允许独立升级系统的不同部分，而不必一次改变所有，使更改更为容易。兼容性是执行编码的一个进程和执行解码的另一个进程之间的关系。

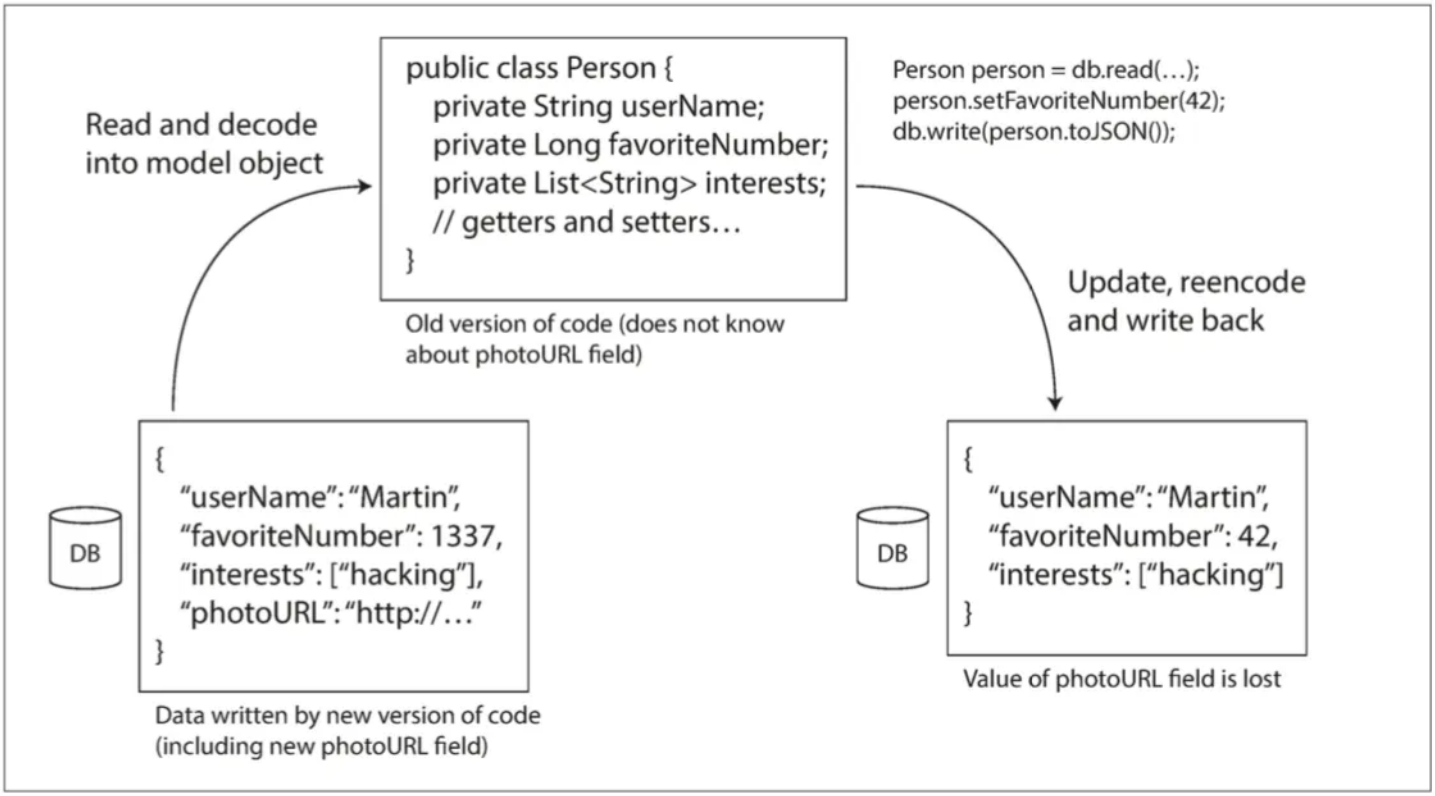
数据可以通过多种方式从一个进程流向另一个进程。谁编码数据？谁解码数据？

最常见的进程间数据流动的方式：

- 通过数据库
- 通过服务调用
- 通过异步消息传递

数据库中的数据流

- 如果只有一个进程访问数据库，向后兼容性显然是必要的。
- 一般来说，会有多个进程访问数据库，可能会有某些进程运行较新代码、某些运行较旧的代码。因此数据库也经常需要向前兼容。
- 假设增加字段，那么较新的代码会写入把该值吸入数据库。而旧版本的代码将读取记录，理想的行为是旧代码保持领域完整。
- 用旧代码读取并重新写入数据库时，有可能会导致数据丢失



在不同的时间写入不同的值

- 单一的数据库中，可能有一些值是五毫秒前写的，而一些值是五年前写的。
- 架构演变允许整个数据库看起来好像是用单个模式编码的，即使底层存储可能包含用模式的各种历史版本编码的记录。

归档存储

- 建立数据库快照，比如备份或者加载到数据仓库：即使有不同时代的模式版本的混合，但通常使用最新模式进行编码。
- 由于数据转储是一次写入的，以后不变，所以 Avro 对象容器文件等格式非常适合。
- 也是很好的机会，把数据编码成面向分析的列式格式。

服务中的数据流：REST 和 RPC

- 网络通信方式：常见安排是客户端+服务器
- Web 服务：通过 GET 和 POST 请求
- 服务端可以是另一个服务的客户端：微服务架构。
- 微服务架构允许某个团队能够经常发布新版本服务，期望服务的新旧版本同时运行。

Web 服务

- 当服务使用HTTP作为底层通信协议时，可称之为Web服务。
- 有两种流行的Web服务方法：REST和SOAP。

REST

- REST不是一个协议，而是一个基于HTTP原则的设计哲学。
- 它强调简单的数据格式，使用URL来标识资源，并使用HTTP功能进行缓存控制，身份验证和内容类型协商。
