

Segundo Edição_n

Kafka

O Guia Definitivo

Processamento de dados e fluxo em
tempo real em escala

Gwen Shapira, Todd Palino,
Rajini Sivaram e Krit Petty

Kafka: o guia definitivo

Cada aplicativo corporativo cria dados, sejam eles mensagens de log, métricas, atividade do usuário ou mensagens de saída. Mover todos esses dados é tão importante quanto os próprios dados. Com

Nesta edição atualizada, arquitetos de aplicativos, desenvolvedores e engenheiros de produção novos na plataforma de streaming Kafka aprenderão como lidar com dados em movimento. Capítulos adicionais cobrem a API AdminClient do Kafka, transações, novos recursos de segurança e alterações de ferramentas.

Os engenheiros da Confluent e do LinkedIn responsáveis pelo desenvolvimento do Kafka explicam como implantar clusters Kafka de

produção, escrever microsserviços confiáveis orientados a eventos e criar aplicativos escalonáveis de processamento de fluxo com esta plataforma. Por meio de exemplos detalhados, você aprenderá os princípios de design, garantias de confiabilidade, APIs principais e detalhes de arquitetura do Kafka, incluindo o protocolo de replicação, o controlador e a camada de armazenamento.

Você examinará:

- Melhores práticas para implantar e configurar o Kafka
- Produtores e consumidores Kafka para escrever e ler mensagens
- Padrões e requisitos de casos de uso para garantir a entrega confiável de dados
- Melhores práticas para construir pipelines de

dados e aplicativos com Kafka

- Como realizar tarefas de monitoramento, ajuste e manutenção com Kafka em produção
- As métricas mais críticas entre as medições operacionais do Kafka
- Capacidades de entrega do Kafka para sistemas de processamento de fluxo

DADOS | BANCOS DE DADOS

US\$ 69,99 PODE US\$ 92,99

“Um must-have para desenvolvedores e operadores igualmente. Você precisa deste livro se estiver usando ou executando o Kafka.”

—Chris Riccomini

Engenheiro de software, consultor de startups e coautor de O LEIA-ME que falta

Gwen Shapira é líder de engenharia na Confluent e gerencia a equipe Kafka nativa da nuvem, responsável pelo desempenho, elasticidade e multitenancy do Kafka.

Todd Palino, principal engenheiro de confiabilidade de sites do LinkedIn, é responsável pelo planejamento de capacidade e eficiência.

Rajini Sivaram é engenheiro principal da Confluent, projetando e desenvolvendo replicação entre clusters e recursos de segurança para Kafka.

Krit Petty é a confiabilidade do site gerente de engenharia da Kafka no LinkedIn.

Twitter: @oreillymedia
facebook.com/oreilly

ISBN: 978-1-492-04308-9

Elogio por *Kafka: o guia definitivo*

Kafka: o guia definitivo tem tudo que você precisa saber para aproveitar ao máximo o Kafka, seja na nuvem ou no local. Um item obrigatório para desenvolvedores e operadores. Gwen, Todd, Rajini e Krit reúnem anos de sabedoria em um livro conciso. Você precisa deste livro se estiver usando ou executando o Kafka.

—Chris Riccomini, engenheiro de software, consultor de startups, e coautor de O LEIA-ME que falta

Um guia completo sobre os fundamentos do Kafka e como

operacionalizá-lo. —**Sumant**

Tambe, engenheiro de software sênior do LinkedIn

Este livro é uma leitura essencial para qualquer desenvolvedor ou administrador Kafka. Leia-o de capa a capa para mergulhar em seus detalhes ou mantenha-o à mão para referência rápida. De qualquer forma,

sua clareza de redação e precisão técnica são excelentes.

—Robin Hoffatt, defensor da equipe de desenvolvedores da Confluent

Esta é uma literatura fundamental para todos os engenheiros interessados em Kafka. Foi fundamental para ajudar Robinhood a navegar no dimensionamento, atualização e ajuste do Kafka para apoiar nosso rápido crescimento de usuários.

—Jaren H. Clover, engenheiro inicial da Robinhood, investidor anjo

Uma leitura obrigatória para todos que trabalham com Apache Kafka: desenvolvedor ou administrador, iniciante ou especialista, usuário ou colaborador.

—Matthias J. Sax, engenheiro de software da Confluent e membro do Apache Kafka PMC

Ótima orientação para qualquer equipe que usa seriamente o Apache Kafka na produção e para engenheiros que trabalham em sistemas distribuídos em geral. Este livro vai muito além da cobertura usual de nível introdutório e aborda como Kafka realmente funciona, como deve ser usado e onde estão as armadilhas.

Para cada grande recurso do Kafka, os autores listam claramente as advertências que você só ouviria falar dos veteranos grisalhos do Kafka. Esta informação não está facilmente disponível em um lugar em qualquer outro lugar. A clareza e profundidade das explicações são tantas que eu recomendaria até mesmo para engenheiros que não usam Kafka: aprender sobre os princípios, opções de design e dicas operacionais os ajudará a tomar melhores decisões ao criar outros sistemas.

—Dmitriy Ryaboy, vice-presidente de engenharia de software da Zymergen
SEGUNDA EDIÇÃO

Kafka: o guia definitivo

Processamento de dados e fluxo em tempo real em escala

***Gwen Shapira, Todd Palino,
Rajini Sivaram e Krit Petty***

Pequim Boston Farnham Sebastopol Tóquio

Kafka: o guia definitivo

por Gwen Shapira, Todd Palino, Rajini Sivaram e Krit Petty

Copyright © 2022 Chen Shapira, Todd Palino, Rajini Sivaram e Krit Petty. Todos os direitos reservados. Impresso nos Estados Unidos da América.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Os livros da O'Reilly podem ser adquiridos para uso educacional, comercial ou promocional de vendas. Edições online também estão disponíveis para a maioria dos títulos (**<http://oreilly.com>**). Para mais informações, entre em contato com nosso departamento de vendas corporativo/institucional: 800-998-9938 ou **corporate@oreilly.com**.

Editor de Aquisições: Jéssica

Haberman **Editor de**

Desenvolvimento: Gary O'Brien

Editor de produção: Kate Galloway

Editor de texto: Sônia Saruba

Revisor: Piper Consultoria Editorial,
LLC

Setembro de 2017: Primeira Edição

Novembro de 2021: Segunda Edição

Histórico de revisões para a

segunda edição 05/11/2021: Primeiro
lançamento

Indexador: Ellen Troutman-Zaig

Designer de Interiores: David Futato

Ver

<http://oreilly.com/catalog/errata.csp?isbn=9781492043089>
para detalhes do lançamento.

O logotipo O'Reilly é uma marca registrada da O'Reilly Media, Inc. **Kafka: o guia definitivo**, a imagem da capa e a imagem comercial relacionada são marcas registradas da O'Reilly Media, Inc.

As opiniões expressas neste trabalho são de responsabilidade dos autores e não representam as opiniões do editor. Embora a editora e os autores tenham envidado esforços de boa fé para garantir que as informações e instruções contidas neste trabalho sejam precisas, a editora e os autores isentam-se de qualquer responsabilidade por erros ou omissões, incluindo, sem limitação, responsabilidade por danos resultantes do uso ou confiança neste trabalho. O uso das informações e instruções contidas neste trabalho é por sua conta e risco. Se quaisquer exemplos de código ou outra tecnologia que este trabalho contém ou descreve estão sujeitos a licenças de código aberto ou aos direitos de propriedade intelectual de terceiros, é sua responsabilidade garantir que seu uso esteja em conformidade com tais licenças e/ou direitos.

Este trabalho faz parte de uma colaboração entre O'Reilly e Confluent. Veja nosso [declaração de independência editorial](#).

978-1-492-04308-9

[LSI]

Índice

Prefácio à segunda edição.	
. xv Prefácio à Primeira Edição.	

1. Conheça Kafka.

..... **1** Mensagens de publicação/assinatura
1 Como começa 2 Sistemas de filas individuais 3 Digite Kafka 4 Mensagens
e lotes 4 Esquemas 5 Tópicos e partições 5 Produtores e consumidores 6
Corretores e clusters 8 Clusters múltiplos 9 Por que Kafka? 10 Vários
produtores 10 Vários consumidores 10 Retenção baseada em disco 11
Escalável 11 Alto desempenho 11 Recursos da plataforma 11 O ecossistema
de dados 12 Casos de uso 12 A origem do Kafka 14

v

O problema do LinkedIn 14 O nascimento do Kafka 15 Código aberto 16
Engajamento comercial 16 O nome 17 Primeiros passos com o Kafka 17

2. Instalando o Kafka.

..... **19** Configuração do ambiente 19
Escolhendo um sistema operacional 19 Instalando o Java 19 Instalando o
ZooKeeper 20 Instalando um Broker Kafka 23 Configurando o Broker 24
Parâmetros gerais do Broker 25 Padrões do tópico 27 Seleccionando
hardware 33 Taxa de transferência do disco 33 Capacidade do disco 34
Memória 34 Rede 35 CPU 35 Kafka na nuvem 35 Microsoft Azure 36
Amazon Web Services 36 Configurando clusters Kafka 36 Quantos
corretores? 37 Configuração do Broker 38 Ajuste do sistema operacional 38
Preocupações com a produção 42 Opções do coletor de lixo 42 Layout do
datacenter 43 Colocação de aplicativos no ZooKeeper 44 Resumo 46

3. Produtores de Kafka: escrevendo mensagens para Kafka. . .

..... **47** Visão geral do
produtor 48 Construindo um produtor Kafka 50 Enviando uma mensagem
para Kafka 52
Enviando uma mensagem de forma síncrona 52 Enviando uma
mensagem de forma assíncrona 53

vi | Índice

Configurando produtores 54 client.id 55 acks 55 Tempo de entrega de
mensagens 56 linger.ms 59 buffer.memory 59 compression.type 59
batch.size 59 max.in.flight.requests.per.connection 60 max.request.size
60 receiver.buffer.bytes e send.buffer.bytes 61 enable.idempotence 61
Serializadores 61 Serializadores personalizados 62 Serializando usando
Apache Avro 64 Usando registros Avro com Kafka 65

4. Consumidores Kafka: Lendo dados de Kafka.

.....	77	Conceitos do consumidor Kafka	77
Consumidores e grupos de consumidores	77	Grupos de consumidores e rebalanceamento de partição	80
Associação a grupos estáticos	83	Criando um consumidor Kafka	84
Assinando tópicos	85	O loop de pesquisa	86
Thread Safety	87	Configurando consumidores	88
fetch.min.bytes	88	fetch.max.wait.ms	88
fetch.max.bytes	89	max.poll.records	89
max.partition.fetch.bytes	89	session.timeout.ms e heartbeat.interval.ms	89
max.poll.interval.ms	90	default.api.timeout.ms	90
solicitação.timeout.ms	90		

Índice | vii

auto.offset.reset	91	enable.auto.commit	91	partição.assignment.strategy	91
cliente.id	93	cliente.rack	93	grupo.instance.id	93
receber.buffer.bytes	93	e enviar.buffer.bytes	93	deslocamentos.retenção.minutos	93
Confirmações e deslocamentos	94	Confirmação automática	95	Confirmação do deslocamento atual	96
Confirmação assíncrona	97	Combinando confirmações síncronas e assíncronas	99	Confirmando um deslocamento especificado	100
Rebalancear Listeners	101	Consumindo Registros com Compensações Específicas	104	Mas Como Saímos?	105
Desserializadores	106	Desserializadores personalizados	107	Usando a desserialização Avro com Kafka Consumer	109
Consumidor autônomo: por que e como usar um consumidor sem um grupo	110	Resumo	111		

5. Gerenciando o Apache Kafka programaticamente.

.....	113	Visão geral do AdminClient	114	API assíncrona e eventualmente consistente	114
Opções	114	Hierarquia plana	115	Notas adicionais	115
Ciclo de vida do AdminClient: criação, configuração e fechamento	115	client.dns.lookup	116	request.timeout.ms	117
Gerenciamento de tópicos essenciais	118	Gerenciamento de configuração	121	Gerenciamento de grupo de consumidores	123
Explorando grupos de consumidores	123	Modificando grupos de consumidores	125	Metadados de cluster	127
Operações administrativas avançadas	127	Adicionando partições a um tópico	127	Excluindo registros de um tópico	128

viii | Índice

Eleição do Líder	128	Reatribuindo Réplicas	129	Testes	131	Resumo	133
------------------	-----	-----------------------	-----	--------	-----	--------	-----

6. Kafka Internos.

.....	135	Associação ao Cluster	135	O Controlador	136
KRaft: Novo controlador baseado em jangada de Kafka	137	Replicação			

139	Processamento de solicitação	142
Solicitações de produção	144	Solicitações de busca
145	Outras solicitações	147
Armazenamento físico	149	Armazenamento em camadas
149	Alocação de partições	151
Gerenciamento de arquivos	152	Formato de arquivo
153	Índices	155
Compactação	156	Como funciona a compactação
156	Eventos excluídos	158
Quando os tópicos são compactados?	159	Resumo
159		

7. Entrega confiável de dados.

.	161	Garantias de confiabilidade	162
Replicação	163	Configuração do Broker	164
Fator de replicação	165	Eleição de líder impuro	166
Mínimo de réplicas sincronizadas	167	Mantendo as réplicas sincronizadas	168
Persistindo no disco	169		
Usando produtores em um sistema confiável	169	Enviar confirmações	170
Configurando novas tentativas do produtor	171	Tratamento adicional de erros	171
Usando Consumidores em um Sistema Confiável	172	Propriedades Importantes de Configuração do Consumidor para Processamento Confiável	173

Índice | ix

Comprometendo explicitamente compensações em consumidores	174
Validando a confiabilidade do sistema	176
Validando a configuração	176
Validando aplicativos	177
Monitorando a confiabilidade na produção	178
Resumo	180

8. Semântica Exatamente Uma Vez.

.	181	Produtor Idempotente	182
Como funciona o Produtor Idempotente?	182	Limitações do Produtor Idempotente	184
Como faço para usar o Produtor Idempotente Kafka?	185	Transações	186
Casos de uso de transações	187	Que problemas as transações resolvem?	187
Como as transações são garantidas exatamente uma vez?	188	Quais problemas não são resolvidos por transações?	191
Como faço para usar transações?	193	IDs transacionais e cercas	196
Como funcionam as transações	198	Desempenho das transações	200
Resumo	201		

9. Construindo pipelines de dados.

.	203	Considerações ao construir pipelines de dados	204
Oportunidade	204	Confiabilidade	205
Taxa de transferência alta e variável	205	Formatos de dados	206
Transformações	207	Segurança	208
Tratamento de falhas	209	Acoplamento e agilidade	209
Quando usar o Kafka Connect versus produtor e consumidor	210	Kafka Connect	211
Executando o Kafka Connect	211	Exemplo de conector: arquivo Fonte e coletor de arquivos	214
Exemplo de conector: MySQL para Elasticsearch	216	Transformações de mensagem única	223
Uma análise			

x | Índice

Ingerir estruturas para outros armazenamentos de dados 229
Ferramentas ETL baseadas em GUI 229 Estruturas de processamento de
fluxo 230 Resumo 230

10. Espelhamento de dados entre clusters.

233 Casos de uso de
espelhamento entre clusters 234 Arquiteturas multicluster 235
Algumas realidades da comunicação entre datacenters 235 Arquitetura
Hub-and-Spoke 236 Arquitetura Ativo-Ativo 238 Arquitetura
Ativo-Standby 240 Stretch Clusters 246
MirrorMaker do Apache Kafka 247 Configurando o MirrorMaker 249
Topologia de replicação multicluster 251 Protegendo o MirrorMaker 252
Implantando o MirrorMaker na produção 253 Ajustando o MirrorMaker
257
Outras soluções de espelhamento entre clusters 259 Uber uReplicator
259 LinkedIn Brooklin 260 Soluções de espelhamento entre datacenters
confluentes 261 Resumo 263

11. Protegendo Kafka.

265 Bloqueando protocolos de
segurança Kafka 265 268 Autenticação 269
SSL 270 SASL 275 Reautenticação 286 Atualizações de segurança
sem tempo de inatividade 288
Criptografia 289 Criptografia ponta a ponta 289 Autorização 291
AclAuthorizer 292 Personalizando autorização 295 Considerações de
segurança 297 Auditoria 298 Protegendo o ZooKeeper 299

Índice | xii

SASL 299 SSL 300 Autorização 301
Protegendo a Plataforma 301 Proteção por Senha 301 Resumo 303

12. Administração Kafka.

305 Operações de tópico 305 Criando um
novo tópico 306 Listando todos os tópicos em um cluster 308 Descrevendo
detalhes do tópico 308 Adicionando partições 310 Reduzindo partições 311
Excluindo um tópico 311 Grupos de consumidores 312 Listando e
descrevendo grupos 312 Excluindo grupo 313 Gerenciamento de
deslocamento 314 Alterações de configuração dinâmica 315 Substituindo
configuração de tópico Padrões 315 Substituindo Padrões de Configuração
de Cliente e Usuário 317 Substituindo Padrões de Configuração do Broker
318 Descrevendo Substituições de Configuração 319 Removendo
Substituições de Configuração 319 Produzindo e Consumindo 320 Produtor
de Console 320 Consumidor de Console 322 Gerenciamento de Partição 326

Eleição de Réplica Preferencial	326
Alterando Réplicas de uma Partição	327
Dumping de Segmentos de Log	332
Verificação de plica	334
Outras ferramentas	334
Operações inseguras	335
Mover o Cluster Controller	335
Remover tópicos a eliminar	336
Eliminar tópicos manualmente	336
Resumo	337

xii | Índice

13. Monitorando Kafka.	339
Noções básicas de métricas	339
Onde estão as métricas?	339
Quais métricas eu preciso?	341
Verificações de integridade do aplicativo	343
Objetivos de nível de serviço	343
Definições de nível de serviço	343
Quais métricas constituem bons SLIs?	344
Usando SLOs em alertas	345
Métricas do corretor Kafka	346
Diagnosticando problemas de cluster	347
A arte das partições sub-replicadas	348
Métricas do corretor	354
Métricas de tópico e partição	364
Monitoramento de JVM	366
Monitoramento de sistema operacional	367
Registro em log	369
Monitoramento de cliente	370
Métricas do produtor	370
Métricas do consumidor	373
Cotas	376
g Monitoramento	377
Monitoramento de ponta a ponta	378
Resumo	378

14. Processamento de fluxo.	381
O que é processamento de fluxo?	382
Conceitos de Processamento de Fluxo	385
Topologia	385
Tempo	386
Estado	388
Dualidade de tabela de fluxo	389
Tempo Windows	390
Garantias de processamento	392
Padrões de projeto de processamento de fluxo	392
Processamento de evento único	392
Processamento com estado local	393
Processamento/reparticionamento multifásico	395
Processamento com pesquisa externa: Junção de tabela de fluxo	396
Junção de tabela-tabela	398
Junção de streaming	398

Índice | xiii

Eventos Fora de Sequência	399
Reprocessamento	400
Consultas Interativas	401
Fluxos Kafka por exemplo	402
Contagem de palavras	402
Estatísticas do mercado de ações	405
Enriquecimento ClickStream	408
Fluxos Kafka: Visão Geral da Arquitetura	410
Construindo uma Topologia	410
Otimizando uma Topologia	411
Testando uma Topologia	411
Dimensionando uma Topologia	412
Sobrevivendo a Falhas	415
Casos de uso de processamento de fluxo	416
Como escolher uma estrutura de processamento de fluxo	417
Resumo	419

UM. Instalando o Kafka em outros sistemas operacionais.

.....	421	B. Ferramentas
adicionais do Kafka.		
.....	427	Índice.
.....		433

XIV | Índice

Prefácio à segunda edição

A primeira edição do ***Kafka: o guia definitivo*** foi publicado há cinco anos. Na época, estimamos que o Apache Kafka era usado em 30% das empresas Fortune 500. Hoje, mais de 70% das empresas Fortune 500 usam Apache Kafka. Ainda é um dos projetos de código aberto mais populares do mundo e está no centro de um enorme ecossistema.

Por que toda essa excitação? Penso que é porque existe uma enorme lacuna na nossa infra-estrutura de dados. Tradicionalmente, o gerenciamento de dados girava em torno do armazenamento – os armazenamentos de

arquivos e bancos de dados que mantêm nossos dados seguros e nos permitem procurar a parte certa no momento certo. Enormes quantidades de energia intelectual e investimento comercial foram investidas nestes sistemas. Mas uma empresa moderna não é apenas um software com um banco de dados. Uma empresa moderna é um sistema incrivelmente complexo construído a partir de centenas ou até milhares de aplicativos personalizados, microsserviços, bancos de dados, camadas SaaS e plataformas analíticas. E cada vez mais, o problema que enfrentamos é como conectar tudo isso em uma empresa e fazer com que tudo funcione em conjunto em tempo real.

