# Designing Data Intensive Applications

## Capítulo 4: Codificação e Evolução

Bruna Maciel
Abril/2018

# Introdução

- Codificação
- Evolução
- Compatibilidade
  - Forward
  - Backward

# Codificação de dados

Normalmente os programas trabalham com dados em (pelo menos) duas representações diferentes:

- na memória :

  estruturas, listas, matrizes, tabelas de hash, árvores, etc.

- para gravar arquivos ou enviar pela rede:

  Codificação para outros processos (encoding)

# Codificação de dados

- Linguagem de programação (Java, Python, Ruby)

São restritas a uma única linguagem de programação e geralmente não fornecem compatibilidade com versões anteriores e posteriores.

- Formatos textuais (JSON, XML, CSV)

A compatibilidade depende do uso e são um tanto vagos em relação aos tipos de dados. Têm linguagens de esquema opcionais.

- Esquemas Binários (Thrift, Protocol Buffers e Avro)

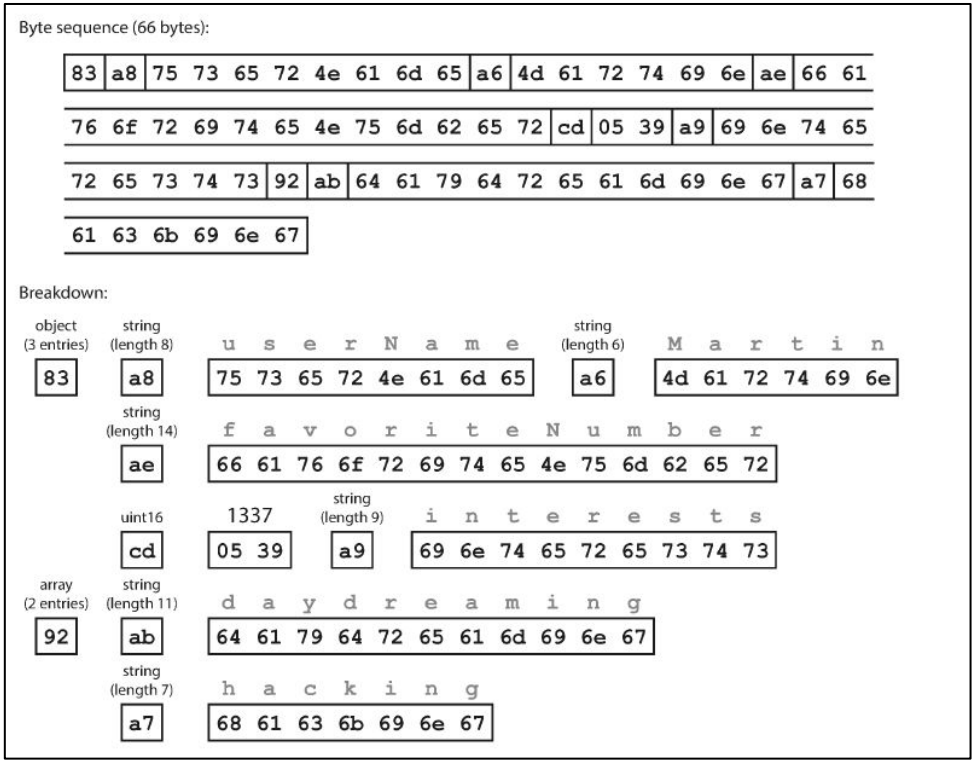
Permitem codificação compacta e eficiente com semânticas definidas. Não são legíveis para humanos mas são bons para geração de código em linguagens estaticamente tipadas

# MessagePack (JSON)

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

Sequencia de bytes

| Qtd objetos | Tipo campo | Nome campo | Tam dados | Dados |
|---|---|---|---|---|