- 与SOAP相比，REST已经越来越受欢迎，至少在跨组织服务集成的背景下，并经常与微服务相关。
- 根据REST原则设计的API称为RESTful。

SOAP

- SOAP是用于制作网络API请求的基于XML的协议。
- 它最常用于HTTP，但其目的是独立于HTTP，并避免使用大多数HTTP功能。
- SOAP Web服务的API使用称为Web服务描述语言（WSDL）的基于XML的语言来描述。 WSDL支持代码生成，客户端可以使用本地类和方法调用（编码为XML消息并由框架再次解码）访问远程服务。
- 尽管SOAP及其各种扩展表面上是标准化的，但是不同厂商的实现之间的互操作性往往会造成问题。
- 尽管许多大型企业仍然使用SOAP，但在大多数小公司中已经不再受到青睐

远程过程调用（RPC）的问题

RPC模型试图向远程网络服务发出请求，看起来与在同一进程中调用编程语言中的函数或方法相同（这种抽象称为位置透明）。

RPC 的缺陷：

- 本地函数调用是可预测的，并且成功或失败仅取决于受您控制的参数。而网络请求是不可预知的。
- 本地函数调用要么返回结果，要么抛出异常，或者永远不返回（因为进入无限循环或进程崩溃）。网络请求有另一个可能的结果：由于超时，它可能会返回没有结果。无法得知远程服务的响应发生了什么。
- 如果您重试失败的网络请求，可能会发生请求实际上正在通过，只有响应丢失。在这种情况下，重试将导致该操作被执行多次，除非您在协议中引入除重（ 幂等（idempotence）） 机制。本地函数调用没有这个问题。
- 每次调用本地功能时，通常需要大致相同的时间来执行。网络请求慢得多，不可预知。
- 调用本地函数时，可以高效地将引用（指针）传递给本地内存中的对象。当你发出一个网络请求时，所有这些参数都需要被编码成可以通过网络发送的一系列字节。如果参数是像数字或字符串这样的基本类型倒是没关系，但是对于较大的对象很快就会变成问题。
- 客户端和服务端可以用不同的编程语言实现，RPC 框架必须把数据类型做翻译，可能会出问题。