Este problema não se trata de gerenciar dados em repouso – trata-se de gerenciar dados em movimento. E bem no centro desse movimento está o Apache Kafka, que se tornou a base de fato para qualquer plataforma de dados em movimento.

Ao longo desta jornada, Kafka não permaneceu estático. O que começou como um log de commit básico também evoluiu: adicionando conectores e recursos de processamento de fluxo e reinventando sua própria arquitetura ao longo do caminho. A comunidade não apenas desenvolveu APIs, opções de configuração, métricas e ferramentas existentes para melhorar a usabilidade e a confiabilidade do Kafka, mas também introduziu uma nova API de administração programática, a próxima geração de replicação global e DR com MirrorMaker 2.0, um novo Protocolo de consenso baseado em Raft que permite executar o Kafka em um único executável e elasticidade real com suporte de armazenamento em camadas. Talvez o mais importante seja que tornamos o Kafka um acéfalo em casos de uso corporativo críticos, adicionando suporte para opções avançadas de segurança – autenticação, autorização e criptografia.

xv

À medida que o Kafka evolui, vemos os casos de uso evoluir também. Quando a primeira edição foi publicada, a maioria das instalações do Kafka ainda estavam em data centers locais tradicionais, usando scripts de implantação tradicionais. Os casos de uso mais populares foram ETL e mensagens; os casos de uso de processamento de fluxo ainda estavam dando os primeiros passos. Cinco anos depois, a maioria das instalações do Kafka estão na nuvem e muitas estão rodando no Kubernetes. ETL e mensagens ainda são populares, mas são acompanhados por microsserviços orientados a eventos, processamento de fluxo em tempo real, IoT, pipelines de aprendizado de máquina e centenas de casos de uso e padrões específicos do setor que vão desde processamento de sinistros em companhias de seguros até sistemas de negociação em bancos para ajudar a potencializar o jogo em tempo real e a personalização em videogames e serviços de streaming.

Mesmo que o Kafka se expanda para novos ambientes e casos de uso, escrever aplicativos que usem bem o Kafka e implementá-lo com confiança na produção requer aclimação à maneira única de pensar do Kafka. Este livro cobre tudo o que os desenvolvedores e SREs precisam para usar o

Kafka em todo o seu potencial, desde as APIs e configurações mais básicas até os recursos mais recentes e avançados. Abrange não apenas o que você pode fazer com Kafka e como fazê-lo, mas também o que não fazer e os antipadrões a serem evitados. Este livro pode ser um guia confiável para o mundo do Kafka, tanto para novos usuários quanto para profissionais experientes.

-Jay Kreps
Cofundador e CEO da Confluent

xvi | Prefácio à segunda edição

Prefácio à Primeira Edição

É um momento emocionante para Apache Kafka. Kafka está sendo usado por dezenas de milhares de organizações, incluindo mais de um terço das empresas Fortune 500. Está entre os projetos de código aberto de crescimento mais rápido e gerou um imenso ecossistema ao seu redor. Está no centro de um movimento em direção ao gerenciamento e processamento

de fluxos de dados.

Então de onde veio Kafka? Por que o construímos? E o que é exatamente?

Kafka começou como um sistema de infraestrutura interna que construímos no LinkedIn. Nossa observação foi muito simples: havia muitos bancos de dados e outros sistemas construídos para **loja** dados, mas o que faltava em nossa arquitetura era algo que nos ajudasse a lidar com o contínuo **fluxo** de dados. Antes de construir o Kafka, experimentamos todos os tipos de opções disponíveis no mercado, desde sistemas de mensagens até agregação de logs e ferramentas ETL, mas nenhuma delas nos deu o que queríamos.

Finalmente decidimos construir algo do zero. Nossa ideia era que, em vez de nos concentrarmos em manter pilhas de dados, como nossos bancos de dados relacionais, armazenamentos de valores-chave, índices de pesquisa ou caches, nos concentraríamos em tratar os dados como um fluxo em constante evolução e crescimento e construiríamos um sistema de dados - e na verdade, uma arquitetura de dados – orientada em torno dessa ideia.

Esta ideia revelou-se ainda mais amplamente aplicável do que esperávamos. Embora o Kafka tenha começado alimentando aplicativos em tempo real e fluxo de dados nos bastidores de uma rede social, agora você pode vê-lo no centro das arquiteturas de próxima geração em todos os setores imagináveis. Os grandes retalhistas estão a reformular os seus processos empresariais fundamentais em torno de fluxos de dados contínuos, as empresas automóveis estão a recolher e a processar fluxos de dados em tempo real de automóveis ligados à Internet e os bancos também estão a repensar os seus processos e sistemas fundamentais em torno de Kafka.

Então, do que se trata essa coisa de Kafka? Como ele se compara aos sistemas que você já conhece e usa?

xvii

Passamos a pensar em Kafka como um **plataforma de streaming**, um sistema que permite publicar e assinar fluxos de dados, armazená-los e processá-los, e é exatamente para isso que o Apache Kafka foi criado. Acostumar-se com essa maneira de pensar sobre os dados pode ser um pouco diferente do que você está acostumado, mas acaba sendo uma abstração incrivelmente poderosa para a construção de aplicativos e arquiteturas. Kafka é frequentemente comparado a algumas categorias de tecnologia existentes: sistemas de mensagens empresariais, sistemas de big data como Hadoop e integração de dados ou ferramentas ETL. Cada uma dessas comparações tem alguma validade, mas também fica um pouco aquém.

Kafka é como um sistema de mensagens, pois permite publicar e assinar fluxos de mensagens. Dessa forma, é semelhante a produtos como ActiveMQ, RabbitMQ, MQSeries da IBM e outros produtos. Mas mesmo com essas semelhanças, Kafka tem uma série de diferenças fundamentais em relação aos sistemas de mensagens tradicionais que o tornam um tipo

totalmente diferente de animal. Aqui estão as três grandes diferenças: primeiro, ele funciona como um sistema distribuído moderno que funciona como um cluster e pode ser dimensionado para lidar com todos os aplicativos, mesmo nas empresas mais massivas. Em vez de executar dezenas de corretores de mensagens individuais, conectados manualmente a diferentes aplicativos, isso permite que você tenha uma plataforma central que pode ser dimensionada de forma elástica para lidar com todos os fluxos de dados em uma empresa. Em segundo lugar, o Kafka é um verdadeiro sistema de armazenamento criado para armazenar dados pelo tempo que você desejar. Isso traz enormes vantagens em usá-lo como camada de conexão, pois fornece garantias reais de entrega – seus dados são replicados, persistentes e podem ser mantidos pelo tempo que você desejar. Finalmente, o mundo do processamento de fluxo aumenta significativamente o nível de abstração. Os sistemas de mensagens geralmente apenas distribuem mensagens. Os recursos de processamento de fluxo no Kafka permitem calcular fluxos e conjuntos de dados derivados dinamicamente de seus fluxos com muito menos código. Essas diferenças tornam Kafka tão único que realmente não faz sentido pensar nele como “mais uma fila”.

Outra visão do Kafka – e uma de nossas lentes motivadoras ao projetá-lo e construí-lo – foi pensá-lo como uma espécie de versão em tempo real do Hadoop. O Hadoop permite armazenar e processar periodicamente dados de arquivos em grande escala. Kafka permite armazenar e processar continuamente fluxos de dados, também em grande escala. No nível técnico, há definitivamente semelhanças, e muitas pessoas veem a área emergente do processamento de fluxo como um superconjunto do tipo de processamento em lote que as pessoas fizeram com o Hadoop e suas diversas camadas de processamento. O que essa comparação deixa passar é que os casos de uso que o processamento contínuo e de baixa latência abre são bastante diferentes daqueles que ocorrem naturalmente em um sistema de processamento em lote. Enquanto o Hadoop e os aplicativos de análise de big data são direcionados, muitas vezes no espaço de armazenamento de dados, a natureza de baixa latência do Kafka o torna aplicável para o tipo de aplicativos principais que alimentam diretamente um negócio. Isso faz sentido: os eventos em uma empresa acontecem o tempo todo, e a capacidade de reagir a eles à medida que ocorrem torna muito mais fácil criar serviços que impulsionam diretamente a operação da empresa, retroalimentam as experiências dos clientes e assim por diante.

XVIII | Prefácio à Primeira Edição

A área final com a qual Kafka é comparado é ETL ou ferramentas de integração de dados. Afinal, essas ferramentas movimentam dados e Kafka movimenta dados. Há alguma validade nisso também, mas acho que a principal diferença é que Kafka inverteu o problema. Em vez de uma ferramenta para extrair dados de um sistema e inseri-los em outro, Kafka é uma plataforma orientada para fluxos de eventos em tempo real. Isso significa que ele não apenas pode conectar aplicativos e sistemas de dados

prontos para uso, mas também pode alimentar aplicativos personalizados criados para serem acionados a partir desses mesmos fluxos de dados. Achemos que essa arquitetura centrada em fluxos de eventos é algo realmente importante. De certa forma, estes fluxos de dados são o aspecto mais central de uma empresa digital moderna, tão importantes como os fluxos de caixa que veríamos numa demonstração financeira.

A capacidade de combinar essas três áreas – reunir todos os fluxos de dados em todos os casos de uso – é o que torna a ideia de uma plataforma de streaming tão atraente para as pessoas.

Ainda assim, tudo isso é um pouco diferente, e aprender como pensar e construir aplicações orientadas em torno de fluxos contínuos de dados é uma grande mudança de mentalidade se você vem do mundo dos aplicativos do tipo solicitação/resposta e bancos de dados relacionais. Este livro é absolutamente a melhor maneira de aprender sobre Kafka, desde aspectos internos até APIs, escrito por algumas das pessoas que melhor o conhecem. Espero que você goste de ler tanto quanto eu!

-Jay Kreps
Cofundador e CEO da Confluent

Prefácio à Primeira Edição | XIX

Prefácio

O maior elogio que você pode fazer a um autor de um livro técnico é “Este é o livro que eu gostaria de ter quando comecei neste assunto”. Este é o objetivo que estabelecemos para nós mesmos quando começamos a escrever este livro. Analisamos nossa experiência escrevendo Kafka, executando Kafka em produção e ajudando muitas empresas a usar Kafka para construir arquiteturas de software e gerenciar seus pipelines de dados, e nos perguntamos: “Quais são as coisas mais úteis que podemos compartilhar com novos usuários para levar em consideração? do iniciante ao especialista?” Este livro é um reflexo do trabalho que fazemos todos os dias: executar o Apache Kafka e ajudar outras pessoas a usá-lo da melhor maneira.

Incluímos o que acreditamos que você precisa saber para executar com êxito o Apache Kafka em produção e construir aplicativos robustos e de alto desempenho com base nele. Destacamos os casos de uso populares: barramentos de mensagens para microsserviços orientados a eventos, aplicativos de processamento de fluxo e pipelines de dados em grande escala. Também nos concentramos em tornar o livro geral e abrangente o suficiente para que seja útil para qualquer pessoa que use o Kafka, independentemente do caso de uso ou da arquitetura. Cobrimos questões práticas, como instalar e configurar o Kafka e como usar as APIs do Kafka, e também dedicamos espaço aos princípios de design e garantias de confiabilidade do Kafka, e exploramos vários detalhes encantadores da arquitetura do Kafka: o protocolo de replicação, controlador, e camada de armazenamento. Acreditamos que o conhecimento do design e dos componentes internos do Kafka não é apenas uma leitura divertida para aqueles interessados em sistemas distribuídos, mas também é incrivelmente útil para aqueles que buscam tomar decisões informadas ao implantar o Kafka em aplicações de produção e design que usam o Kafka. Quanto melhor você entender como o Kafka funciona, mais poderá tomar decisões informadas sobre as muitas compensações envolvidas na engenharia.

Um dos problemas da engenharia de software é que sempre há mais de uma maneira de fazer qualquer coisa. Plataformas como Apache Kafka oferecem bastante flexibilidade, o que é ótimo para especialistas, mas proporciona uma curva de aprendizado acentuada para iniciantes. Muitas vezes, o Apache Kafka informa como usar um recurso, mas não por que você deveria ou não usá-lo. Sempre que possível, procuramos esclarecer as escolhas existentes, as

compensações envolvidas e quando você deve ou não usar as diferentes opções apresentadas pelo Apache Kafka.

Quem deveria ler este livro

Kafka: o guia definitivo foi escrito para engenheiros de software que desenvolvem aplicativos que usam APIs do Kafka e para engenheiros de produção (também chamados de SREs, DevOps ou administradores de sistemas) que instalam, configuram, ajustam e monitoram o Kafka em produção. Também escrevemos o livro pensando em arquitetos e engenheiros de dados – aqueles responsáveis por projetar e construir toda a infraestrutura de dados de uma organização. Alguns dos capítulos, especialmente capítulos 3, 4, e 14, são voltados para desenvolvedores Java. Esses capítulos pressupõem que o leitor esteja familiarizado com os fundamentos da linguagem de programação Java, incluindo tópicos como tratamento de exceções e simultaneidade. Outros capítulos, especialmente capítulos 2, 10, 12, e 13, suponham que o leitor tenha alguma experiência em execução do Linux e alguma familiaridade com armazenamento e configuração de rede no Linux. O restante do livro discute Kafka e arquiteturas de software em termos mais gerais e não pressupõe conhecimento especial.

Outra categoria de pessoas que podem achar este livro interessante são os gestores e arquitetos que não trabalham diretamente com Kafka, mas trabalham com as pessoas que o fazem. É igualmente importante que compreendam as garantias que Kafka oferece e as compensações que os seus funcionários e colegas de trabalho terão de fazer ao construir sistemas baseados em Kafka. O livro pode fornecer munção para gerentes que desejam treinar sua equipe no Apache Kafka ou garantir que suas equipes saibam o que precisam saber.

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

italico

Indica novos termos, URLs, endereços de e-mail, nomes de arquivos e extensões de arquivos.

Largura constante

Usado para listagens de programas, bem como dentro de parágrafos para se referir a elementos do programa, como nomes de variáveis ou funções, bancos de dados, tipos de dados, variáveis de ambiente, instruções e palavras-chave.

Largura constante em negrito

Mostra comandos ou outros textos que devem ser digitados literalmente pelo usuário.

Largura constante em itálico

Mostra o texto que deve ser substituído por valores fornecidos pelo usuário ou por valores determinados pelo contexto.



Este elemento significa uma dica ou sugestão.



Este elemento significa uma nota geral.



Este elemento indica um aviso ou cuidado.

Usando exemplos de código

Se você tiver alguma dúvida técnica ou problema ao usar os exemplos de código, envie um e-mail para

bookquestions@oreilly.com.

Este livro está aqui para ajudá-lo a realizar seu trabalho. Em geral, se um código de exemplo for oferecido com este livro, você poderá usá-lo em seus programas e documentação. Você não precisa entrar em contato conosco para obter permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que utilize vários trechos de código deste livro não requer permissão. Vender ou distribuir exemplos de livros da O'Reilly requer permissão. Responder a uma pergunta citando este livro e citando código de exemplo não requer permissão. Incorporar uma quantidade significativa de código de exemplo deste livro na documentação do seu produto requer permissão.

Agradecemos, mas não exigimos, atribuição. Uma atribuição geralmente inclui título, autor, editora e ISBN. Por exemplo: "***Kafka: o guia definitivo*** por Gwen Shapira, Todd Palino, Rajini Sivaram e Krit Petty (O'Reilly). Copyright 2021 Chen Shapira, Todd Palino, Rajini Sivaram e Krit Petty, 978-1-491-93616-0."

Se você acha que o uso de exemplos de código está fora do uso justo ou da permissão dada acima, sinta-se à vontade para entrar em contato conosco em **permissões@oreilly.com**.

Aprendizagem on-line O'Reilly

O'REILLY

Há mais de 40 anos, **O'Reilly Media** forneceu tecnologia treinamento em ciência e negócios, conhecimento e visão para ajudar as empresas têm sucesso.

Nossa rede exclusiva de especialistas e inovadores compartilha seu conhecimento e experiência por meio de livros, artigos e nossa plataforma de aprendizagem on-line. A plataforma de aprendizagem on-line da O'Reilly oferece acesso sob demanda a cursos de treinamento ao vivo, caminhos de aprendizagem aprofundados, ambientes de codificação interativos e uma vasta coleção de textos e vídeos da O'Reilly e de mais de 200 outros editores. Para mais informações, visite **<http://oreilly.com>**.

Como entrar em contato conosco

Por favor, envie comentários e perguntas sobre este livro à editora:

O'Reilly Media, Inc.
Rodovia Gravenstein 1005 Norte
Sebastopol, CA 95472
800-998-9938 (nos Estados Unidos ou Canadá)
707-829-0515 (internacional ou local)
707-829-0104 (fax)

Temos uma página web para este livro, onde listamos erratas, exemplos e qualquer informação adicional. Você pode acessar esta página em **<https://oreil.ly/kafka-tdg2>**.

E-mail **bookquestions@oreilly.com** para comentar ou fazer perguntas técnicas sobre este livro.

Para novidades e informações sobre nossos livros e cursos, acesse

<http://oreilly.com>. Encontre-nos no Facebook:

<http://facebook.com/oreilly>

Siga-nos no Twitter: **<http://twitter.com/oreillymedia>**

Assista-nos no YouTube: **<http://youtube.com/oreillymedia>**

Agradecimentos

Gostaríamos de agradecer aos muitos colaboradores do Apache Kafka e seu ecossistema. Sem o trabalho deles, este livro não existiria. Agradecimentos especiais a Jay Kreps, Neha Narkhede e Jun Rao, bem como a seus colegas e à liderança do LinkedIn, por cocriar o Kafka e contribuir com ele para a Apache Software Foundation.

xxiv | Prefácio

Muitas pessoas forneceram feedback valioso sobre as primeiras versões do livro, e agradecemos seu tempo e experiência: Apurva Mehta, Arseniy Tashoyan, Dylan Scott, Ewen Cheslack-Postava, Grant Henke, Ismael Juma, James Cheng, Jason Gustafson, Jeff Holoman, Joel Koshy, Jonathan Seidman, Jun Rao, Matthias Sax, Michael Noll, Paolo Castagna e Jesse Anderson. Também queremos agradecer aos muitos leitores que deixaram comentários e feedback através do site de feedback preliminar.

Muitos revisores nos ajudaram e melhoraram muito a qualidade deste livro, portanto, quaisquer erros deixados serão nossos.

Gostaríamos de agradecer à nossa editora da primeira edição da O'Reilly, Shannon Cutt, por seu incentivo e paciência, e por estar muito mais atualizada do que nós. Nossos editores da segunda edição, Jess Haberman e Gary O'Brien, nos mantiveram no caminho certo em meio aos desafios globais. Trabalhar com O'Reilly é uma ótima experiência para um autor – o suporte que eles fornecem, desde ferramentas até sessões de autógrafos, é incomparável. Somos gratos a todos os envolvidos em fazer isso acontecer e agradecemos sua escolha de trabalhar conosco.

E gostaríamos de agradecer aos nossos gestores e colegas por nos permitirem e encorajarem enquanto escrevíamos o livro.

Gwen quer agradecer ao marido, Omer Shapira, pelo apoio e paciência durante os muitos meses que passou escrevendo mais um livro; seus gatos, Luke e Lea, por serem fofinhos; e ao pai, Lior Shapira, por ensiná-la a sempre dizer sim às oportunidades, mesmo quando elas parecem assustadoras.

Todd não estaria em lugar nenhum sem sua esposa, Marcy, e suas filhas, Bella e Kaylee, sempre atrás dele. O apoio deles por todo o tempo extra escrevendo e pelas longas horas correndo para clarear a cabeça o faz continuar.

Rajini gostaria de agradecer ao marido, Manjunath, e ao filho, Tarun, pelo apoio e incentivo inabaláveis, por passarem os fins de semana revisando os primeiros rascunhos e por sempre estarem ao seu lado.

Krit compartilha seu amor e gratidão com sua esposa, Cecília, e dois filhos, Lucas e Lizabeth. O amor e o apoio deles tornam cada dia uma alegria, e ele não seria capaz de seguir suas paixões sem eles. Ele também quer agradecer à sua mãe, Cindy Petty, por incutir em Krit o desejo de ser sempre a melhor versão de si mesmo.