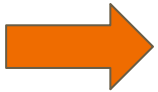


# Thrift e  Protocol Buffers

Esquema de dados e ferramenta de geração de dados

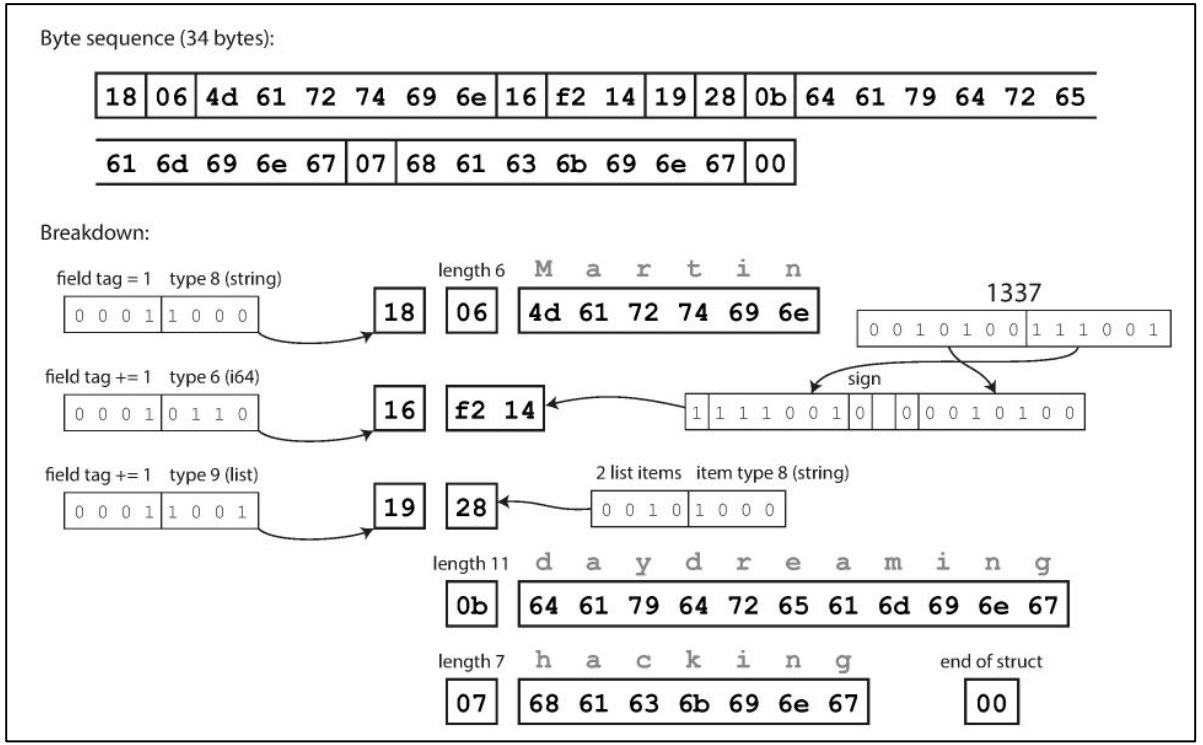
Codificação:

| Tag + Tipo | Tamanho | Dados |
|---|---|---|

Compatibilidade  Tag exclusiva

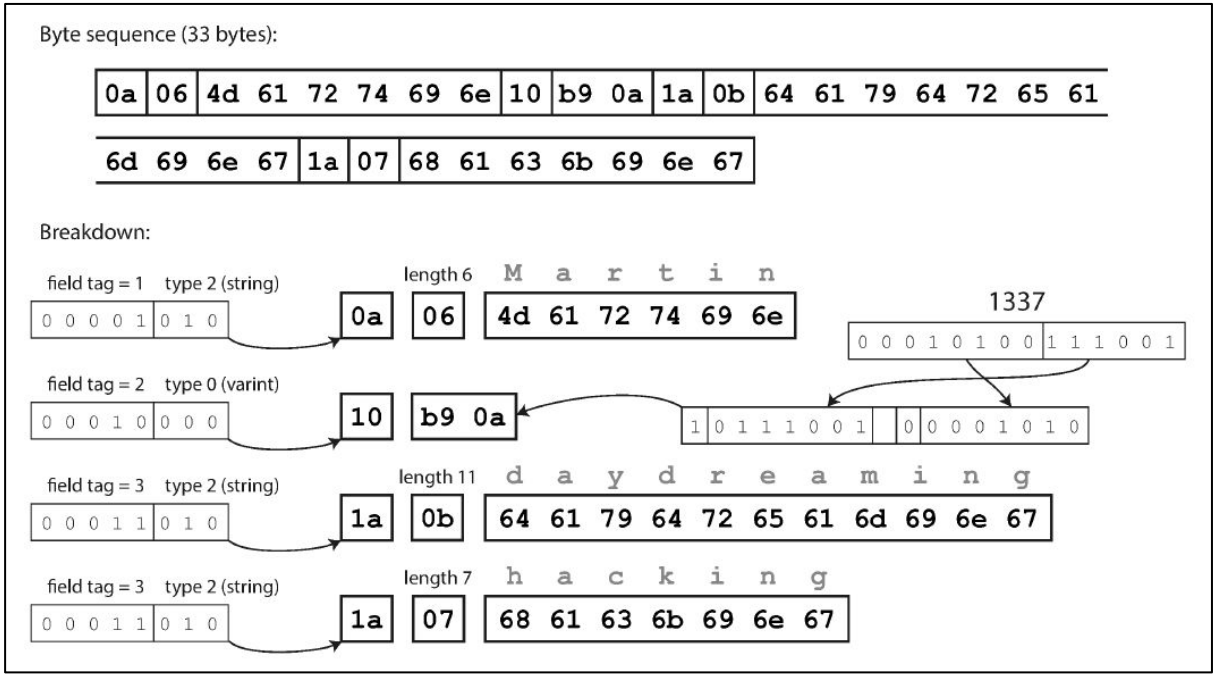
# Thrift

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



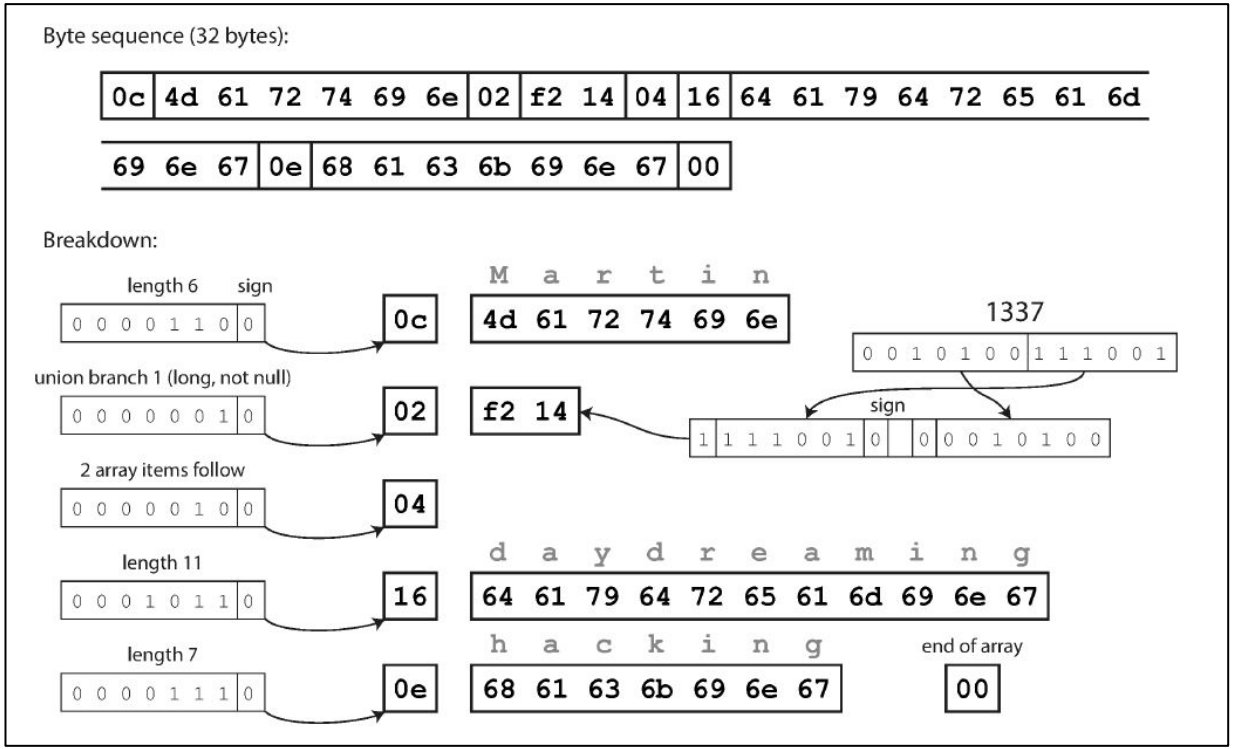
# Protocol Buffers

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



# Avro

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



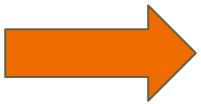
# Avro

Esquema de dados e ferramenta de geração de dados opcional

Codificação:

| Tamanho | Dados |
|---|---|

Percorre os campos na ordem em que eles aparecem no esquema e usa o esquema para informar o tipo de dados de cada campo

Compatibilidade  Versão dos esquemas leitor e escritor

# Avro

Decodificação: compara os esquemas leitor e escritor