RPC的当前方向

RPC 不会消失。

- Thrift和Avro带有RPC支持
- gRPC是使用Protocol Buffers的RPC实现
- Finagle也使用Thrift
- Rest.li使用JSON over HTTP。

当前方向：

- 这种新一代的RPC框架更加明确的是，远程请求与本地函数调用不同。
- 其中一些框架还提供服务发现，即允许客户端找出在哪个IP地址和端口号上可以找到特定的服务。
- REST似乎是公共API的主要风格。
    - REST 使用二进制编码性能更好
    - 方便实验和调试
    - 能被所有主流的编程语言和平台支持
    - 大量可用的工具的生态系统

数据编码与RPC的演化

- 可演化性，重要的是可以独立更改和部署RPC客户端和服务器。
- 我们可以做个假定：假定所有的服务器都会先更新，其次是所有的客户端。
- 您只需要在请求上具有向后兼容性，并且对响应具有前向兼容性。

RPC方案的前后向兼容性属性从它使用的编码方式中继承：

- Thrift，gRPC（Protobuf）和Avro RPC可以根据相应编码格式的兼容性规则进行演变。
- 在SOAP中，请求和响应是使用XML模式指定的。
- RESTful API 通常使用 JSON（没有正式指定的模式）用于响应，以及用于请求的JSON或URI编码/表单编码的请求参数。

服务的提供者无法控制其客户，所以可能无限期保持兼容性。 对于 RESTful API，常用方法是在 URL 或者 HTTP Accept 头部使用版本号。

消息传递中的数据流

RPC和数据库之间的异步消息传递系统，与RPC的相似之处在于，客户端的请求（消息）以低延迟传递到另一个进程。它们和数据库的相似之处在于，不是通过直接的网络连接发送消息，而是通过称为消息代理（也称消息队列，或面向消息的中间件）的中介发送的，该中介会暂存消息。

与直接RPC相比，使用消息代理（消息队列）有几个优点：

- 如果收件人不可用或过载，可以充当缓冲区，从而提高系统的可靠性。
- 它可以自动将消息重新发送到已经崩溃的进程，从而防止消息丢失。
- 避免发件人需要知道收件人的IP地址和端口号（这在虚拟机经常出入的云部署中特别有用）。
- 它允许将一条消息发送给多个收件人。
- 将发件人与收件人逻辑分离（发件人只是发布邮件，不关心使用者）。

与 PRC 相比，差异在于

- 消息传递通常是单向的：发送者通常不期望收到其消息的回复。
- 通信模式是异步的：发送者不会等待消息被传递，而只是发送它，然后忘记它。

消息代理

- RabbitMQ，ActiveMQ，HornetQ，NATS和Apache Kafka这样的开源实现已经流行起来。
- 通常情况下，消息代理的使用方式如下：
  - 一个进程将消息发送到指定的队列或主题；
  - 代理确保将消息传递给那个队列或主题的一个或多个消费者或订阅者。
  - 在同一主题上可以有许多生产者和许多消费者。
- 一个主题只提供单向数据流。但是，消费者本身可能会将消息发布到另一个主题上，或者发送给原始消息的发送者使用的回复队列（允许请求/响应数据流，类似于RPC）。
- 消息代理通常不会执行任何特定的数据模型，消息知识包含一些元数据的字节序列，可以用任何编码格式。
- 如果消费者重新发布消息到另一个主题，则消息保留未知字段，防止前面数据库环境中描述的问题。