CAPÍTULO 1

Conheça Kafka

Toda empresa é alimentada por dados. Nós absorvemos informações, analisamos, manipulamos e criamos mais como resultado. Cada aplicativo cria dados, sejam mensagens de log, métricas, atividade do usuário, mensagens enviadas ou qualquer outra coisa. Cada byte de dados tem uma história para contar, algo importante que informará o próximo passo a ser feito. Para saber o que é isso, precisamos levar os dados de onde são criados até onde podem ser analisados. Vemos isso todos os dias em sites como o Amazon, onde nossos cliques em itens de nosso interesse se transformam em recomendações que nos são mostradas um pouco mais tarde.

Quanto mais rápido pudermos fazer isso, mais ágeis e receptivas nossas organizações poderão ser. Quanto menos esforço gastarmos na movimentação de dados, mais poderemos nos concentrar no negócio principal em questão. É por isso que o pipeline é um componente crítico na empresa orientada por dados. A forma como movemos os dados torna-se quase tão importante quanto os próprios dados.

Sempre que os cientistas discordam, é porque não temos dados suficientes. Então poderemos chegar a um acordo sobre que tipo de dados obter; obtemos os dados; e os dados resolvem o problema. Ou eu estou certo, ou você está certo, ou nós dois estamos errados. E seguimos em frente.

—Neil de Grasse Tyson

Publicar/assinar mensagens

Antes de discutir as especificidades do Apache Kafka, é importante

entendermos o conceito de mensagens de publicação/assinatura e por que ele é um componente crítico de aplicativos orientados a dados. **Mensagens de publicação/assinatura (pub/sub)** é um padrão caracterizado pelo remetente (publicador) de um dado (mensagem) não o direcionar especificamente a um destinatário. Em vez disso, o editor classifica a mensagem de alguma forma e o receptor (assinante) se inscreve para receber certas classes de mensagens. Pub/assinatura

1

os sistemas geralmente têm um intermediário, um ponto central onde as mensagens são publicadas, para facilitar esse padrão.

Como tudo começa

Muitos casos de uso para publicação/assinatura começam da mesma maneira: com uma fila de mensagens simples ou um canal de comunicação entre processos. Por exemplo, você cria um aplicativo que precisa enviar informações de monitoramento para algum lugar, então você abre uma conexão direta do seu aplicativo para um aplicativo que exibe suas métricas em um painel e envia métricas por meio dessa conexão, como visto em [Figura 1-1](#).

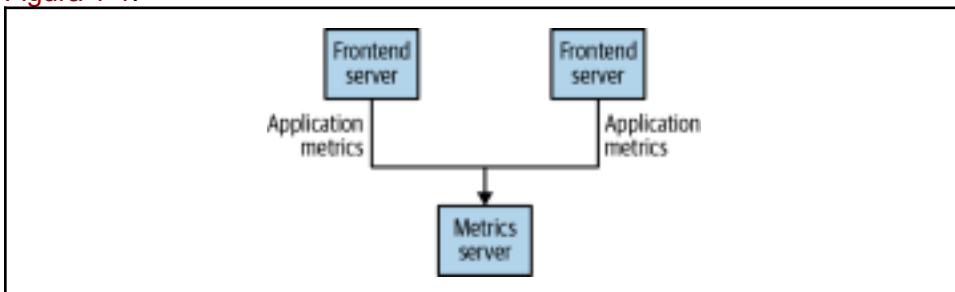


Figura 1-1. Um editor de métricas único e direto

Esta é uma solução simples para um problema simples que funciona quando você está começando a monitorar. Em pouco tempo, você decide que gostaria de analisar suas métricas em um prazo mais longo, e isso não funciona bem no painel. Você inicia um novo serviço que pode receber métricas, armazená-las e analisá-las. Para oferecer suporte a isso, você modifica seu aplicativo para gravar métricas em ambos os sistemas. Até agora você tem mais três aplicações que estão gerando métricas, e todas elas fazem as mesmas conexões com esses dois serviços. Seu colega de trabalho acha que seria uma boa ideia fazer uma pesquisa ativa dos serviços para alertas também, então você adiciona um servidor em cada um dos aplicativos para fornecer métricas mediante solicitação. Depois de um tempo, você terá mais aplicativos que usam esses servidores para obter métricas individuais e usá-los para diversos fins. Essa arquitetura pode

parecer muito com **Figura 1-2**, com conexões ainda mais difíceis de rastrear.

2 | Capítulo 1: Conheça Kafka

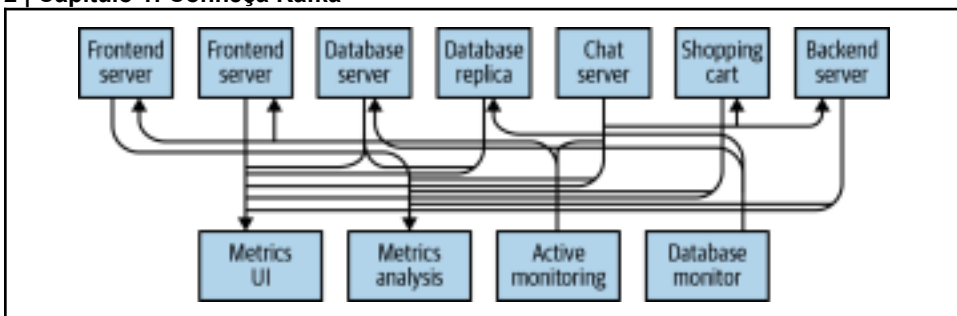


Figura 1-2. Muitos editores de métricas, usando conexões diretas

A dívida técnica acumulada aqui é óbvia, então você decide pagar parte dela. Você configura um único aplicativo que recebe métricas de todos os aplicativos existentes e fornece um servidor para consultar essas métricas para qualquer sistema que precise delas. Isso reduz a complexidade da arquitetura a algo semelhante a **Figura 1-3**. Parabéns, você construiu um sistema de mensagens de publicação/assinatura!

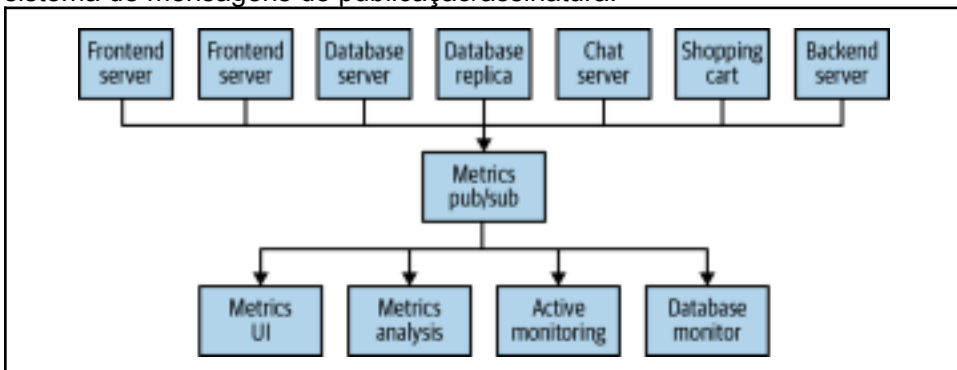


Figura 1-3. Um sistema de publicação/assinatura de métricas

Sistemas de filas individuais

Ao mesmo tempo que você trava essa guerra com métricas, um de seus colegas de trabalho tem feito um trabalho semelhante com mensagens de log. Outro tem trabalhado no rastreamento do comportamento do usuário no site frontend e no fornecimento dessas informações aos desenvolvedores que estão trabalhando em aprendizado de máquina, além de criar alguns relatórios para gerenciamento. Todos vocês seguiram um caminho semelhante de construção de sistemas que dissociam os editores da informação dos assinantes dessa informação. **Figura 1-4** mostra tal infraestrutura, com três sistemas pub/sub separados.

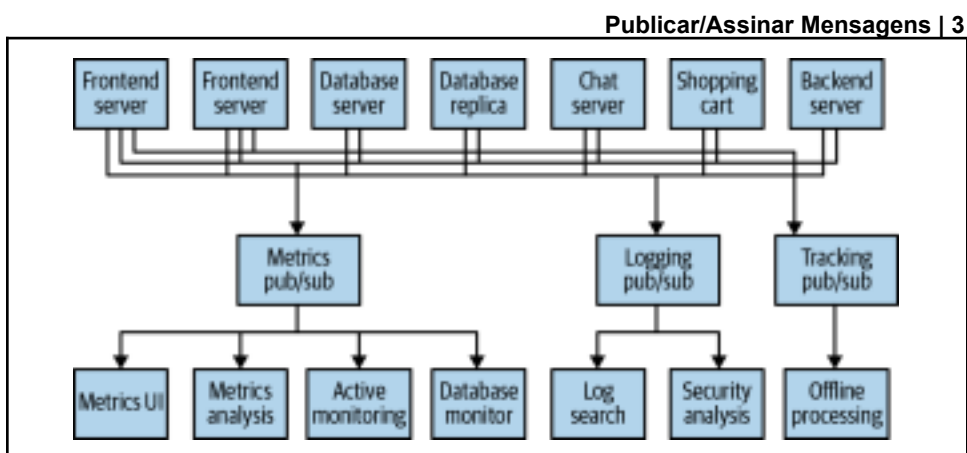


Figura 1-4. Vários sistemas de publicação/assinatura

Isto é certamente muito melhor do que utilizar conexões ponto a ponto (como em **Figura 1-2**), mas há muita duplicação. Sua empresa mantém vários sistemas para enfileiramento de dados, todos com bugs e limitações individuais. Você também sabe que em breve haverá mais casos de uso de mensagens. O que você gostaria de ter é um sistema único centralizado que permita a publicação de tipos genéricos de dados, que crescerão à medida que seu negócio crescer.

Entra Kafka

Apache Kafka foi desenvolvido como um sistema de mensagens de publicação/assinatura projetado para resolver esse problema. Muitas vezes é descrito como um “log de commit distribuído” ou, mais recentemente, como uma “plataforma de distribuição de streaming”. Um sistema de arquivos ou

log de confirmação de banco de dados é projetado para fornecer um registro durável de todas as transações para que possam ser reproduzidas para construir consistentemente o estado de um sistema. Da mesma forma, os dados dentro do Kafka são armazenados de forma durável, em ordem, e podem ser lidos de forma determinística. Além disso, os dados podem ser distribuídos dentro do sistema para fornecer proteções adicionais contra falhas, bem como oportunidades significativas para dimensionar o desempenho.

Mensagens e lotes

A unidade de dados dentro do Kafka é chamada de **mensagem**. Se você estiver abordando o Kafka com base em um banco de dados, poderá pensar nisso como semelhante a um **linka** ou um **registro**. Uma mensagem é simplesmente uma matriz de bytes no que diz respeito a Kafka, portanto, os dados contidos nela não possuem um formato ou significado específico para Kafka. Uma mensagem pode ter uma parte opcional de metadados, que é chamada de **chave**. A chave também é uma matriz de bytes e, assim como a mensagem, não tem nenhum significado específico para Kafka. As chaves são usadas quando as mensagens devem ser gravadas em partições de maneira mais controlada. O esquema mais simples é gerar um hash consistente da chave e então selecionar o número da partição para esse

4 | Capítulo 1: Conheça Kafka

mensagem tomando o resultado do módulo hash o número total de partições no tópico. Isso garante que as mensagens com a mesma chave sejam sempre gravadas na mesma partição (desde que a contagem de partições não mude).

Para maior eficiência, as mensagens são gravadas no Kafka em lotes. UM **lote** é apenas uma coleção de mensagens, todas sendo produzidas para o mesmo tópico e partição. Uma viagem de ida e volta individual pela rede para cada mensagem resultaria em sobrecarga excessiva, e a coleta de mensagens em um lote reduz isso. É claro que isso é uma compensação entre latência e taxa de transferência: quanto maiores os lotes, mais mensagens podem ser tratadas por unidade de tempo, mas mais tempo leva para uma mensagem individual se propagar. Os lotes também são normalmente compactados, proporcionando transferência e armazenamento de dados mais eficientes ao custo de algum poder de processamento. Tanto as chaves quanto os lotes são discutidos com mais detalhes em [Capítulo 3](#).

Esquemas

Embora as mensagens sejam matrizes de bytes opacas para o próprio Kafka, é recomendado que uma estrutura ou esquema adicional seja imposta ao conteúdo da mensagem para que ela possa ser facilmente compreendida. Existem muitas opções disponíveis para mensagens **esquema**, dependendo das necessidades individuais da sua aplicação. Sistemas simplistas, como JavaScript Object Notation (JSON) e

Extensible Markup Language (XML), são fáceis de usar e legíveis por humanos. No entanto, eles carecem de recursos como manipulação robusta de tipos e compatibilidade entre versões de esquema. Muitos desenvolvedores Kafka preferem o uso do Apache Avro, que é uma estrutura de serialização desenvolvida originalmente para Hadoop. Avro fornece um formato de serialização compacto, esquemas que são separados das cargas úteis da mensagem e que não exigem a geração de código quando eles mudam, e tipagem de dados forte e evolução de esquema, com compatibilidade retroativa e futura.

Um formato de dados consistente é importante no Kafka, pois permite desacoplar a escrita e a leitura de mensagens. Quando essas tarefas estão fortemente acopladas, os aplicativos que assinam mensagens devem ser atualizados para lidar com o novo formato de dados, em paralelo com o formato antigo. Só então os aplicativos que publicam as mensagens poderão ser atualizados para utilizar o novo formato. Ao usar esquemas bem definidos e armazená-los em um repositório comum, as mensagens no Kafka podem ser compreendidas sem coordenação. Esquemas e serialização são abordados com mais detalhes em [Capítulo 3](#).

Tópicos e partições

As mensagens no Kafka são categorizadas em **tópicos**. As analogias mais próximas para um tópico são uma tabela de banco de dados ou uma pasta em um sistema de arquivos. Os tópicos também são divididos em vários **partições**. Voltando à descrição do “log de commit”, uma partição é um único log. As mensagens são gravadas nele apenas como anexos e lidas em ordem, do início ao fim. Observe que como um tópico normalmente possui múltiplas partições, não há garantia de ordenação de mensagens em todo o tópico, apenas dentro de uma única partição.

Entra Kafka | 5

partição. [Figura 1-5](#) mostra um tópico com quatro partições, com escritas anexadas ao final de cada uma. As partições também são a forma como o Kafka fornece redundância e escalabilidade. Cada partição pode ser hospedada em um servidor diferente, o que significa que um único tópico pode ser dimensionado horizontalmente em vários servidores para fornecer desempenho muito além da capacidade de um único servidor. Além disso, as partições podem ser replicadas, de forma que diferentes servidores armazenem uma cópia da mesma partição caso um servidor falhe.

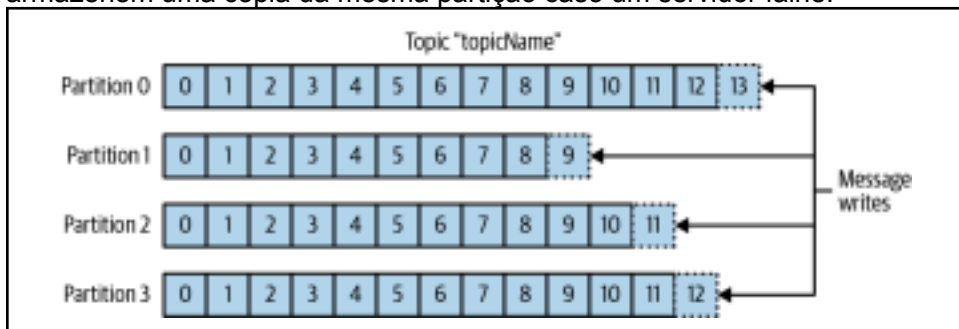


Figura 1-5. Representação de um tópico com múltiplas partições

O termo **Fluxo** é frequentemente usado ao discutir dados em sistemas como o Kafka. Na maioria das vezes, um fluxo é considerado um único tópico de dados, independentemente do número de partições. Isto representa um único fluxo de dados que passa dos produtores para os consumidores. Essa forma de se referir a mensagens é mais comum quando se discute o processamento de fluxo, que ocorre quando estruturas – algumas das quais são Kafka Streams, Apache Samza e Storm – operam nas mensagens em tempo real. Este método de operação pode ser comparado à forma como as estruturas offline, nomeadamente o Hadoop, são projetadas para trabalhar com dados em massa posteriormente. Uma visão geral do processamento de fluxo é fornecida em [Capítulo 14](#).

Produtores e Consumidores

Os clientes Kafka são usuários do sistema e existem dois tipos básicos: produtores e consumidores. Há também APIs de cliente avançadas: API Kafka Connect para integração de dados e Kafka Streams para processamento de stream. Os clientes avançados usam produtores e consumidores como blocos de construção e fornecem funcionalidade de nível superior.

Produtores criar novas mensagens. Em outros sistemas de publicação/assinatura, eles podem ser chamados **editores** ou **escritores**. Uma mensagem será produzida para um tópico específico. Por padrão, o produtor equilibrará as mensagens uniformemente em todas as partições de um tópico. Em alguns casos, o produtor direcionará mensagens para partições específicas. Isso normalmente é feito usando a chave da mensagem e um particionador que irá gerar um hash da chave e mapeá-lo para uma partição específica. Isto garante que todas as mensagens produzidas com uma determinada chave serão gravadas na mesma partição. O produtor também poderia usar um particionador personalizado que

6 | Capítulo 1: Conheça Kafka

segue outras regras de negócios para mapear mensagens para partições.

Os produtores são abordados com mais detalhes em [Capítulo 3](#).

Consumidores ler mensagens. Em outros sistemas de publicação/assinatura, esses clientes podem ser chamados **assinantes** ou **leitores**. O consumidor assina um ou mais tópicos e lê as mensagens na ordem em que foram produzidas para cada partição. O consumidor acompanha quais mensagens já consumiu, acompanhando o deslocamento das mensagens. O **desvio**—um valor inteiro que aumenta continuamente—é outro metadado que Kafka adiciona a cada mensagem à medida que ela é produzida. Cada mensagem em uma

determinada partição possui um deslocamento exclusivo e a mensagem seguinte possui um deslocamento maior (embora não necessariamente monotonicamente maior). Ao armazenar o próximo deslocamento possível para cada partição, normalmente no próprio Kafka, um consumidor pode parar e reiniciar sem perder seu lugar.

Os consumidores trabalham como parte de um **grupo de consumidores**, que é um ou mais consumidores que trabalham juntos para consumir um tópico. O grupo garante que cada partição seja consumida apenas por um membro. Em [Figura 1-6](#), existem três consumidores em um único grupo consumindo um tópico. Dois dos consumidores estão trabalhando em uma partição cada, enquanto o terceiro consumidor está trabalhando em duas partições. O mapeamento de um consumidor para uma partição é frequentemente chamado **propriedade** da partição pelo consumidor.

Dessa forma, os consumidores podem escalar horizontalmente para consumir tópicos com um grande número de mensagens. Além disso, se um único consumidor falhar, os membros restantes do grupo reatribuirão as partições que estão sendo consumidas para assumir o controle do membro ausente. Os consumidores e grupos de consumidores são discutidos mais detalhadamente em [Capítulo 4](#).

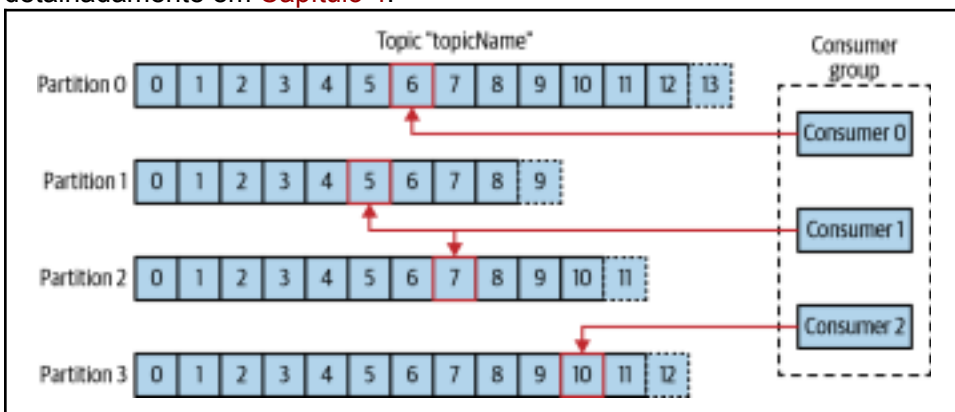


Figura 1-6. Um grupo de consumidores lendo um tópico

Corretores e Clusters

Um único servidor Kafka é chamado de **corretor**. O intermediário recebe mensagens dos produtores, atribui compensações a eles e grava as mensagens no armazenamento em disco. Também atende consumidores,

respondendo a solicitações de busca de partições e respondendo com as mensagens que foram publicadas. Dependendo do hardware específico e de suas características de desempenho, um único agente pode facilmente lidar com milhares de partições e milhões de mensagens por segundo.