Esquema escritor

| Datatype | Field name |
|---|---|
| string | userName |
| union {null, long} | favoriteNumber |
| array<string> | interests |
| string | photoURL |

Esquema leitor

| Datatype | Field name |
|---|---|
| long | userID |
| union {null, int} | favoriteNumber |
| string | userName |
| array<string> | interests |

# Avro

Para o leitor conhecer o esquema do escritor :

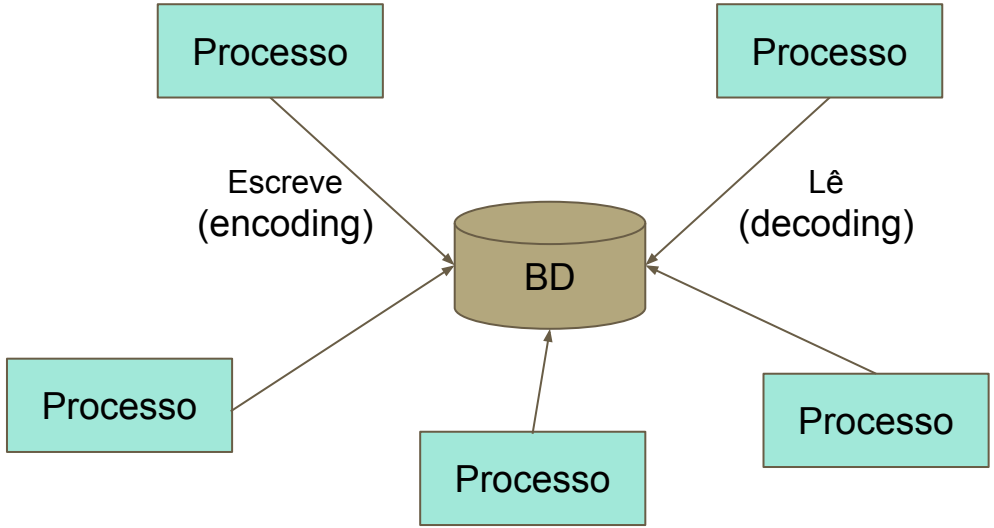
- Arquivo grande com muitos registros todos codificados com o mesmo esquema
- Banco de dados com registros gravados individualmente
- Enviando registros por uma conexão de rede

# Fluxo de dados


Mas como os dados saem de um processo para o outro?

- Bancos de dados
- Chamadas de serviço: REST e RPC
- Passagem assíncrona de mensagens.