分布式的Actor框架

- Actor模型是单个进程中并发的编程模型。
- 逻辑被封装在actor中，而不是直接处理线程（以及竞争条件，锁定和死锁的相关问题）。
- 每个actor通常代表一个客户或实体，它可能有一些本地状态（不与其他任何角色共享），它通过发送和接收异步消息与其他角色通信。
- 不保证消息传送：在某些错误情况下，消息将丢失。
- 由于每个角色一次只能处理一条消息，因此不需要担心线程，每个角色可以由框架独立调度。

分布式 Actor 框架

- 在分布式Actor框架中，此编程模型用于跨多个节点伸缩应用程序。
- 不管发送方和接收方是在同一个节点上还是在不同的节点上，都使用相同的消息传递机制。
- 如果它们在不同的节点上，则该消息被透明地编码成字节序列，通过网络发送，并在另一侧解码。

位置透明

- 位置透明在actor模型中比在RPC中效果更好，因为actor模型已经假定消息可能会丢失，即使在单个进程中也是如此。
- 尽管网络上的延迟可能比同一个进程中的延迟更高，但是在使用actor模型时，本地和远程通信之间的基本不匹配是较少的。

升级

- 分布式的Actor框架实质上是将消息代理和actor编程模型集成到一个框架中。
- 升级仍然要担心向前和向后兼容问题。

三个流行的分布式actor框架处理消息编码如下：

- 默认情况下，Akka使用Java的内置序列化，不提供前向或后向兼容性。 但是，你可以用类似Prototol Buffers的东西替代它，从而获得滚动升级的能力。
- Orleans 默认使用不支持滚动升级部署的自定义数据编码格式; 要部署新版本的应用程序，您需要设置一个新的集群，将流量从旧集群迁移到新集群，然后关闭旧集群。 像Akka一样，可以使用自定义序列化插件。
- 在Erlang OTP中，对记录模式进行更改是非常困难的（尽管系统具有许多为高可用性设计的功能）。 滚动升级是可能的，但需要仔细计划。

小结

本章研究了将内存数据结构转换为网络或磁盘上字节流的多种方式。这些编码的细节不仅影响其效率，更重要的是还影响应用程序的体系结构和部署时的支持选项。

许多服务需要支持滚动升级，即每次将新版本的服务逐步部署到几个节点，而不是同时部署到所有节点。滚动升级允许在不停机的情况下发布新版本的服务（因此鼓励频繁地发布小版本而不是大版本），并降低部署风险（允许错误版本在影响大量用户之前检测并回滚）。这些特性非常有利于应用程序的演化和更改。

滚动升级期间，由于各种其他原因，必须假设不同的节点正在运行应用代码的不同版本。因此，在系统内流动的所有数据都以提供向后兼容性（新代码可以读取旧数据）和向前兼容性（旧代码可以读取新数据）的方式进行编码非常重要。

还讨论了多种数据编码格式及其兼容性情况：

- 编程语言特定的编码仅限于单一编程语言，并且往往无法提供前向和后向兼容性。

- JSON，XML和CSV等文本格式非常普遍，其兼容性取决于您如何使用它们。他们有可选的模式语言，这有时是有用的，有时是一个障碍。这些格式对于数据类型有些模糊，所以你必须小心数字和二进制字符串。
- 像Thrift，Protocol Buffers和Avro这样的二进制模式驱动格式允许使用清晰定义的前向和后向兼容性语义进行紧凑，高效的编码。这些模式可以用于静态类型语言的文档和代码生成。但是，他们有一个缺点，就是在数据可读之前需要对数据进行解码。

我们还讨论了数据流的几种模式，说明了数据编码重要性的不同场景：

- 数据库，写入数据库的进程对数据进行编码，并从数据库读取进程对其进行解码
- RPC和REST API，客户端对请求进行编码，服务器对请求进行解码并对响应进行编码，客户端最终对响应进行解码
- 异步消息传递（使用消息代理或参与者），其中节点之间通过发送消息进行通信，消息由发送者编码并由接收者解码

结论：前向兼容性和滚动升级在某种程度上是完全可以实现的。