Os corretores Kafka são projetados para operar como parte de um **conjunto**. Dentro de um cluster de corretores, um corretor também funcionará como cluster **controlador** (eleito automaticamente entre os membros ativos do cluster). O controlador é responsável pelas operações administrativas, incluindo a atribuição de partições aos intermediários e o monitoramento de falhas do intermediário. Uma partição pertence a um único intermediário no cluster, e esse intermediário é chamado de **líder** da partição. Uma partição replicada (como visto em [Figura 1-7](#)) é atribuído a corretores adicionais, chamados **seguidores** da partição. A replicação fornece redundância de mensagens na partição, de modo que um dos seguidores possa assumir a liderança se houver uma falha do corretor. Todos os produtores devem se conectar ao líder para publicar mensagens, mas os consumidores podem buscar mensagens no líder ou em um dos seguidores. As operações de cluster, incluindo replicação de partição, são abordadas detalhadamente em [Capítulo 7](#).

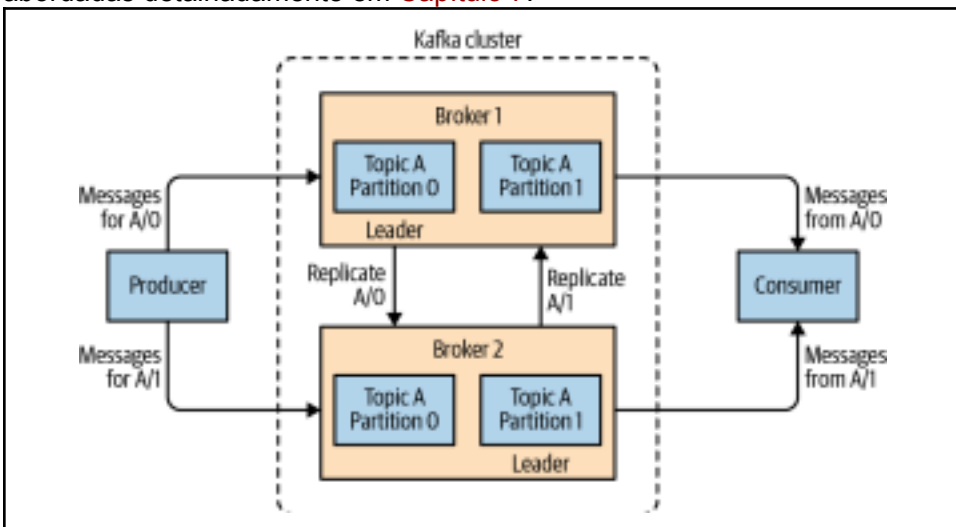


Figura 1-7. Replicação de partições em um cluster

Uma característica fundamental do Apache Kafka é a de **retenção**, que é o armazenamento durável de mensagens por um determinado período de tempo. Os brokers Kafka são configurados com uma configuração de retenção padrão para tópicos, retendo mensagens por algum período de tempo (por exemplo, 7 dias) ou até que a partição atinja um determinado tamanho em bytes (por exemplo, 1 GB). Quando esses limites são atingidos, as mensagens expiram e são excluídas. Desta forma, a configuração de retenção

define uma quantidade mínima de dados disponíveis a qualquer momento. Tópicos individuais também podem ser configurados com suas próprias configurações de retenção para que as mensagens sejam armazenadas apenas enquanto forem úteis. Por exemplo, um tópico de rastreamento pode ser retido por vários dias, enquanto as métricas do aplicativo podem ser retidas por apenas algumas horas. Os tópicos também podem ser configurados como **log compactado**, o que significa que Kafka reterá apenas a última mensagem produzida com uma chave específica. Isto pode ser útil para dados do tipo changelog, onde apenas a última atualização é interessante.

Vários clusters

À medida que as implantações do Kafka crescem, muitas vezes é vantajoso ter vários clusters. Existem vários motivos pelos quais isso pode ser útil:

- Segregação de tipos de dados
- Isolamento para requisitos de segurança
- Vários datacenters (recuperação de desastres)

Especialmente ao trabalhar com vários datacenters, muitas vezes é necessário que as mensagens sejam copiadas entre eles. Desta forma, os aplicativos online podem ter acesso à atividade do usuário em ambos os sites. Por exemplo, se um usuário alterar informações públicas em seu perfil, essa alteração precisará estar visível, independentemente do datacenter em que os resultados da pesquisa são exibidos. Ou os dados de monitoramento podem ser coletados de vários locais em um único local central onde os sistemas de análise e alerta estão hospedados. Os mecanismos de replicação nos clusters Kafka são projetados apenas para funcionar dentro de um único cluster, não entre vários clusters.

O projeto Kafka inclui uma ferramenta chamada **Criador de espelhos**, usado para replicar dados para outros clusters. Em sua essência, MirrorMaker é simplesmente um consumidor e produtor Kafka, vinculado por uma fila. As mensagens são consumidas de um cluster Kafka e produzidas para outro. **Figura 1-8** mostra um exemplo de arquitetura que usa MirrorMaker, agregando mensagens de dois clusters locais em um cluster agregado e depois copiando esse cluster para outros datacenters. A natureza simples do aplicativo desmente seu poder na criação de pipelines de dados sofisticados, que serão detalhados mais adiante. **Capítulo 9.**

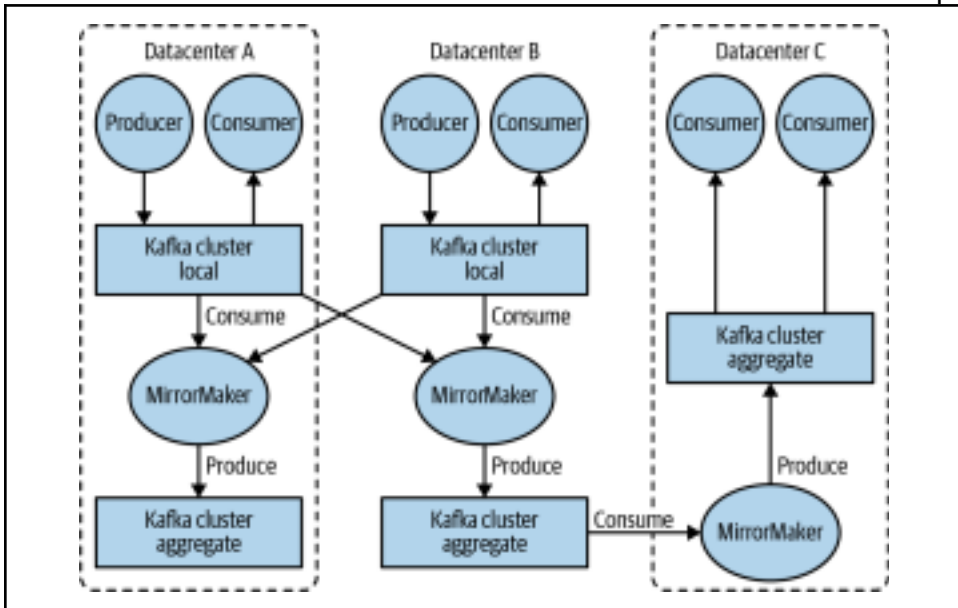


Figura 1-8. Arquitetura de vários datacenters

Por que Kafka?

Existem muitas opções para sistemas de mensagens de publicação/assinatura, então o que torna o Apache Kafka uma boa escolha?

Vários Produtores

Kafka é capaz de lidar perfeitamente com vários produtores, estejam esses clientes usando muitos tópicos ou o mesmo tópico. Isso torna o sistema ideal para agregar dados de muitos sistemas front-end e torná-los consistentes. Por exemplo, um site que fornece conteúdo aos usuários por meio de vários microsserviços pode ter um único tópico para visualizações de página no qual todos os serviços podem gravar usando um formato comum. Os aplicativos de consumo podem então receber um único fluxo de visualizações de página para todos os aplicativos no site sem precisar coordenar o consumo de vários tópicos, um para cada aplicativo.

Vários consumidores

Além de vários produtores, o Kafka foi projetado para que vários consumidores leiam qualquer fluxo único de mensagens sem interferir uns nos outros clientes. Isto contrasta com muitos sistemas de filas onde uma vez que uma mensagem é consumida por um cliente, ela não fica disponível para nenhum outro. Vários consumidores Kafka podem optar por operar como parte de um grupo e compartilhar um fluxo, garantindo que todo o grupo processe uma determinada mensagem apenas uma vez.

10 | Capítulo 1: Conheça Kafka

Retenção baseada em disco

O Kafka não apenas pode lidar com vários consumidores, mas a retenção durável de mensagens significa que os consumidores nem sempre precisam trabalhar em tempo real. As mensagens são gravadas em disco e armazenadas com regras de retenção configuráveis. Essas opções podem ser selecionadas por tópico, permitindo que diferentes fluxos de mensagens tenham diferentes quantidades de retenção, dependendo das necessidades do consumidor. A retenção durável significa que se um consumidor ficar para trás, seja devido a um processamento lento ou a uma explosão no tráfego, não há perigo de perda de dados. Isso também significa que a manutenção pode ser realizada nos consumidores, deixando os aplicativos off-line por um curto período de tempo, sem preocupação com o backup de mensagens no produtor ou com a perda. Os consumidores podem ser interrompidos e as mensagens serão retidas no Kafka. Isso permite que eles reiniciem e retomem o processamento de mensagens de onde pararam, sem perda de dados.

Escalável

A escalabilidade flexível do Kafka facilita o gerenciamento de qualquer quantidade de dados. Os usuários podem começar com um único corretor como prova de conceito, expandir para um pequeno cluster de desenvolvimento de três corretores e passar para a produção com um cluster maior de dezenas ou mesmo centenas de corretores que cresce ao longo do tempo à medida que os dados aumentam. As expansões podem ser realizadas enquanto o cluster está online, sem impacto na disponibilidade do sistema como um todo. Isso também significa que um cluster de vários corretores pode lidar com a falha de um corretor individual e continuar atendendo aos clientes. Clusters que precisam tolerar mais falhas simultâneas podem ser configurados com fatores de replicação mais elevados. A replicação é discutida com mais detalhes em [Capítulo 7](#).

Alto desempenho

Todos esses recursos se unem para tornar o Apache Kafka um sistema de mensagens de publicação/assinatura com excelente desempenho sob alta carga. Produtores, consumidores e corretores podem ser ampliados para

lidar com fluxos de mensagens muito grandes com facilidade. Isso pode ser feito enquanto ainda fornece latência de mensagem em subsegundos, desde a produção de uma mensagem até a disponibilidade para os consumidores.

Recursos da plataforma

O projeto principal do Apache Kafka também adicionou alguns recursos da plataforma de streaming que podem tornar muito mais fácil para os desenvolvedores realizarem tipos comuns de trabalho. Embora não sejam plataformas completas, que normalmente incluem um ambiente de tempo de execução estruturado como o YARN, esses recursos estão na forma de APIs e bibliotecas que fornecem uma base sólida para construção e flexibilidade quanto ao local onde podem ser executados. O Kafka Connect auxilia na tarefa de extrair dados de um sistema de dados de origem e enviá-los para o Kafka, ou extrair dados do Kafka e enviá-los para um sistema de dados coletor. Kafka Streams fornece uma biblioteca para desenvolver facilmente aplicativos de processamento de fluxo que são escaláveis e com falhas

Por que Kafka? | 11

tolerante. Conectar é discutido em [Capítulo 9](#), enquanto o Streams é abordado detalhadamente em [Capítulo 14](#).

O ecossistema de dados

Muitos aplicativos participam dos ambientes que construímos para processamento de dados. Definimos entradas na forma de aplicativos que criam dados ou os introduzem no sistema. Definimos resultados na forma de métricas, relatórios e outros produtos de dados. Criamos loops, com alguns componentes lendo dados do sistema, transformando-os usando dados de outras fontes e, em seguida, introduzindo-os de volta na infraestrutura de dados para serem usados em outro lugar. Isso é feito para vários tipos de dados, cada um com qualidades únicas de conteúdo, tamanho e uso.

Apache Kafka fornece o sistema circulatório para o ecossistema de dados, conforme mostrado em [Figura 1-9](#). Ele transporta mensagens entre os diversos membros da infraestrutura, fornecendo uma interface consistente para todos os clientes. Quando acoplados a um sistema para fornecer esquemas de mensagens, produtores e consumidores não necessitam mais de acoplamento forte ou conexões diretas de qualquer tipo. Os componentes podem ser adicionados e removidos à medida que os casos de negócios são criados e dissolvidos, e os produtores não precisam se preocupar com quem está usando os dados ou com o número de aplicativos consumidos.

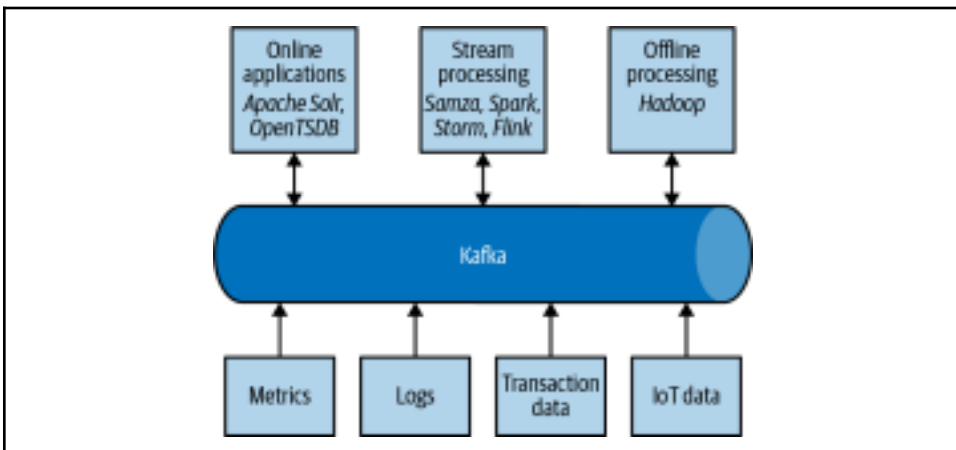


Figura 1-9. Um ecossistema de big data

Casos de uso

Rastreamento de atividades

O caso de uso original do Kafka, conforme foi projetado no LinkedIn, é o rastreamento da atividade do usuário. Os usuários de um site interagem com aplicativos frontend, que geram mensagens sobre as ações que o usuário está realizando. Podem ser informações passivas, como

12 | Capítulo 1: Conheça Kafka

visualizações de páginas e rastreamento de cliques, ou podem ser ações mais complexas, como informações que um usuário adiciona ao seu perfil. As mensagens são publicadas em um ou mais tópicos, que são então consumidos pelos aplicativos no backend. Esses aplicativos podem gerar relatórios, alimentar sistemas de aprendizado de máquina, atualizar resultados de pesquisa ou executar outras operações necessárias para fornecer uma experiência de usuário rica.

Mensagens

Kafka também é usado para mensagens, onde os aplicativos precisam enviar notificações (como e-mails) aos usuários. Esses aplicativos podem produzir mensagens sem a necessidade de se preocupar com a formatação ou com a forma como as mensagens serão realmente enviadas. Um único aplicativo pode então ler todas as mensagens a serem enviadas e tratá-las de forma consistente, incluindo:

- Formatar as mensagens (também conhecido como **decoração**) usando uma aparência comum
- Coletar várias mensagens em uma única notificação a ser enviada
- Aplicar as preferências de um usuário sobre como

ele deseja receber mensagens

Usar um único aplicativo para isso evita a necessidade de duplicar funcionalidades em vários aplicativos, além de permitir operações como agregação que de outra forma não seriam possíveis.

Métricas e registro

Kafka também é ideal para coletar métricas e logs de aplicativos e sistemas. Este é um caso de uso em que brilha a capacidade de ter vários aplicativos produzindo o mesmo tipo de mensagem. Os aplicativos publicam métricas regularmente em um tópico do Kafka, e essas métricas podem ser consumidas por sistemas para monitoramento e alertas. Eles também podem ser usados em um sistema offline como o Hadoop para realizar análises de longo prazo, como projeções de crescimento. As mensagens de log podem ser publicadas da mesma maneira e roteadas para sistemas de pesquisa de log dedicados, como Elasticsearch ou aplicativos de análise de segurança. Outro benefício adicional do Kafka é que quando o sistema de destino precisa mudar (por exemplo, é hora de atualizar o sistema de armazenamento de log), não há necessidade de alterar os aplicativos frontend ou os meios de agregação.

Registro de confirmação

Como o Kafka é baseado no conceito de log de commit, as alterações do banco de dados podem ser publicadas no Kafka e os aplicativos podem monitorar facilmente esse fluxo para receber atualizações ao vivo conforme elas acontecem. Esse fluxo de changelog também pode ser usado para replicar atualizações de banco de dados para um sistema remoto ou para consolidar alterações de vários aplicativos em uma única visualização de banco de dados. A retenção durável é útil aqui para fornecer um buffer para o changelog, o que significa que pode ser reproduzido no caso de falha do

O ecossistema de dados | 13

consumindo aplicativos. Como alternativa, os tópicos compactados em log podem ser usados para fornecer retenção mais longa, retendo apenas uma única alteração por chave.

Processamento de fluxo

Outra área que oferece vários tipos de aplicativos é o processamento de fluxo. Embora quase todo o uso do Kafka possa ser considerado como processamento de fluxo, o termo é normalmente usado para se referir a aplicativos que fornecem funcionalidade semelhante para mapear/reduzir o processamento no Hadoop. O Hadoop geralmente depende da agregação de dados durante um longo período de tempo, seja horas ou dias. O processamento de fluxo opera em dados em tempo real, tão rapidamente quanto as mensagens são produzidas. As estruturas de fluxo permitem que os usuários escrevam pequenas aplicações para operar em mensagens Kafka, executando tarefas como contagem de métricas, particionamento de mensagens para processamento eficiente por outras aplicações ou

transformação de mensagens usando dados de múltiplas fontes. O processamento de fluxo é abordado em **Capítulo 14**.

A origem de Kafka

Kafka foi criado para resolver o problema de pipeline de dados no LinkedIn. Ele foi projetado para fornecer um sistema de mensagens de alto desempenho que pode lidar com muitos tipos de dados e fornecer dados limpos e estruturados sobre a atividade do usuário e métricas do sistema em tempo real.

Os dados realmente impulsionam tudo o que fazemos.

—Jeff Weiner, ex-CEO do LinkedIn

O problema do LinkedIn

Semelhante ao exemplo descrito no início deste capítulo, o LinkedIn tinha um sistema para coletar métricas de sistemas e aplicativos que usava coletores personalizados e ferramentas de código aberto para armazenar e apresentar dados internamente. Além das métricas tradicionais, como uso de CPU e desempenho de aplicativos, havia um recurso sofisticado de rastreamento de solicitações que usava o sistema de monitoramento e podia fornecer introspecção sobre como uma única solicitação de usuário se propagava por meio de aplicativos internos. O sistema de monitoramento teve muitas falhas, no entanto. Isso incluía a coleta de métricas com base em pesquisas, grandes intervalos entre as métricas e nenhuma capacidade dos proprietários de aplicativos gerenciarem suas próprias métricas. O sistema era de alto contato, exigindo intervenção humana para a maioria das tarefas simples, e inconsistente, com nomes de métricas diferentes para a mesma medição em sistemas diferentes.

Ao mesmo tempo, foi criado um sistema para rastrear informações de atividades do usuário. Este era um serviço HTTP ao qual os servidores frontend se conectavam periodicamente e publicavam um lote de mensagens (em formato XML) no serviço HTTP. Esses lotes foram então movidos para plataformas de processamento offline, onde os arquivos foram analisados e agrupados. Este sistema tinha muitas falhas. A formatação XML era inconsistente,

14 | Capítulo 1: Conheça Kafka

e analisá-lo era computacionalmente caro. Alterar o tipo de atividade do usuário rastreada exigiu uma quantidade significativa de trabalho coordenado entre front-ends e processamento offline. Mesmo assim, o sistema quebraria constantemente devido à mudança de esquemas. O rastreamento foi criado em lotes de hora em hora, portanto, não pôde ser usado em tempo real.