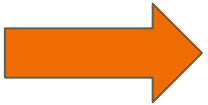
# Fluxo por banco de dados




Evolução e compatibilidade

# Fluxo por chamada de serviço

Comunicação na rede: clientes e servidores

Serviço (API)  Restrições na consulta

Abordagens
- REST
- RPC

# REST

Utiliza o protocolo HTTP para comunicar com os serviços

## REST

- Não é protocolo
- Usa recursos do HTTP
- Abordagens mais simples, geralmente envolvendo menos geração de código e ferramentas automatizadas
- Pode fazer solicitações a ela usando um navegador da Web ou a ferramenta de linha de comando
- É suportada por todas as linguagens de programação e plataformas mainstream
- Predominante para APIs públicas

# RPC (Remote Procedure Call)

O modelo RPC tenta fazer um pedido para um serviço de rede remota parecer o mesmo que chamar uma função ou método em sua linguagem de programação, dentro do mesmo processo.

**PROBLEMAS?**

Muitos! uma solicitação de rede é muito diferente de uma chamada de função local

# Problemas RPC

| Chamada de função | | Solicitação de rede |
|---|---|---|
| é previsível: tem êxito ou falha | → | é imprevisível: a solicitação ou resposta pode ser perdida |
| retorna um resultado, lança uma exceção ou entra em loop | → | pode retornar sem resultado, devido ao timeout |
| tempo para ser executada é quase sempre o mesmo | → | é muito mais lenta, e sua latência também é muito variável |
| Passagem de referências (ponteiros) a objetos na memória local com eficiência | → | parâmetros precisam ser codificados em uma seqüência de bytes |
| | | solicitação de rede com falha na resposta |

# RPC (Remote Procedure Call)

Enquanto o  REST é predominante para APIs públicas, o foco principal das estruturas de RPC é em solicitações entre serviços pertencentes à mesma organização, geralmente dentro do mesmo datacenter.

**Evolução:**

Clientes e servidores devem ser alterados e implementados de forma independente

**Compatibilidade:**

Atualizando os servidores primeiro e todos os clientes em segundo, só  é preciso de compatibilidade backward em solicitações e forward nas respostas

**Problema:** o RPC é frequentemente usado para comunicação entre limites organizacionais, portanto, o provedor de um serviço geralmente não tem controle sobre seus clientes e não pode forçá-los a atualizar
**Solução:**  o provedor de serviços geralmente manterá várias versões da API de serviço lado a lado.

# Passagem assíncrona de mensagens

- Mensagem é entregue a outro processo com baixa latência (similar ao RPC)
- Mensagem passa por um intermediário de mensagem (fila de mensagens ou middleware orientado a mensagens), que armazena a mensagem temporariamente (similar ao BD)
- Não aguarda resposta

**Intermediários de mensagens**

- Um processo envia uma mensagem para uma fila ou tópico nomeado e o intermediário assegura que a mensagem seja entregue a um ou mais consumidores ou assinantes para essa fila ou tópico
- Não impõem nenhum modelo de dados específico - uma mensagem é apenas uma sequência de bytes com alguns metadados, para usar qualquer formato de codificação

# Passagem assíncrona de mensagens

**Modelo de ator**

É um modelo de programação para controle de concorrência em um único processo. Cada ator representa um cliente ou entidade, pode ter algum estado local (que não é compartilhado com qualquer outro ator) e se comunica com outros atores enviando e recebendo mensagens assíncronas.

- Usado para dimensionar um aplicativo em vários nós.
- Pressupõe que as mensagens podem ser perdidas (melhor de RPC)

**Framework de ator distribuído**

Integra um intermediário de mensagem e o modelo de ator em uma única estrutura.
Compatibilidade com versões anteriores e posteriores é pela versão dos nós.

# Perguntas?

# Exercícios

1) Para que serve a codificação e o que a compatibilidade (backward e forward) influenciam na evolução?

2) Explique as principais diferenças entre os fluxos de dados apresentados