O monitoramento e o rastreamento da atividade do usuário não poderiam usar o mesmo serviço de back-end. O serviço de monitoramento era muito desajeitado, o formato dos dados não era orientado para rastreamento de atividades e o modelo de pesquisa para monitoramento não era compatível

com o modelo push para rastreamento. Ao mesmo tempo, o serviço de rastreamento era muito frágil para ser usado em métricas, e o processamento orientado em lote não era o modelo certo para monitoramento e alertas em tempo real. No entanto, os dados de monitoramento e rastreamento compartilhavam muitas características, e a correlação das informações (como a forma como tipos específicos de atividade do usuário afetavam o desempenho do aplicativo) era altamente desejável. Uma queda em tipos específicos de atividade do usuário poderia indicar problemas com o aplicativo que o atendia, mas horas de atraso no processamento de lotes de atividades significavam uma resposta lenta a esses tipos de problemas.

Inicialmente, as soluções de código aberto existentes foram minuciosamente investigadas para encontrar um novo sistema que fornecesse acesso em tempo real aos dados e fosse expandido para lidar com a quantidade de tráfego de mensagens necessária. Os sistemas protótipos foram configurados usando ActiveMQ, mas na época ele não conseguia lidar com a escala. Também foi uma solução frágil pela forma como o LinkedIn precisava utilizá-la, descobrindo muitas falhas no ActiveMQ que fariam com que os corretores parassem. Essas pausas fariam backup das conexões com os clientes e interfeririam na capacidade dos aplicativos de atender solicitações aos usuários. Foi tomada a decisão de avançar com uma infraestrutura personalizada para o pipeline de dados.

O Nascimento de Kafka

A equipe de desenvolvimento do LinkedIn foi liderada por Jay Kreps, principal engenheiro de software que anteriormente foi responsável pelo desenvolvimento e lançamento de código aberto do Voldemort, um sistema distribuído de armazenamento de valores-chave. A equipe inicial também incluiu Neha Narkhede e, posteriormente, Jun Rao. Juntos, eles decidiram criar um sistema de mensagens que pudesse atender às necessidades dos sistemas de monitoramento e rastreamento e escalar para o futuro. Os objetivos principais eram:

- Separar produtores e consumidores usando um modelo push-pull
 - Fornecer persistência para dados de mensagens dentro do sistema de mensagens para permitir que vários consumidores
 - Otimize para alto rendimento de mensagens
- Permitir que o dimensionamento horizontal do sistema cresça à medida que os fluxos de dados crescem

O resultado foi um sistema de mensagens de publicação/assinatura que tinha uma interface típica de sistemas de mensagens, mas uma camada de armazenamento mais parecida com um sistema de agregação de logs. Combinado com a adoção do Apache Avro para serialização de mensagens, o Kafka foi eficaz para

lidar com métricas e rastreamento de atividades do usuário em uma escala de bilhões de mensagens por dia. A escalabilidade do Kafka ajudou o uso do LinkedIn a crescer mais de sete trilhões de mensagens produzidas (em fevereiro de 2020) e mais de cinco petabytes de dados consumidos diariamente.

Código aberto

Kafka foi lançado como um projeto de código aberto no GitHub no final de 2010. À medida que começou a ganhar atenção na comunidade de código aberto, foi proposto e aceito como um projeto de incubadora da Apache Software Foundation em julho de 2011. Apache Kafka se formou na incubadora em outubro de 2012. Desde então, tem sido continuamente trabalhado e encontrou uma comunidade robusta de colaboradores e committers fora do LinkedIn. Kafka é agora usado em alguns dos maiores pipelines de dados do mundo, incluindo os da Netflix, Uber e muitas outras empresas.

A adoção generalizada do Kafka também criou um ecossistema saudável em torno do projeto principal. Existem grupos Meetup ativos em dezenas de países ao redor do mundo, proporcionando discussão local e suporte para processamento de stream. Existem também vários projetos de código aberto relacionados ao Apache Kafka. O LinkedIn continua a manter vários, incluindo Cruise Control, Kafka Monitor e Burrow. Além de suas ofertas comerciais, a Confluent lançou projetos incluindo ksqiDB, um registro de esquema e um proxy REST sob uma licença comunitária (que não é estritamente de código aberto, pois inclui restrições de uso). Vários dos projetos mais populares estão listados em [Apêndice B](#).

Engajamento Comercial

No outono de 2014, Jay Kreps, Neha Narkhede e Jun Rao deixaram o LinkedIn para fundar a Confluent, uma empresa centrada no fornecimento de desenvolvimento, suporte empresarial e treinamento para Apache Kafka. Eles também se juntaram a outras empresas (como a Heroku) no fornecimento de serviços em nuvem para Kafka. A Confluent, por meio de uma parceria com o Google, fornece clusters Kafka gerenciados no Google Cloud Platform, bem como serviços semelhantes na Amazon Web Services e Azure. Uma das outras iniciativas importantes do Confluent é organizar a série de conferências Kafka Summit. Iniciado em 2016, com conferências realizadas anualmente nos Estados Unidos e em Londres, o Kafka Summit oferece um local para a comunidade se reunir em escala global e compartilhar conhecimento sobre o Apache Kafka e projetos relacionados.

O Nome

As pessoas costumam perguntar como Kafka recebeu esse nome e se isso significa algo específico sobre o aplicativo em si. Jay Kreps ofereceu o seguinte insight:

Achei que, como Kafka era um sistema otimizado para escrita, faria sentido usar o nome do escritor. Tive muitas aulas de literatura na faculdade e gostei de Franz Kafka. Além disso, o nome parecia legal para um projeto de código aberto.

Então basicamente não há muito relacionamento.

Primeiros passos com Kafka

Agora que sabemos tudo sobre Kafka e sua história, podemos configurá-lo e construir nosso próprio pipeline de dados. No próximo capítulo, exploraremos a instalação e configuração do Kafka. Também abordaremos a seleção do hardware certo para executar o Kafka e algumas coisas que você deve ter em mente ao passar para as operações de produção.

CAPÍTULO 2

Instalando o Kafka

Este capítulo descreve como começar a usar o broker Apache Kafka, incluindo como configurar o Apache ZooKeeper, que é usado pelo Kafka para armazenar metadados para os brokers. O capítulo também abordará opções básicas de configuração para implantações do Kafka, bem como algumas sugestões para selecionar o hardware correto para executar os brokers. Por fim, abordamos como instalar vários corretores Kafka como parte de um único cluster e coisas que você deve saber ao usar o Kafka em

um ambiente de produção.

Configuração do ambiente

Antes de usar o Apache Kafka, seu ambiente precisa ser configurado com alguns pré-requisitos para garantir que funcione corretamente. As seções a seguir irão guiá-lo nesse processo.

Escolhendo um sistema operacional

Apache Kafka é um aplicativo Java e pode ser executado em vários sistemas operacionais. Embora o Kafka seja capaz de ser executado em muitos sistemas operacionais, incluindo Windows, macOS, Linux e outros, o Linux é o sistema operacional recomendado para o caso de uso geral. As etapas de instalação neste capítulo se concentrarão na configuração e no uso do Kafka em um ambiente Linux. Para obter informações sobre como instalar o Kafka no Windows e no macOS, consulte [Apêndice A](#).

Instalando Java

Antes de instalar o ZooKeeper ou o Kafka, você precisará de um ambiente Java configurado e funcionando. Kafka e ZooKeeper funcionam bem com todas as implementações Java baseadas em OpenJDK, incluindo Oracle JDK. As versões mais recentes do Kafka suportam Java 8 e Java 11. A versão exata instalada pode ser a versão fornecida pelo seu sistema operacional ou baixada diretamente da web – por exemplo, [o site da Oracle para o](#)

19

[Versão Oracle](#). Embora ZooKeeper e Kafka funcionem com uma edição runtime de Java, é recomendado ter o Java Development Kit (JDK) completo ao desenvolver ferramentas e aplicativos. Recomenda-se instalar a versão de patch lançada mais recente do seu ambiente Java, pois versões mais antigas podem ter vulnerabilidades de segurança. As etapas de instalação presumirão que você instalou o JDK versão 11, atualização 10, implantado em **`/usr/java/jdk-11.0.10`**.

Instalando o ZooKeeper

Apache Kafka usa Apache ZooKeeper para armazenar metadados sobre o cluster Kafka, bem como detalhes do cliente consumidor, conforme mostrado em [Figura 2-1](#). ZooKeeper é um serviço centralizado para manter informações de configuração, nomenclatura, fornecer sincronização distribuída e fornecer serviços de grupo. Este livro não entrará em muitos detalhes sobre o ZooKeeper, mas limitará as explicações apenas ao que é necessário para operar o Kafka. Embora seja possível executar um servidor ZooKeeper usando scripts contidos na distribuição Kafka, é trivial instalar uma versão completa do ZooKeeper a partir da distribuição.

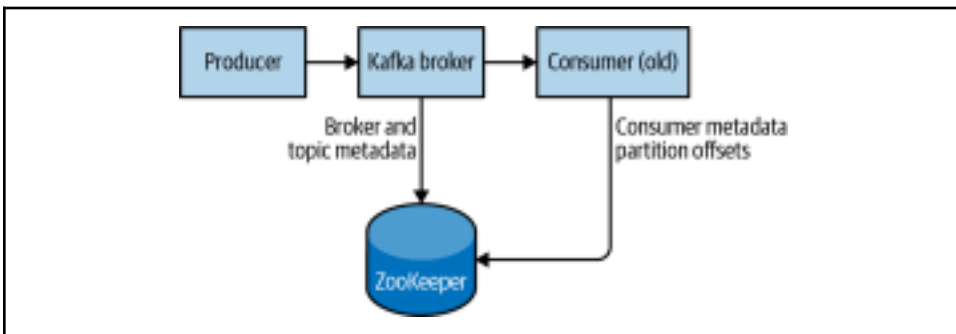


Figura 2-1. Kafka e ZooKeeper

Kafka foi testado extensivamente com a versão estável 3.5 do ZooKeeper e é atualizado regularmente para incluir a versão mais recente. Neste livro, usaremos o ZooKeeper 3.5.9, que pode ser baixado do site [Site do ZooKeeper](#).

Servidor autônomo

O ZooKeeper vem com um arquivo de configuração de exemplo básico que funcionará bem para a maioria dos casos de uso em **`/usr/local/zookeeper/conf/zoo_sample.cfg`**. No entanto, criaremos manualmente o nosso com algumas configurações básicas para fins de demonstração neste livro. O exemplo a seguir instala o ZooKeeper com uma configuração básica em **`/usr/local/zookeeper`**, armazenando seus dados em **`/var/lib/zookeeper`**.

20 | Capítulo 2: Instalando o Kafka

```
# tar -zxf apache-zookeeper-3.5.9-bin.tar.gz
# mv apache-zookeeper-3.5.9-bin /usr/local/zookeeper
# mkdir -p /var/lib/zookeeper
# cp > /usr/local/zookeeper/conf/zoo.cfg << EOF
> tickTime=2000
> dataDir=/var/lib/zookeeper
> portacliente=2181
> EOF
# exportar JAVA_HOME=/usr/java/jdk-11.0.10
# /usr/local/zookeeper/bin/zkServer.sh iniciar
JMX habilitado por padrão
Usando configuração: /usr/local/zookeeper/bin/./conf/zoo.cfg
Iniciando o zookeeper... INICIADO
#
```

Agora você pode validar se o ZooKeeper está funcionando corretamente no modo autônomo conectando-se à porta do cliente e enviando o comando de quatro letras `srvr`. Isso retornará informações básicas do ZooKeeper do servidor em execução:


```
# telnet host local 2181
Tentando 127.0.0.1...
Conectado ao host local.
O caractere de escape é '^]'.
Srvr
Versão do Zookeeper: 3.5.9-83df9301aa5c2a5d284a9940177808c01bc35cef,
construído em 01/06/2021 19:49 GMT
Latência mín/média/máx: 0/0/0
Recebido: 1
Enviado: 0
Conexões: 1
Excelente: 0
Zxid: 0x0
Modo: autônomo
Contagem de nós: 5
Conexão fechada por host estrangeiro.
#
```

Conjunto ZooKeeper

O ZooKeeper foi projetado para funcionar como um cluster, chamado **conjunto**, para garantir alta disponibilidade. Devido ao algoritmo de balanceamento utilizado, recomenda-se que os conjuntos contenham um número ímpar de servidores (por exemplo, 3, 5 e assim por diante), já que a maioria dos membros do conjunto (um **quorum**) deve estar funcionando para que o ZooKeeper responda às solicitações. Isso significa que em um conjunto de três nós, você pode executar com um nó faltando. Com um conjunto de cinco nós, você pode executar com dois nós faltando.

Configuração do ambiente | 21

Dimensionando seu conjunto ZooKeeper



Considere executar o ZooKeeper em um conjunto de cinco nós. Para fazer alterações de configuração no conjunto, incluindo troca de um nó, você precisará recarregar os nós, um de cada vez. Se o seu conjunto puder não tolerar mais de um nó inativo, fazendo manutenção trabalho introduz riscos adicionais. Também não é recomendado executar mais de sete nós, pois o desempenho pode começar a degradar devido a a natureza do protocolo de consenso.

Além disso, se você achar que cinco ou sete nós não estão suportando a carga devido a muitas conexões de cliente, considere adicionar nós observadores adicionais para ajudar no balanceamento do tráfego somente leitura.

Para configurar servidores ZooKeeper em um conjunto, eles devem ter uma configuração comum que liste todos os servidores, e cada servidor precisa de um **meuid** arquivo no diretório de dados que especifica o número de ID do servidor. Se os nomes de host dos servidores do conjunto forem

zoo1.exemplo.com, zoo2.exemplo.com, e zoo3.exemplo.com, o arquivo de configuração pode ter esta aparência:

```
tickTime=2000
dadosDir=/var/lib/zookeeper
portacliente=2181
limite de inicialização=20
limite de sincronização=5
servidor.1=zoo1.exemplo.com:2888:3888
servidor.2=zoo2.exemplo.com:2888:3888
servidor.3=zoo3.exemplo.com:2888:3888
```

Nesta configuração, o limite de inicialização é a quantidade de tempo para permitir que os seguidores se conectem com um líder. O limite de sincronização o valor limita quanto tempo os seguidores fora de sincronia podem ficar com o líder. Ambos os valores são um número de tickTime unidades, o que faz com que Limite de inicialização 20×2.000 ms ou 40 segundos. A configuração também lista cada servidor do conjunto. Os servidores são especificados no formato *servidor.X = nome do host: ponto Porta:leaderPort*, com os seguintes parâmetros:

X

O número de identificação do servidor. Deve ser um número inteiro, mas não precisa ser baseado em zero ou sequencial.

nome do host

O nome do host ou endereço IP do servidor.

peerPort

A porta TCP pela qual os servidores do conjunto se comunicam entre si.

22 | Capítulo 2: Instalando o Kafka

líderPort

A porta TCP na qual a eleição do líder é executada.

Os clientes só precisam ser capazes de se conectar ao conjunto através do *portacliente*, mas os membros do conjunto devem ser capazes de comunicar entre si através de todas as três portas.

Além do arquivo de configuração compartilhado, cada servidor deve ter um arquivo no ***diretório de dados*** diretório com o nome ***meuId***. Este arquivo deve conter o número de ID do servidor, que deve corresponder ao arquivo de configuração. Assim que essas etapas forem concluídas, os servidores serão inicializados e se comunicarão entre si em conjunto.

Testando o ZooKeeper Ensemble em uma única máquina

É possível testar e executar um conjunto ZooKeeper em um único máquina especificando todos os nomes de host na configuração como *host local* e ter portas exclusivas especificadas para *peerPort* e *líderPort* para cada instância. Além disso, um separado ***zoo1ógico.cfg*** precisaria ser criado



para cada instância com um único ***dadosDir*** e *portacliente* definido para cada instância. Isso pode ser útil para fins de teste só, mas é ***não*** recomendado para sistemas de produção.

Instalando um corretor Kafka

Depois que o Java e o ZooKeeper estiverem configurados, você estará pronto para instalar o Apache Kafka. A versão atual pode ser baixada no site [Site Kafka](#). Até o momento, essa versão é 2.8.0 rodando no Scala versão 2.13.0. Os exemplos neste capítulo são mostrados usando a versão 2.7.0.

O exemplo a seguir instala o Kafka em ***/usr/local/kafka***, configurado para usar o servidor Zoo-Keeper iniciado anteriormente e para armazenar os segmentos de log de mensagens armazenados em ***/tmp/kafka-logs***.

```
# tar -zxf kafka_2.13-2.7.0.tgz
# mv kafka_2.13-2.7.0 /usr/local/kafka
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk-11.0.10
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

Assim que o broker Kafka for iniciado, podemos verificar se ele está funcionando realizando algumas operações simples no cluster: criar um tópico de teste, produzir algumas mensagens e consumir as mesmas mensagens.

Instalando um corretor Kafka | 23

Crie e verifique um tópico:

```
# /usr/local/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092
-- criar
--fator de replicação 1 --partições 1 --topic test
Tópico "teste" criado.
# /usr/local/kafka/bin/kafka-topics.sh --bootstrap-server
localhost:9092 --describe --topic test
Tópico: teste PartitionCount: 1 ReplicationFactor: 1 Configurações:
Tópico: teste Partição: 0 Líder: 0 Réplicas: 0 Isr: 0 #
```

Produza mensagens para um tópico de teste (use Ctrl-C para parar o produtor a qualquer momento):

```
# /usr/local/kafka/bin/kafka-console-producer.sh
--bootstrap-server localhost:9092 --topic test
Mensagem de teste 1
Mensagem de teste 2
^ C
#
```

Consumir mensagens de um tópico de teste:

```
# /usr/local/kafka/bin/kafka-console-consumer.sh
```

```
--bootstrap-server localhost:9092 --topic test --from-beginning
Mensagem de teste 1
Mensagem de teste 2
^ C
Processou um total de 2 mensagens
#
```

Descontinuação das conexões ZooKeeper em utilitários Kafka CLI

Se você estiver familiarizado com versões mais antigas dos utilitários Kafka, você pode estar acostumado a usar um `--zookeeper` cadeia de conexão. Isso tem sido descontinuado em quase todos os casos. A melhor prática atual é usar o mais novo `--bootstrap-servidor` opção e conecte-se diretamente ao corretor Kafka. Se você estiver executando em um cluster, poderá fornecer o `host:port` de qualquer corretor no cluster.

Configurando o Broker

O exemplo de configuração fornecido com a distribuição Kafka é suficiente para executar um servidor independente como prova de conceito, mas provavelmente não será suficiente para grandes instalações. Existem inúmeras opções de configuração para Kafka que controlam todos os aspectos de configuração e ajuste. A maioria das opções pode ser deixada com as configurações padrão, pois elas lidam com aspectos de ajuste do corretor Kafka que não serão aplicáveis até que você tenha um caso de uso específico que exija o ajuste dessas configurações.

24 | Capítulo 2: Instalando o Kafka

Parâmetros Gerais do Corretor

Existem vários parâmetros de configuração do agente que devem ser revisados ao implantar o Kafka para qualquer ambiente que não seja um agente independente em um único servidor. Esses parâmetros tratam da configuração básica do broker, e a maioria deles deve ser alterada para funcionar corretamente em um cluster com outros brokers.

corretor.id

Cada corretor Kafka deve ter um identificador inteiro, que é definido usando o `corretor.id` configuração. Por padrão, esse número inteiro é definido como 0, mas pode ser qualquer valor. É essencial que o número inteiro seja exclusivo para cada corretor dentro de um único cluster Kafka. A seleção deste número é tecnicamente arbitrária e pode ser movimentada entre corretores se necessário para tarefas de manutenção. No entanto, é altamente recomendável definir esse valor como algo intrínseco ao host, para que, ao realizar a manutenção, não seja oneroso mapear os números de ID do corretor para os hosts. Por exemplo, se seus nomes de host contiverem um número exclusivo (como `host1.exemplo.com`, `host2.exemplo.com`, etc.), então 1 e 2 seriam boas escolhas para `corretor.id`.

valores, respectivamente.

ouvintes

Versões mais antigas do Kafka usavam um simples porta configuração. Isso ainda pode ser usado como backup para configurações simples, mas é uma configuração obsoleta. O arquivo de configuração de exemplo inicia o Kafka com um ouvinte na porta TCP 9092. O novo ouvintes config é uma lista separada por vírgulas de URIs que ouvimos com os nomes dos ouvintes. Se o nome do ouvinte não for um protocolo de segurança comum, então outra configuração `listener.security.protocol.map` também deve ser configurado. Um ouvinte é definido como `<protocolo>://<hostname>:<porta>`. Um exemplo de uma lei ouvinte configuração é `TEXT SIMPLES://localhost:9092,SSL://:9091`. Especificando o nome do host como `0.0.0.0` será vinculado a todas as interfaces. Deixar o nome do host vazio irá vinculá-lo à interface padrão. Lembre-se de que se for escolhida uma porta inferior a 1024, o Kafka deverá ser iniciado como root. Executar o Kafka como root não é uma configuração recomendada.

tratador do zoológico.connect

A localização do ZooKeeper usado para armazenar os metadados do corretor é definida usando o comando `tratador do zoológico.connect` parâmetro de configuração. A configuração de exemplo usa um Zoo-Keeper em execução na porta 2181 no host local, que é especificado como `host local:2181`. O formato deste parâmetro é uma lista separada por ponto e vírgula de nome do host:porta/caminho cordas, que incluem:

nome do host

O nome do host ou endereço IP do servidor ZooKeeper.

Configurando o Broker | 25

porta

O número da porta do cliente para o servidor.

/caminho

Um caminho opcional do ZooKeeper para usar como ambiente chroot para o cluster Kafka. Se for omitido, o caminho raiz será usado.

Se um caminho chroot (um caminho designado para atuar como diretório raiz para um determinado aplicativo) for especificado e não existir, ele será criado pelo broker quando ele for inicializado.

Por que usar um caminho Chroot?

Geralmente é considerado uma boa prática usar um caminho chroot para o cluster Kafka. Isso permite que o conjunto ZooKeeper seja compartilhado com outros aplicativos, incluindo outros clusters Kafka, sem conflito. Também é melhor especificar vários ZooKeeper servidores (que fazem parte do mesmo conjunto) nesta configuração. Isso permite que o corretor Kafka se conecte a outro membro



do conjunto ZooKeeper em caso de falha do servidor.

log.dirs

Kafka persiste todas as mensagens no disco e esses segmentos de log são armazenados no diretório especificado no arquivo `log.dir` configuração. Para vários diretórios, a configuração `log.dirs` é preferível. Se este valor não for definido, o padrão voltará para `log.dir`. `log.dirs` é uma lista de caminhos separados por vírgula no sistema local. Se mais de um caminho for especificado, o intermediário armazenará partições neles de uma forma “menos usada”, com os segmentos de log de uma partição armazenados no mesmo caminho. Observe que o broker colocará uma nova partição no caminho que tenha o menor número de partições atualmente armazenadas nele, e não a menor quantidade de espaço em disco usado, portanto, uma distribuição uniforme de dados em vários diretórios não é garantida.

num.recovery.threads.per.data.dir

Kafka usa um conjunto configurável de threads para lidar com segmentos de log. Atualmente, este pool de threads é usado:

- Ao iniciar normalmente, para abrir os segmentos de log de cada partição
- Ao iniciar após uma falha, para verificar e truncar os segmentos de log de cada partição
- Ao desligar, para fechar corretamente os segmentos de log

Por padrão, apenas um thread por diretório de log é usado. Como esses threads são usados apenas durante a inicialização e o encerramento, é razoável definir um número maior de threads para paralelizar as operações. Especificamente, ao se recuperar de um desligamento incorreto, isso pode significar a diferença de várias horas ao reiniciar um corretor com um

26 | Capítulo 2: Instalando o Kafka

grande número de partições! Ao definir este parâmetro, lembre-se que o número configurado é por diretório de log especificado com `log.dirs`. Isto significa que se `num. recuperação.threads.per.data.dir` está definido como 8 e há 3 caminhos especificados em `log.dirs`, isso é um total de 24 threads.

auto.create.topics.enable

A configuração padrão do Kafka especifica que o broker deve criar automaticamente um tópico nas seguintes circunstâncias:

- Quando um produtor começa a escrever mensagens para o tópico
- Quando um consumidor começa a ler mensagens do tópico
- Quando qualquer cliente solicita metadados para o tópico

Em muitas situações, este pode ser um comportamento indesejável, especialmente porque não há como validar a existência de um tópico através do protocolo Kafka sem causar a sua criação. Se você estiver gerenciando

explicitamente a criação de tópicos, seja manualmente ou por meio de um sistema de provisionamento, poderá definir o `auto.create.topics.enable` configuração para `false`.

auto.leader.rebalance.enable

Para garantir que um cluster Kafka não fique desequilibrado por ter toda a liderança de tópico em um corretor, esta configuração pode ser especificada para garantir que a liderança seja o mais equilibrada possível. Ele habilita um thread em segundo plano que verifica a distribuição de partições em intervalos regulares (esse intervalo é configurável via `lider.desequilíbrio.verificar.intervalo.segundos`). Se o desequilíbrio de liderança exceder outra configuração, `lider.desequilíbrio.por.corretor.percentagem`, então um rebalanceamento de líderes preferenciais para partições é iniciado.

excluir.topic.enable

Dependendo do seu ambiente e das diretrizes de retenção de dados, talvez você queira bloquear um cluster para evitar exclusões arbitrárias de tópicos. A desativação da exclusão de tópicos pode ser definida definindo este sinalizador como `false`.

Padrões de tópico

A configuração do servidor Kafka especifica muitas configurações padrão para tópicos criados. Vários desses parâmetros, incluindo contagens de partições e retenção de mensagens, podem ser definidos por tópico usando as ferramentas administrativas (abordadas em [Capítulo 12](#)). Os padrões na configuração do servidor devem ser definidos com valores de linha de base apropriados para a maioria dos tópicos no cluster.

Configurando o Broker | 27

Usando substituições por tópico

Nas versões mais antigas do Kafka, era possível especificar tópicos por tópico. passeios para essas configurações na configuração do corretor usando o parâmetros `log.retenção.horas.por.tópico`, `log.retenção.bytes.por.tópico`, e `log.segment.bytes.per.topic`. Esses parâmetros não são mais suportados e as substituições devem ser especificadas fiado usando as ferramentas administrativas.

num.partições

O `num.partições` O parâmetro determina com quantas partições um novo tópico é criado, principalmente quando a criação automática de tópicos está habilitada (que é a configuração padrão). Este parâmetro é padronizado para uma partição. Lembre-se de que o número de partições de um tópico só pode ser aumentado, nunca diminuído. Isso significa que se um tópico precisar ter menos partições do que `num.partições`, será necessário tomar cuidado ao criar manualmente o tópico (discutido em [Capítulo 12](#)).



Conforme descrito em **Capítulo 1**, as partições são a forma como um tópico é dimensionado dentro de um cluster Kafka, o que torna importante o uso de contagens de partição que equilibrarão a carga de mensagens em todo o cluster à medida que os agentes são adicionados. Muitos usuários terão a contagem de partições de um tópico igual ou um múltiplo do número de agentes no cluster. Isso permite que as partições sejam distribuídas uniformemente aos intermediários, que distribuirão uniformemente a carga da mensagem. Por exemplo, um tópico com 10 partições operando em um cluster Kafka com 10 hosts com liderança equilibrada entre todos os 10 hosts terá rendimento ideal. Entretanto, isso não é um requisito, pois você também pode equilibrar a carga de mensagens de outras maneiras, como ter vários tópicos.

Como escolher o número de partições

Existem vários fatores a serem considerados ao escolher o número de partições:

- Qual é o rendimento que você espera alcançar para o tópico? Por exemplo, você espera gravar 100 KBps ou 1 GBps?
- Qual é o rendimento máximo que você espera alcançar ao consumir de uma única partição? Uma partição sempre será consumida completamente por um único consumidor (mesmo quando não estiver utilizando grupos de consumidores, o consumidor deverá ler todas as mensagens na partição). Se você sabe que seu consumidor mais lento grava os dados em um banco de dados e esse banco de dados nunca lida com mais de 50 MBps de cada thread gravado nele, então você sabe que está limitado a uma taxa de transferência de 50 MBps ao consumir de uma partição.
- Você pode realizar o mesmo exercício para estimar o rendimento máximo por produtor para uma única partição, mas como os produtores são normalmente muito mais rápidos que os consumidores, geralmente é seguro ignorar isso.

28 | Capítulo 2: Instalando o Kafka

- Se você estiver enviando mensagens para partições com base em chaves, adicionar partições posteriormente pode ser muito desafiador, portanto calcule a taxa de transferência com base no uso futuro esperado e não no uso atual.
- Considere o número de partições que você colocará em cada agente e o espaço em disco disponível e a largura de banda da rede por agente.
 - Evite superestimar, pois cada partição utiliza memória e outros recursos do intermediário e aumentará o tempo para atualizações de metadados e transferências de liderança.
- Você irá espelhar dados? Talvez você também precise considerar o rendimento da sua configuração de espelhamento. Partições grandes podem se tornar um gargalo em muitas configurações de espelhamento.
- Se estiver usando serviços em nuvem, você tem limitações de IOPS (operações de entrada/saída por segundo) em suas VMs ou discos?

Pode haver limites rígidos no número de IOPS permitidos, dependendo do seu serviço de nuvem e da configuração da VM, o que fará com que você atinja as cotas. Ter muitas partições pode ter o efeito colateral de aumentar a quantidade de IOPS devido ao paralelismo envolvido.

Com tudo isso em mente, fica claro que você deseja muitas partições, mas não muitas. Se tiver alguma estimativa relativamente ao rendimento alvo do tópico e ao rendimento esperado dos consumidores, pode dividir o rendimento alvo pelo rendimento esperado do consumidor e derivar o número de divisórias desta forma. Portanto, se quisermos escrever e ler 1 GBps de um tópico e sabemos que cada consumidor só pode processar 50 MBps, sabemos que precisamos de pelo menos 20 partições. Dessa forma, podemos ter 20 consumidores lendo o tópico e atingir 1 GBps.

Se você não tiver essas informações detalhadas, nossa experiência sugere que limitar o tamanho da partição no disco a menos de 6 GB por dia de retenção geralmente fornece resultados satisfatórios. Começar pequeno e expandir conforme necessário é mais fácil do que começar muito grande.

padrão.fator de replicação

Se a criação automática de tópicos estiver habilitada, esta configuração definirá qual deve ser o fator de replicação para novos tópicos. A estratégia de replicação pode variar dependendo da durabilidade ou disponibilidade desejada de um cluster e será discutida mais detalhadamente em capítulos posteriores. A seguir está uma breve recomendação se você estiver executando o Kafka em um cluster que evitará interrupções devido a fatores fora dos recursos internos do Kafka, como falhas de hardware.

É altamente recomendado definir o fator de replicação para pelo menos 1 acima do `min.insync.replicas` contexto. Para configurações mais resistentes a falhas, se você tiver clusters grandes e hardware suficientes, defina o fator de replicação como 2 acima do `min.insync.replicas` (abreviado como RF++) pode ser preferível. RF++ permitirá uma manutenção mais fácil e evitará interrupções. O raciocínio por trás desta recomendação é

Configurando o Broker | 29

para permitir que uma interrupção planejada no conjunto de réplicas e uma interrupção não planejada ocorram simultaneamente. Para um cluster típico, isso significaria que você teria no mínimo três réplicas de cada partição. Um exemplo disso é se houver uma interrupção no switch de rede, falha de disco ou algum outro problema não planejado durante uma implantação contínua ou atualização do Kafka ou do sistema operacional subjacente, você pode ter certeza de que ainda haverá uma réplica adicional disponível. Isso será discutido mais em [Capítulo 7](#).

log.retenção.ms

A configuração mais comum de quanto tempo o Kafka reterá as mensagens é por tempo. O padrão é especificado no arquivo de configuração usando o `log.retenção.horas` parâmetro e é definido como 168 horas ou uma semana.

No entanto, existem dois outros parâmetros permitidos, `log.retenção.minutos` e `log.retenção.ms`. Todos os três controlam o mesmo objetivo (a quantidade de tempo após o qual as mensagens podem ser excluídas), mas o parâmetro recomendado a ser usado é `log.retenção.ms`, pois o tamanho menor da unidade terá precedência se mais de um for especificado. Isso garantirá que o valor definido para `log.retenção.ms` é sempre o usado. Se mais de um for especificado, o tamanho menor da unidade terá precedência.



Retenção por tempo e horários da última modificação

A retenção por tempo é realizada examinando a última modificação `time (mtime)` em cada arquivo de segmento de log no disco. Sob condições normais após as operações, este é o horário em que o segmento de log foi fechado e representa o carimbo de data/hora da última mensagem no arquivo. No entanto, ao usar ferramentas administrativas para mover partições entre corretoras, este tempo não é preciso e resultará em retenção excessiva para essas partições. Para obter mais informações sobre isso, consulte [Capítulo 12](#) discutindo movimentos de partição.

`log.retenção.bytes`

Outra forma de expirar mensagens é baseada no número total de bytes de mensagens retidas. Este valor é definido usando o `log.retenção.bytes` parâmetro e é aplicado por partição. Isso significa que se você tiver um tópico com 8 partições, e `log.retenção.bytes` estiver definido como 1 GB, a quantidade de dados retidos para o tópico será de no máximo 8 GB. Observe que toda a retenção é executada para partições individuais, não para o tópico. Isto significa que se o número de partições de um tópico for expandido, a retenção também aumentará se `log.retenção.bytes` é usado. Definir o valor como `-1` permitirá retenção infinita.

30 | Capítulo 2: Instalando o Kafka

Configurando a retenção por tamanho e tempo

Se você especificou um valor para ambos `log.retenção.bytes` e `log.retenção.ms` (ou outro parâmetro para retenção por tempo), as mensagens podem ser removidas quando qualquer um dos critérios for atendido. Por exemplo, se `log.retenção.ms` está definido como 86400000 (1 dia) e registro. `retenção.bytes` estiver definido como 1000000000 (1 GB), é possível mensagens com menos de 1 dia serão excluídas se o volume total

O número de mensagens ao longo do dia é superior a 1 GB. Por outro lado, se o volume for inferior a 1 GB, as mensagens poderão ser excluídas após 1 dia, mesmo que o tamanho total da partição seja inferior a 1 GB. Isto é recomendado, por simplicidade, escolher o tamanho ou o tempo baseada na retenção – e não em ambos – para evitar surpresas e perda de dados, mas ambos podem ser usados para fins mais avançados



configurações.

log.segmento.bytes

As configurações de retenção de log mencionadas anteriormente operam em segmentos de log, não em mensagens individuais. À medida que as mensagens são produzidas para o agente Kafka, elas são anexadas ao segmento de log atual da partição. Depois que o segmento de log atingir o tamanho especificado pelo `log.segmento.bytes` parâmetro, cujo padrão é 1 GB, o segmento de log é fechado e um novo é aberto. Depois que um segmento de log for fechado, ele poderá ser considerado para expiração. Um tamanho de segmento de log menor significa que os arquivos devem ser fechados e alocados com mais frequência, o que reduz a eficiência geral das gravações em disco.

Ajustar o tamanho dos segmentos de toras pode ser importante se os tópicos tiverem uma taxa de produção baixa. Por exemplo, se um tópico recebe apenas 100 megabytes de mensagens por dia e `log.segmento.bytes` estiver definido como padrão, levará 10 dias para preencher um segmento. Como as mensagens não podem expirar até que o segmento de log seja fechado, se `log.retenção.ms` estiver definido como 604800000 (1 semana), na verdade haverá até 17 dias de mensagens retidas até que o segmento de log fechado expire. Isso ocorre porque uma vez que o segmento de log é fechado com os atuais 10 dias de mensagens, esse segmento de log deve ser retido por 7 dias antes de expirar com base na política de tempo (já que o segmento não pode ser removido até que a última mensagem no segmento possa ser expirado).



Recuperando deslocamentos por carimbo de data/hora

O tamanho do segmento de log também afeta o comportamento de busca conjuntos por carimbo de data/hora. Ao solicitar compensações para uma partição em um determinado momento timestamp específico, Kafka encontra o arquivo de segmento de log que estava sendo escrito naquela época. Isso é feito usando a criação e o último hora de modificação do arquivo e procurando por um arquivo que foi criado antes do carimbo de data/hora especificado e modificado pela última vez após o horário carimbo. O deslocamento no início desse segmento de log (que é também o nome do arquivo) é retornado na resposta.

Configurando o Broker | 31

log.roll.ms

Outra maneira de controlar quando os segmentos de log são fechados é usando o comando `log.roll.ms` parâmetro, que especifica o período de tempo após o qual um segmento de log deve ser fechado. Tal como acontece com o `log.retenção.bytes` e `log.retenção.ms` parâmetros, `log.segmento.bytes` e `log.roll.ms` não são propriedades mutuamente exclusivas. Kafka fechará um segmento de log quando o limite de tamanho for atingido ou quando o limite de tempo for atingido, o que ocorrer primeiro. Por padrão, não há configuração para `log.roll.ms`, o que resulta no fechamento apenas de segmentos de log por tamanho.



Desempenho do disco ao usar segmentos baseados em tempo

Ao usar um limite de segmento de log baseado em tempo, é importante considerar considere o impacto no desempenho do disco quando vários segmentos de log são fechados simultaneamente. Isso pode acontecer quando há muitos partições que nunca atingem o limite de tamanho para segmentos de log, como o relógio para o limite de tempo começará quando o corretor iniciar e irá sempre execute ao mesmo tempo para essas partições de baixo volume.

`min.insync.replicas`

Ao configurar seu cluster para durabilidade de dados, definindo `min.insync.replicas` para 2 garante que pelo menos duas réplicas estejam atualizadas e “sincronizadas” com o produtor. Isso é usado em conjunto com a configuração da configuração do produtor para confirmar “todas” as solicitações. Isso garantirá que pelo menos duas réplicas (líder e outra) reconheçam uma gravação para que ela seja bem-sucedida. Isso pode evitar a perda de dados em cenários em que o líder reconhece uma gravação, depois sofre uma falha e a liderança é transferida para uma réplica que não possui uma gravação bem-sucedida. Sem essas configurações duráveis, o produtor pensaria que produziu com sucesso e a(s) mensagem(s) seria(ão) jogada(s) no chão e perdidas. No entanto, configurar para maior durabilidade tem o efeito colateral de ser menos eficiente devido à sobrecarga extra envolvida, portanto, clusters com alto rendimento que podem tolerar perdas ocasionais de mensagens não são recomendados para alterar essa configuração do padrão 1. Consulte [Capítulo 7](#) para mais informações.

`mensagem.max.bytes`

O broker Kafka limita o tamanho máximo de uma mensagem que pode ser produzida, configurado pelo `mensagem.max.bytes` parâmetro, cujo padrão é 1000000 ou 1 MB. Um produtor que tentar enviar uma mensagem maior que esta receberá um erro do broker e a mensagem não será aceita. Tal como acontece com todos os tamanhos de bytes especificados no broker, esta configuração lida com o tamanho da mensagem compactada, o que significa que os produtores podem enviar mensagens muito maiores que esse valor descompactado, desde que compactados abaixo do configurado. `mensagem.max.bytes` tamanho.

32 | Capítulo 2: Instalando o Kafka

Há impactos visíveis no desempenho decorrentes do aumento do tamanho permitido da mensagem. Mensagens maiores significarão que os threads do broker que lidam com o processamento de conexões e solicitações de rede trabalharão por mais tempo em cada solicitação. Mensagens maiores também aumentam o tamanho das gravações em disco, o que afetará o rendimento de E/S. Outras soluções de armazenamento, como armazenamentos de blobs e/ou armazenamento em camadas, podem ser outro método para resolver grandes problemas de gravação em disco, mas

não serão abordadas neste capítulo.



Coordenando configurações de tamanho de mensagem

O tamanho da mensagem configurado no broker Kafka deve ser coordenado com o `buscar.mensagem.max.bytes` configuração em clientes sumários. Se esse valor for menor que `mensagem.max.bytes`, então os consumidores que encontrarem mensagens maiores não conseguirão buscar essas mensagens, resultando em uma situação em que o consumidor recebe preso e não pode prosseguir. A mesma regra se aplica ao `réplica.fetch.max.bytes` configuração nos corretores quando configurado em um cluster.

Selecionando Hardware

Selecionar uma configuração de hardware apropriada para um corretor Kafka pode ser mais arte do que ciência. O próprio Kafka não possui requisitos rígidos em uma configuração de hardware específica e será executado sem problemas na maioria dos sistemas. No entanto, quando o desempenho se torna uma preocupação, há vários fatores que podem contribuir para os gargalos gerais de desempenho: capacidade e rendimento do disco, memória, rede e CPU. Ao dimensionar o Kafka muito grande, também pode haver restrições no número de partições que um único corretor pode manipular devido à quantidade de metadados que precisam ser atualizados. Depois de determinar quais tipos de desempenho são mais críticos para seu ambiente, você poderá selecionar uma configuração de hardware otimizada e apropriada para seu orçamento.

Taxa de transferência de disco

O desempenho dos clientes produtores será influenciado mais diretamente pelo rendimento do disco intermediário usado para armazenar segmentos de log. As mensagens Kafka devem ser enviadas para armazenamento local quando são produzidas, e a maioria dos clientes esperará até que pelo menos um corretor tenha confirmado que as mensagens foram confirmadas antes de considerar o envio bem-sucedido. Isso significa que gravações mais rápidas no disco equivalerão a uma latência de produção menor.

A decisão óbvia quando se trata de rendimento do disco é usar unidades de disco rígido (HDDs) tradicionais ou discos de estado sólido (SSDs). Os SSDs têm tempos de busca e acesso drasticamente mais baixos e fornecem o melhor desempenho. Os HDDs, por outro lado, são mais econômicos e oferecem mais capacidade por unidade. Você também pode

Selecionando Hardware | 33

melhore o desempenho dos HDDs usando mais deles em um intermediário, seja por ter vários diretórios de dados ou configurando as unidades em uma configuração de matriz redundante de discos independentes (RAID). Outros fatores, como a tecnologia específica da unidade (por exemplo, armazenamento conectado serial ou ATA serial), bem como a qualidade do controlador da unidade, afetarão o rendimento. Geralmente, as observações

mostram que as unidades HDD são normalmente mais úteis para clusters com necessidades de armazenamento muito altas, mas não são acessadas com tanta frequência, enquanto os SSDs são opções melhores se houver um número muito grande de conexões de clientes.

Capacidade do disco

A capacidade é o outro lado da discussão sobre armazenamento. A quantidade de capacidade de disco necessária é determinada por quantas mensagens precisam ser retidas a qualquer momento. Se for esperado que o corretor receba 1 TB de tráfego por dia, com 7 dias de retenção, então o corretor precisará de um mínimo de 7 TB de armazenamento utilizável para segmentos de log. Você também deve levar em consideração pelo menos 10% de sobrecarga para outros arquivos, além de qualquer buffer que deseja manter para flutuações no tráfego ou crescimento ao longo do tempo.

A capacidade de armazenamento é um dos fatores a considerar ao dimensionar um cluster Kafka e determinar quando expandi-lo. O tráfego total de um cluster pode ser equilibrado em todo o cluster com múltiplas partições por tópico, o que permitirá que agentes adicionais aumentem a capacidade disponível se a densidade em um único agente não for suficiente. A decisão sobre quanta capacidade de disco será necessária também será informada pela estratégia de replicação escolhida para o cluster (que é discutida com mais detalhes em [Capítulo 7](#)).

Memória

O modo normal de operação para um consumidor Kafka é ler a partir do final das divisórias, onde o consumidor fica muito atrás dos produtores e fica muito pouco atrás dos produtores, se é que fica. Nessa situação, as mensagens que o consumidor está lendo são armazenadas de maneira ideal no cache de páginas do sistema, resultando em leituras mais rápidas do que se o corretor tivesse que reler as mensagens do disco. Portanto, ter mais memória disponível para o sistema para cache de páginas melhorará o desempenho dos clientes consumidores.

O próprio Kafka não precisa de muita memória heap configurada para a Java Virtual Machine (JVM). Mesmo um intermediário que processa 150.000 mensagens por segundo e uma taxa de dados de 200 megabits por segundo pode ser executado com um heap de 5 GB. O restante da memória do sistema será usado pelo cache da página e beneficiará o Kafka, permitindo que o sistema armazene em cache os segmentos de log em uso. Este é o principal motivo pelo qual não é recomendado colocar o Kafka em um sistema com qualquer outro aplicativo significativo, pois ele terá que compartilhar o uso do cache da página. Isso diminuirá o desempenho do consumidor para Kafka.

Rede

A taxa de transferência de rede disponível especificará a quantidade máxima de tráfego que o Kafka pode suportar. Este pode ser um fator determinante, combinado com o armazenamento em disco, para o dimensionamento do cluster. Isso é complicado pelo desequilíbrio inerente entre o uso da rede de entrada e saída criado pelo suporte do Kafka a vários consumidores. Um produtor pode escrever 1 MB por segundo para um determinado tópico, mas pode haver qualquer número de consumidores que criem um multiplicador no uso da rede de saída. Outras operações, como replicação de cluster (abordadas em [Capítulo 7](#)) e espelhamento (discutido em [Capítulo 10](#)), também aumentará os requisitos. Caso a interface de rede fique saturada, não é incomum que a replicação do cluster fique para trás, o que pode deixar o cluster em um estado vulnerável. Para evitar que a rede seja um fator importante de governança, é recomendável operar com NICs (placas de interface de rede) de pelo menos 10 Gb. Máquinas mais antigas com NICs de 1 Gb ficam facilmente saturadas e não são recomendadas.

CPU

O poder de processamento não é tão importante quanto o disco e a memória até que você comece a dimensionar o Kafka muito grande, mas afetará o desempenho geral do corretor até certo ponto. Idealmente, os clientes devem compactar mensagens para otimizar o uso da rede e do disco. O corretor Kafka deve descompactar todos os lotes de mensagens, entretanto, para validar o soma de verificação das mensagens individuais e atribuir compensações. Em seguida, ele precisa recomparar o lote de mensagens para armazená-lo no disco. É daí que vem a maior parte dos requisitos de poder de processamento de Kafka. Entretanto, esse não deve ser o principal fator na seleção de hardware, a menos que os clusters se tornem muito grandes, com centenas de nós e milhões de partições em um único cluster. Nesse ponto, selecionar uma CPU com melhor desempenho pode ajudar a reduzir o tamanho dos clusters.

Kafka na nuvem

Nos últimos anos, uma instalação mais comum do Kafka é em ambientes de computação em nuvem, como Microsoft Azure, AWS da Amazon ou Google Cloud Platform. Há muitas opções para configurar o Kafka na nuvem e gerenciá-lo por meio de fornecedores como o Confluent ou até mesmo por meio do próprio Kafka do Azure no HDInsight, mas a seguir estão alguns conselhos simples se você planeja gerenciar seus próprios clusters Kafka manualmente. . Na maioria dos ambientes de nuvem, você tem uma seleção de muitas instâncias de computação, cada uma com uma combinação diferente de CPU, memória, IOPS e disco. As diversas características de desempenho do Kafka devem ser priorizadas para selecionar a configuração de instância correta a ser usada.

Microsoft Azure

No Azure, você pode gerenciar os discos separadamente da máquina virtual (VM), portanto, a decisão sobre suas necessidades de armazenamento não precisa estar relacionada ao tipo de VM selecionado. Dito isto, um bom ponto de partida para tomar decisões é com a quantidade de retenção de dados necessária, seguida pelo desempenho necessário dos produtores. Se for necessária uma latência muito baixa, poderão ser necessárias instâncias otimizadas de E/S que utilizem armazenamento SSD premium. Caso contrário, as opções de armazenamento gerido (como os Discos Geridos do Azure ou o Armazenamento de Blobs do Azure) poderão ser suficientes.

Em termos reais, a experiência no Azure mostra que Padrão D16s v3 os tipos de instância são uma boa escolha para clusters menores e têm desempenho suficiente para a maioria dos casos de uso. Para atender às necessidades de hardware e CPU de alto desempenho, D64s v4 as instâncias têm bom desempenho e podem ser escalonadas para clusters maiores. Recomenda-se construir seu cluster em um conjunto de disponibilidade do Azure e equilibrar partições em domínios de falha de computação do Azure para garantir a disponibilidade. Depois de escolher uma VM, a decisão sobre os tipos de armazenamento pode ser a próxima etapa. É altamente recomendável usar Discos Gerenciados do Azure em vez de discos efêmeros. Se uma VM for movida, você corre o risco de perder todos os dados do seu corretor Kafka. Os discos gerenciados HDD são relativamente baratos, mas não possuem SLAs claramente definidos da Microsoft sobre disponibilidade. As configurações de SSDs Premium ou Ultra SSD são muito mais caras, mas são muito mais rápidas e têm bom suporte com SLAs de 99,99% da Microsoft. Como alternativa, usar o Microsoft Blob Storage é uma opção se você não for tão sensível à latência.

Amazon Web Services

Na AWS, se for necessária uma latência muito baixa, poderão ser necessárias instâncias otimizadas de E/S que tenham armazenamento SSD local. Caso contrário, o armazenamento efêmero (como o Amazon Elastic Block Store) poderá ser suficiente.

Uma escolha comum na AWS é a m4 ou r3 tipos de instância. O m4 permitirá períodos de retenção maiores, mas a taxa de transferência para o disco será menor porque ele está no armazenamento em bloco elástico. O r3 A instância terá um rendimento muito melhor com unidades SSD locais, mas essas unidades limitarão a quantidade de dados que podem ser retidos. Para obter o melhor dos dois mundos, pode ser necessário avançar para o i2 ou d2 tipos de instância, mas são significativamente mais caros.

Configurando clusters Kafka

Um único corretor Kafka funciona bem para trabalho de desenvolvimento local ou para um sistema de prova de conceito, mas há benefícios significativos em ter vários corretores configurados como um cluster, como mostrado em **Figura 2-2**. O maior benefício é a capacidade de dimensionar a carga em vários servidores. Em segundo lugar está o uso da replicação para proteção contra perda de dados devido a falhas de um único sistema. A replicação também permitirá realizar

36 | Capítulo 2: Instalando o Kafka

trabalho de manutenção no Kafka ou nos sistemas subjacentes, mantendo a disponibilidade para os clientes. Esta seção concentra-se nas etapas para configurar um cluster básico do Kafka. **Capítulo 7** contém mais informações sobre replicação de dados e durabilidade.

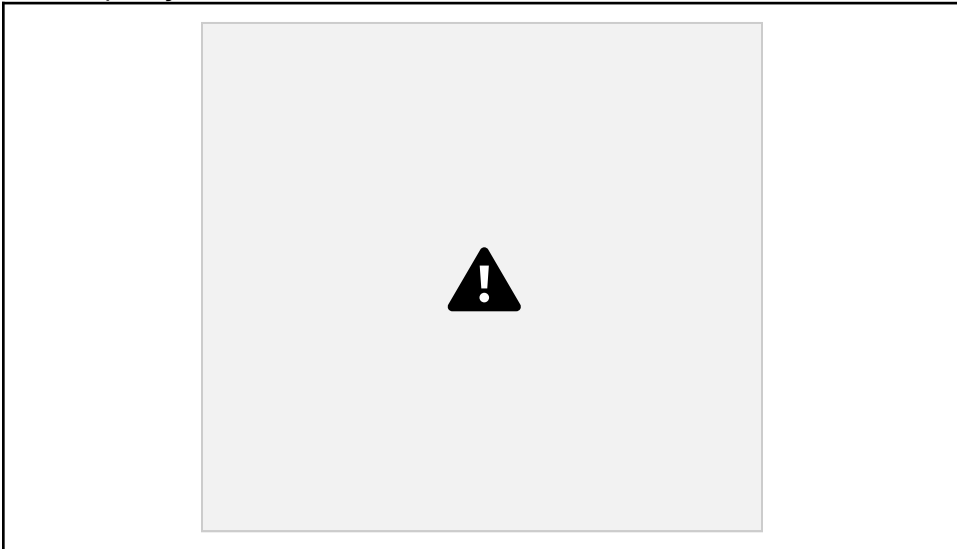


Figura 2-2. Um cluster Kafka simples

Quantos corretores?

O tamanho apropriado para um cluster Kafka é determinado por vários fatores. Normalmente, o tamanho do seu cluster estará limitado às seguintes áreas principais:

- Capacidade do disco
- Capacidade de réplica por corretor
- Capacidade da CPU
- Capacidade de rede

O primeiro fator a considerar é quanta capacidade de disco é necessária

para reter mensagens e quanto armazenamento está disponível em um único broker. Se for necessário que o cluster retenha 10 TB de dados e um único agente possa armazenar 2 TB, o tamanho mínimo do cluster será de 5 agentes. Além disso, aumentar o fator de replicação aumentará os requisitos de armazenamento em pelo menos 100%, dependendo da configuração do fator de replicação escolhida (consulte [Capítulo 7](#)). As réplicas, neste caso, referem-se ao número de corretores diferentes para os quais uma única partição é copiada. Isso significa que esse mesmo cluster, configurado com replicação de 2, agora precisa conter pelo menos 10 brokers.

O outro fator a considerar é a capacidade do cluster de lidar com solicitações. Isso pode ser demonstrado pelos outros três gargalos mencionados anteriormente.

Configurando clusters Kafka | 37

Se você tiver um cluster Kafka com 10 agentes, mas tiver mais de 1 milhão de réplicas (ou seja, 500.000 partições com um fator de replicação de 2) em seu cluster, cada agente estará assumindo aproximadamente 100.000 réplicas em um cenário uniformemente equilibrado. Isso pode levar a gargalos nas filas de produção, consumo e controlador. No passado, as recomendações oficiais eram não ter mais do que 4.000 réplicas de partição por corretor e não mais do que 200.000 réplicas de partição por cluster. No entanto, os avanços na eficiência do cluster permitiram que Kafka crescesse muito mais. Atualmente, em um ambiente bem configurado, é recomendado não ter mais de 14.000 réplicas de partição por broker e 1 milhão ***réplicas*** por cluster.

Conforme mencionado anteriormente neste capítulo, a CPU geralmente não é um grande gargalo para a maioria dos casos de uso, mas pode ser se houver uma quantidade excessiva de conexões e solicitações de clientes em um intermediário. Ficar de olho no uso geral da CPU com base em quantos clientes e grupos de consumidores exclusivos existem e expandir para atender a essas necessidades pode ajudar a garantir um melhor desempenho geral em grandes clusters. Falando em capacidade de rede, é importante ter em mente a capacidade das interfaces de rede e se elas podem lidar com o tráfego do cliente se houver múltiplos consumidores de dados ou se o tráfego não for consistente durante o período de retenção dos dados (por exemplo, , picos de tráfego durante horários de pico). Se a interface de rede em um único intermediário for usada com 80% da capacidade no pico e houver dois consumidores desses dados, os consumidores não conseguirão acompanhar o tráfego de pico, a menos que haja dois intermediários. Se a replicação estiver sendo usada no cluster, este é um consumidor adicional dos dados que deve ser levado em consideração. Você também pode querer expandir para mais agentes em um cluster para lidar com problemas de desempenho causados por menor rendimento de disco ou memória de sistema disponível.

Configuração do corretor

Existem apenas dois requisitos na configuração do agente para permitir que vários agentes Kafka ingressem em um único cluster. A primeira é que todos

os corretores devem ter a mesma configuração para o tratador do `zooógico.connect` parâmetro. Isso especifica o conjunto do ZooKeeper e o caminho onde o cluster armazena metadados. O segundo requisito é que todos os corretores no cluster tenham um valor exclusivo para o `corretor.id` parâmetro. Se dois corretores tentarem ingressar no mesmo cluster com o mesmo `corretor.id`, o segundo corretor registrará um erro e não será iniciado. Existem outros parâmetros de configuração usados ao executar um cluster — especificamente, parâmetros que controlam a replicação, que serão abordados em capítulos posteriores.

Ajuste do sistema operacional

Embora a maioria das distribuições Linux tenha uma configuração pronta para uso para os parâmetros de ajuste do kernel que funcionarão razoavelmente bem para a maioria dos aplicativos, há algumas alterações que podem ser feitas em um corretor Kafka que melhorarão o desempenho. Esses

38 | Capítulo 2: Instalando o Kafka

giram principalmente em torno da memória virtual e dos subsistemas de rede, bem como de preocupações específicas para o ponto de montagem do disco usado para armazenar segmentos de log. Esses parâmetros normalmente são configurados no **`/etc/sysctl.conf`** arquivo, mas você deve consultar a documentação da distribuição Linux para obter detalhes específicos sobre como ajustar a configuração do kernel.

Memória virtual

Em geral, o sistema de memória virtual Linux se ajustará automaticamente à carga de trabalho do sistema. Podemos fazer alguns ajustes na forma como o espaço de troca é tratado, bem como nas páginas de memória suja, para ajustá-las à carga de trabalho do Kafka.

Tal como acontece com a maioria dos aplicativos, especificamente aqueles em que o rendimento é uma preocupação, é melhor evitar a troca a (quase) todos os custos. O custo incorrido pela troca de páginas de memória para o disco aparecerá como um impacto perceptível em todos os aspectos do desempenho no Kafka. Além disso, Kafka faz uso intenso do cache de páginas do sistema e, se o sistema VM estiver trocando para disco, não há memória suficiente alocada para o cache de páginas.

Uma maneira de evitar a troca é simplesmente não configurar nenhum espaço de troca. Ter swap não é um requisito, mas fornece uma rede de segurança caso algo catastrófico aconteça no sistema. Ter swap pode evitar que o sistema operacional interrompa abruptamente um processo devido a uma condição de falta de memória. Por esse motivo, a recomendação é definir o `vm.swappiness` parâmetro para um valor muito baixo, como 1. O parâmetro é uma porcentagem da probabilidade de o subsistema da VM usar espaço de troca em vez de eliminar páginas do cache de páginas. É preferível reduzir a quantidade de memória disponível para o cache de páginas em vez de utilizar qualquer quantidade de memória swap.



Por que não definir a troca como zero?

Anteriormente, a recomendação de `vm.swappiness` sempre foi para defini-lo como 0. Este valor costumava significar “não troque a menos que haja um condição de falta de memória.” No entanto, o significado deste valor mudou a partir da versão 3.5-rc1 do kernel Linux, e essa mudança foi portado para muitas distribuições, incluindo Red Hat Enterprise Kernels Linux a partir da versão 2.6.32-303. Isso mudou o significado do valor 0 para “nunca trocar em nenhuma circunstância”. É por isso um valor de 1 agora é recomendado.

Também há um benefício em ajustar como o kernel lida com páginas sujas que devem ser descarregadas no disco. Kafka depende do desempenho de E/S de disco para fornecer bons tempos de resposta aos produtores. Esta é também a razão pela qual os segmentos de log são geralmente colocados em um disco rápido, seja um disco individual com um tempo de resposta rápido (por exemplo, SSD) ou um subsistema de disco com NVRAM significativo para armazenamento em cache (por exemplo, RAID). O resultado é que o número de páginas sujas permitidas antes do início do processo de liberação em segundo plano

Configurando clusters Kafka | 39

gravá-los no disco pode ser reduzido. Faça isso configurando o `vm.dirty_back ground_ratio` valor inferior ao padrão de 10. O valor é uma porcentagem da quantidade total de memória do sistema e definir esse valor como 5 é apropriado em muitas situações. No entanto, essa configuração não deve ser definida como zero, pois isso faria com que o kernel liberasse páginas continuamente, o que eliminaria a capacidade do kernel de armazenar em buffer gravações em disco contra picos temporários no desempenho do dispositivo subjacente.

O número total de páginas sujas permitidas antes que o kernel force operações síncronas para liberá-las no disco também pode ser aumentado alterando o valor de `vm.dirty_ratio` acima do padrão de 20 (também uma porcentagem da memória total do sistema). Existe uma ampla gama de valores possíveis para esta configuração, mas entre 60 e 80 é um número razoável. Essa configuração apresenta um pequeno risco, tanto em relação à quantidade de atividade de disco não liberada quanto ao potencial para longas pausas de E/S se liberações síncronas forem forçadas. Se uma configuração mais alta para `vm.dirty_ratio` for escolhido, é altamente recomendável que a replicação seja usada no cluster Kafka para proteção contra falhas do sistema.

Ao escolher valores para esses parâmetros, é aconselhável revisar o número de páginas sujas ao longo do tempo enquanto o cluster Kafka está sendo executado sob carga, seja em produção ou simulado. O número atual de páginas sujas pode ser determinado verificando o **`/proc/vmstat`** arquivo:

```
# gato /proc/vmstat | egrep "sujo | write-back"
nr_sujo 21845
```



```
nr_writeback 0
nr_writeback_temp 0
nr_dirty_threshold 32715981
nr_dirty_background_threshold 2726331
#
```

Kafka usa descritores de arquivo para segmentos de log e conexões abertas. Se um corretor tiver muitas partições, então esse corretor precisa de pelo menos **(número_de_partições)** × **(tamanho_da_partição/tamanho_do_segmento)** para rastrear todos os segmentos de log, além do número de conexões que o corretor faz. Como tal, recomenda-se atualizar o `vm.max_map_count` para um número muito grande com base no cálculo acima. Dependendo do ambiente, a alteração deste valor para 400.000 ou 600.000 geralmente tem sido bem-sucedida. Também é recomendado definir `vm.overcommit_memory` como 0. Definir o valor padrão 0 indica que o kernel determina a quantidade de memória livre de um aplicativo. Se a propriedade for definida com um valor diferente de zero, isso poderá fazer com que o sistema operacional ocupe muita memória, privando a memória para que o Kafka opere de maneira ideal. Isso é comum para aplicativos com altas taxas de ingestão.

40 | Capítulo 2: Instalando o Kafka

Disco

Além de selecionar o hardware do dispositivo de disco, bem como a configuração do RAID, se for usado, a escolha do sistema de arquivos para este disco pode ter o próximo maior impacto no desempenho. Existem muitos sistemas de arquivos diferentes disponíveis, mas as opções mais comuns para sistemas de arquivos locais são Ext4 (quarto sistema de arquivos estendido) ou Extents File System (XFS). O XFS se tornou o sistema de arquivos padrão para muitas distribuições Linux, e isso tem um bom motivo: ele supera o Ext4 para a maioria das cargas de trabalho com ajuste mínimo necessário. O Ext4 pode funcionar bem, mas requer o uso de parâmetros de ajuste considerados menos seguros. Isso inclui definir o intervalo de confirmação para um tempo maior que o padrão de cinco para forçar liberações menos frequentes. O Ext4 também introduziu a alocação atrasada de blocos, o que traz consigo uma maior chance de perda de dados e corrupção do sistema de arquivos em caso de falha do sistema. O sistema de arquivos XFS também usa um algoritmo de alocação atrasada, mas geralmente é mais seguro que aquele usado pelo Ext4. O XFS também tem melhor desempenho para a carga de trabalho do Kafka sem exigir ajuste além do ajuste automático realizado pelo sistema de arquivos. Também é mais eficiente ao agrupar gravações em disco, todas combinadas para fornecer melhor rendimento geral de E/S.

Independentemente de qual sistema de arquivos for escolhido para a montagem que contém os segmentos de log, é aconselhável definir o

`noatime` opção de montagem para o ponto de montagem. Os metadados do arquivo contêm três carimbos de data/hora: hora de criação (`ctime`), hora da última modificação (`mtime`) e hora do último acesso (de vez em quando). Por padrão, o de vez em quando é atualizado toda vez que um arquivo é lido. Isso gera um grande número de gravações em disco. O de vez em quando atributo é geralmente considerado de pouca utilidade, a menos que um aplicativo precise saber se um arquivo foi acessado desde a última modificação (nesse caso, o atributo `relatime` opção pode ser usada). O de vez em quando não é usado pelo Kafka, portanto, desativá-lo é seguro. Contexto `noatime` na montagem impedirá que essas atualizações de carimbo de data/hora aconteçam, mas não afetará o manuseio adequado do `ctime` e `mtime` atributos. Usando a opção `grandeio` também pode ajudar a melhorar a eficiência do Kafka quando há gravações maiores no disco.

Rede

Ajustar o ajuste padrão da pilha de rede do Linux é comum para qualquer aplicativo que gera uma grande quantidade de tráfego de rede, pois o kernel não é ajustado por padrão para transferências de dados grandes e de alta velocidade. Na verdade, as alterações recomendadas para Kafka são as mesmas sugeridas para a maioria dos servidores web e outros aplicativos de rede. O primeiro ajuste é alterar a quantidade padrão e máxima de memória alocada para os buffers de envio e recebimento de cada soquete. Isto aumentará significativamente o desempenho para grandes transferências. Os parâmetros relevantes para o tamanho padrão do buffer de envio e recebimento por soquete são `net.core.wmem_default` e `net.core.rmem_default`, e uma configuração razoável para esses parâmetros é 131072 ou 128 KiB. Os parâmetros para os tamanhos máximos do buffer de envio e recebimento são `net.core.wmem_max` e `net.core.rmem_max`, e uma configuração razoável é 2097152, ou

Configurando clusters Kafka | 41

2MiB. Tenha em mente que o tamanho máximo não indica que cada soquete terá tanto espaço de buffer alocado; só permite até esse limite, se necessário.

Além das configurações de soquete, os tamanhos de buffer de envio e recebimento para soquetes TCP devem ser definidos separadamente usando o comando `net.ipv4.tcp_wmem` e `net.ipv4.tcp_rmem` parâmetros. Eles são definidos usando três números inteiros separados por espaço que especificam os tamanhos mínimo, padrão e máximo, respectivamente. O tamanho máximo não pode ser maior que os valores especificados para todos os soquetes usando `net.core.wmem_max` e `net.core.rmem_max`. Um exemplo de configuração para cada um desses parâmetros é “4096 65536 2048000”, que é um buffer mínimo de 4 KiB, padrão de 64 KiB e buffer máximo de 2 MiB. Com base na carga de trabalho real dos seus agentes Kafka, você pode querer aumentar os tamanhos máximos para permitir um maior buffer das conexões de rede.

Existem vários outros parâmetros de ajuste de rede que são úteis para definir. Habilitando o dimensionamento da janela TCP configurando

`net.ipv4.tcp_window_scaling` para 1 permitirá que os clientes transfiram dados com mais eficiência e permitirá que esses dados sejam armazenados em buffer no lado do corretor. Aumentando o valor de `net.ipv4.tcp_max_syn_backlog` acima do padrão 1024 permitirá que um número maior de conexões simultâneas seja aceito. Aumentando o valor de `net.core.netdev_max_backlog` maior que o padrão de 1000 pode ajudar com picos de tráfego de rede, especificamente ao usar velocidades de conexão de rede multigigabit, permitindo que mais pacotes sejam enfileirados para que o kernel os processe.

Preocupações de produção

Quando você estiver pronto para tirar seu ambiente Kafka dos testes e colocá-lo em suas operações de produção, há mais algumas coisas em que pensar que ajudarão na configuração de um serviço de mensagens confiável.

Opções de coletor de lixo

Ajustar as opções de coleta de lixo Java para um aplicativo sempre foi uma espécie de arte, exigindo informações detalhadas sobre como o aplicativo usa a memória e uma quantidade significativa de observação e tentativa e erro. Felizmente, isso mudou com o Java 7 e a introdução do coletor de lixo Garbage-First (G1GC). Embora o G1GC tenha sido considerado instável inicialmente, ele viu uma melhoria acentuada no JDK8 e no JDK11. Agora é recomendado que Kafka use G1GC como coletor de lixo padrão. O G1GC foi projetado para se ajustar automaticamente a diferentes cargas de trabalho e fornecer tempos de pausa consistentes para coleta de lixo durante a vida útil do aplicativo. Ele também lida com grandes tamanhos de heap com facilidade, segmentando o heap em zonas menores e não coletando todo o heap em cada pausa.

42 | Capítulo 2: Instalando o Kafka

O G1GC faz tudo isso com um mínimo de configuração em operação normal. Existem duas opções de configuração do G1GC usadas para ajustar seu desempenho:

`MaxGCPauseMillis`

Esta opção especifica o tempo de pausa preferencial para cada ciclo de coleta de lixo. Não é um máximo fixo – o G1GC pode e irá exceder esse tempo, se necessário. Esse valor é padronizado como 200 milissegundos. Isso significa que o G1GC tentará agendar a frequência dos ciclos do coletor de lixo, bem como o número de zonas que são coletadas em cada ciclo, de forma que cada ciclo leve aproximadamente 200 ms.

`IniciandoHeapOccupancyPercent`

Esta opção especifica a porcentagem do heap total que pode estar em

uso antes do G1GC iniciar um ciclo de coleta. O valor padrão é 45. Isso significa que o G1GC não iniciará um ciclo de coleta até que 45% do heap esteja em uso. Isso inclui o uso da zona nova (Eden) e antiga, no total.

O corretor Kafka é bastante eficiente na forma como utiliza a memória heap e cria objetos de lixo, portanto, é possível definir essas opções para valores mais baixos. As opções de ajuste do coletor de lixo fornecidas nesta seção foram consideradas apropriadas para um servidor com 64 GB de memória, executando o Kafka em um heap de 5 GB. Para `MaxGCPauseMillis`, esse broker pode ser configurado com um valor de 20 ms. O valor para `IniciandoHeapOccupancyPercent` é definido como 35, o que faz com que a coleta de lixo seja executada um pouco antes do valor padrão.

Kafka foi lançado originalmente antes que o coletor G1GC estivesse disponível e fosse considerado estável. Portanto, o padrão do Kafka é usar a coleta de lixo simultânea de marcação e varredura para garantir a compatibilidade com todas as JVMs. A nova prática recomendada é usar G1GC para qualquer coisa no Java 1.8 e posterior. A mudança é fácil de fazer por meio de variáveis de ambiente. Usando o `começar` comando do início do capítulo, modifique-o da seguinte forma:

```
# exportar KAFKA_JVM_PERFORMANCE_OPTS="-servidor -Xmx6g -Xms6g
-XX:Tamanho do metaespaço=96m -XX:+UsarG1GC
-XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35
-XX:G1HeapRegionSize=16M -XX:MinMetaspaceFreeRatio=50
-XX:MaxMetaspaceFreeRatio=80 -XX:+ExplicitGCInvokesConcurrent"
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

Layout do datacenter

Para ambientes de teste e desenvolvimento, a localização física dos agentes Kafka dentro de um datacenter não é tão preocupante, pois não há um impacto tão grave se o cluster estiver parcial ou totalmente indisponível por curtos períodos de tempo. No entanto, ao atender o tráfego de produção, o tempo de inatividade geralmente significa perda de dinheiro, seja pela perda de serviços aos usuários ou pela perda de telemetria sobre o que os usuários estão fazendo. Esse

Preocupações com a produção | 43

é quando se torna crítico configurar a replicação dentro do cluster Kafka (consulte [Capítulo 7](#)), que também é importante considerar a localização física dos brokers em seus racks no datacenter. É preferível um ambiente de datacenter que tenha um conceito de zonas de falha. Se não for resolvido antes da implantação do Kafka, poderá ser necessária uma manutenção cara para mover os servidores.

Kafka pode atribuir novas partições a brokers com reconhecimento de rack, garantindo que as réplicas de uma única partição não compartilhem um rack. Para fazer isso, o `corretor.rack` a configuração de cada corretor deve ser definida corretamente. Essa configuração também pode ser definida para o

domínio de falha em ambientes de nuvem por motivos semelhantes. No entanto, isso se aplica apenas a partições recém-criadas. O cluster Kafka não monitora partições que não reconhecem mais o rack (por exemplo, como resultado de uma reatribuição de partição) nem corrige automaticamente essa situação. É recomendável usar ferramentas que mantenham seu cluster balanceado adequadamente para manter o reconhecimento do rack, como Cruise Control (consulte [Apêndice B](#)). Configurar isso corretamente ajudará a garantir o reconhecimento contínuo do rack ao longo do tempo.

No geral, a prática recomendada é ter cada corretor Kafka em um cluster instalado em um rack diferente ou, pelo menos, não compartilhar pontos únicos de falha para serviços de infraestrutura, como energia e rede. Isso normalmente significa pelo menos implantar os servidores que executarão intermediários com conexões de energia duplas (para dois circuitos diferentes) e switches de rede duplos (com uma interface vinculada nos próprios servidores para fazer failover sem problemas). Mesmo com conexões duplas, há uma vantagem em ter intermediários em racks completamente separados. De tempos em tempos, pode ser necessário realizar manutenção física em um rack ou gabinete que exija que ele esteja off-line (como mover servidores ou religar conexões de energia).

Colocando aplicativos no ZooKeeper

Kafka utiliza ZooKeeper para armazenar informações de metadados sobre corretores, tópicos e partições. As gravações no ZooKeeper são executadas apenas em alterações na associação de grupos de consumidores ou em alterações no próprio cluster Kafka. Essa quantidade de tráfego é geralmente mínima e não justifica o uso de um conjunto ZooKeeper dedicado para um único cluster Kafka. Na verdade, muitas implantações usarão um único conjunto ZooKeeper para vários clusters Kafka (usando um caminho chroot ZooKeeper para cada cluster, conforme descrito anteriormente neste capítulo).

44 | Capítulo 2: Instalando o Kafka

Consumidores Kafka, ferramentas, ZooKeeper e você

Com o passar do tempo, a dependência do ZooKeeper está diminuindo. Em versão

versão 2.8.0, Kafka está introduzindo uma visão de acesso antecipado em um ambiente completamente

Kafka sem ZooKeeper, mas ainda não está pronto para produção. No entanto, ainda podemos ver essa dependência reduzida do ZooKeeper nas versões levando a isso. Por exemplo, em versões mais antigas do Kafka, con-



consumidores (além dos corretores) utilizaram o ZooKeeper para diretamente armazenar informações sobre a composição do grupo de consumidores e quais tópicos ele estava consumindo, e se comprometer periodicamente conjuntos para cada partição que está sendo consumida (para permitir o failover entre consumidores do grupo). Com a versão 0.9.0.0, o consumidor interface foi alterada, permitindo que isso seja gerenciado diretamente com o Corretores Kafka. Em cada versão 2.x do Kafka, vemos etapas adicionais para remover o ZooKeeper de outros caminhos obrigatórios do Kafka. As ferramentas de administração agora se conectam diretamente ao cluster e têm obsoleto a necessidade de conectar-se ao ZooKeeper diretamente para operações como criações de tópicos, alterações dinâmicas de configuração, etc. Como tal, muitas das ferramentas de linha de comando que anteriormente usavam o `--zookeeper` sinalizadores foram atualizados para usar o `--bootstrap` servidor opção. O `--zookeeper` opções ainda podem ser usadas, mas têm sido descontinuado e será removido no futuro, quando Kafka for não é mais necessário se conectar ao ZooKeeper para criar, gerenciar ou consumir de tópicos.

No entanto, existe uma preocupação com os consumidores e com o ZooKeeper sob certas configurações. Embora o uso do ZooKeeper para tais fins esteja obsoleto, os consumidores têm a opção configurável de usar o ZooKeeper ou o Kafka para confirmar compensações e também podem configurar o intervalo entre as confirmações. Se o consumidor usar o ZooKeeper para compensações, cada consumidor executará uma gravação do ZooKeeper a cada intervalo para cada partição que consumir. Um intervalo razoável para commits de compensação é de 1 minuto, pois este é o período de tempo durante o qual um grupo de consumidores lerá mensagens duplicadas no caso de falha do consumidor. Esses commits podem representar uma quantidade significativa de tráfego do ZooKeeper, especialmente em um cluster com muitos consumidores, e precisarão ser levados em consideração. Pode ser necessário usar um intervalo de commit mais longo se o conjunto Zoo-Keeper não for capaz de lidar com o tráfego. No entanto, é recomendado que os consumidores que usam as bibliotecas Kafka mais recentes usem o Kafka para confirmar compensações, removendo a dependência do ZooKeeper.

Além de usar um único conjunto para vários clusters Kafka, não é recomendado compartilhar o conjunto com outros aplicativos, se isso puder ser evitado. Kafka é sensível à latência e aos tempos limite do ZooKeeper, e uma interrupção nas comunicações com o conjunto fará com que os corretores se comportem de maneira imprevisível. Isso pode facilmente fazer com que vários corretores fiquem off-line ao mesmo tempo, caso percam as conexões do ZooKeeper, o que resultará em partições off-line. Isso também sobrecarrega o controlador de cluster,

Preocupações com a produção | 45

que podem aparecer como erros sutis muito depois de a interrupção ter passado, como ao tentar executar um desligamento controlado de um corretor. Outros aplicativos que podem sobrecarregar o conjunto ZooKeeper, seja por uso intenso ou operações inadequadas, devem ser segregados em

seu próprio conjunto.

Resumo

Neste capítulo, aprendemos como colocar o Apache Kafka em funcionamento. Também abordamos a escolha do hardware certo para seus corretores e questões específicas sobre a configuração em um ambiente de produção. Agora que você tem um cluster Kafka, examinaremos os conceitos básicos dos aplicativos cliente Kafka. Os próximos dois capítulos abordarão como criar clientes para produzir mensagens para Kafka (**Capítulo 3**) bem como consumir essas mensagens novamente (**Capítulo 4**).

Quer você use o Kafka como fila, barramento de mensagens ou plataforma de armazenamento de dados, você sempre usará o Kafka criando um produtor que grava dados no Kafka, um consumidor que lê dados do Kafka ou um aplicativo que atende a ambas as funções.

Por exemplo, num sistema de processamento de transações de cartão de crédito, haverá uma aplicação cliente, talvez uma loja online, responsável por enviar cada transação ao Kafka imediatamente quando o pagamento for efetuado. Outro aplicativo é responsável por verificar imediatamente esta transação em relação a um mecanismo de regras e determinar se a transação foi aprovada ou negada. A resposta de aprovação/negação pode então ser escrita de volta no Kafka e a resposta pode ser propagada de volta para a loja online onde a transação foi iniciada. Um terceiro aplicativo pode ler as transações e o status de aprovação do Kafka e armazená-los em um banco de dados onde os analistas podem posteriormente revisar as decisões e talvez melhorar o mecanismo de regras.

O Apache Kafka vem com APIs de cliente integradas que os desenvolvedores podem usar ao desenvolver aplicativos que interagem com o Kafka.

Neste capítulo aprenderemos como usar o produtor Kafka, começando com uma visão geral de seu design e componentes. Mostraremos como criar `Produtor Kafka` e `ProdutorRecord` objetos, como enviar registros para o Kafka e como lidar com os erros que o Kafka pode retornar. Em seguida, revisaremos as opções de configuração mais importantes usadas para controlar o comportamento do produtor. Concluiremos com uma visão mais aprofundada de como usar diferentes métodos de particionamento e serializadores e como escrever seus próprios serializadores e particionadores.

Em **Capítulo 4**, veremos o cliente consumidor de Kafka e leremos os dados

e

K

a

f

k

a

.

4

7

Cientes terceirizados

Além dos clientes integrados, o Kafka possui um protocolo de ligação binária.

Isso significa que é possível que aplicativos leiam mensagens do Kafka ou escreva mensagens para Kafka simplesmente enviando o correto sequência de bytes para a porta de rede do Kafka. Existem vários clientes que implementam o protocolo wire de Kafka em diferentes linguagens de programação, oferecendo maneiras simples de usar o Kafka, não apenas em aplicativos Java, mas também em linguagens como C++, Python, Go, e muito mais. Esses clientes não fazem parte do Apache Kafka projeto, mas uma lista de clientes não-Java é mantida no [projeto semana](#). O protocolo de conexão e os clientes externos estão fora do escopo do capítulo.

Visão geral do produtor

Há muitos motivos pelos quais um aplicativo pode precisar gravar mensagens no Kafka: registrar as atividades do usuário para auditoria ou análise, registrar métricas, armazenar mensagens de log, registrar informações de dispositivos inteligentes, comunicar-se de forma assíncrona com outros aplicativos, armazenar informações em buffer antes de gravar em um banco de dados. e muito mais.

Esses diversos casos de uso também implicam diversos requisitos: cada mensagem é crítica ou podemos tolerar a perda de mensagens? Estamos bem com a duplicação acidental de mensagens? Há algum requisito rigoroso de latência ou taxa de transferência que precisamos oferecer suporte?

No exemplo de processamento de transações de cartão de crédito que apresentamos anteriormente, podemos ver que é fundamental nunca perder uma única mensagem ou duplicar nenhuma mensagem. A latência deve ser baixa, mas latências de até 500 ms podem ser toleradas e a taxa de transferência deve ser muito alta – esperamos processar até um milhão de mensagens por segundo.

Um caso de uso diferente pode ser armazenar informações de cliques de um site. Nesse caso, alguma perda de mensagem ou algumas duplicatas podem ser toleradas; a latência pode ser alta, desde que não haja impacto na experiência do usuário. Em outras palavras, não nos importamos se a mensagem demorar alguns segundos para chegar ao Kafka, desde que a próxima página carregue imediatamente após o usuário clicar em um link. O rendimento dependerá do nível de atividade que antecipamos em nosso site.

Os diferentes requisitos influenciarão a maneira como você usa a API do produtor para escrever mensagens no Kafka e a configuração que você usa.

Embora a API do produtor seja muito simples, há um pouco mais acontecendo nos bastidores do produtor quando enviamos dados. **Figura 3-1** mostra as principais etapas envolvidas no envio de dados para Kafka.

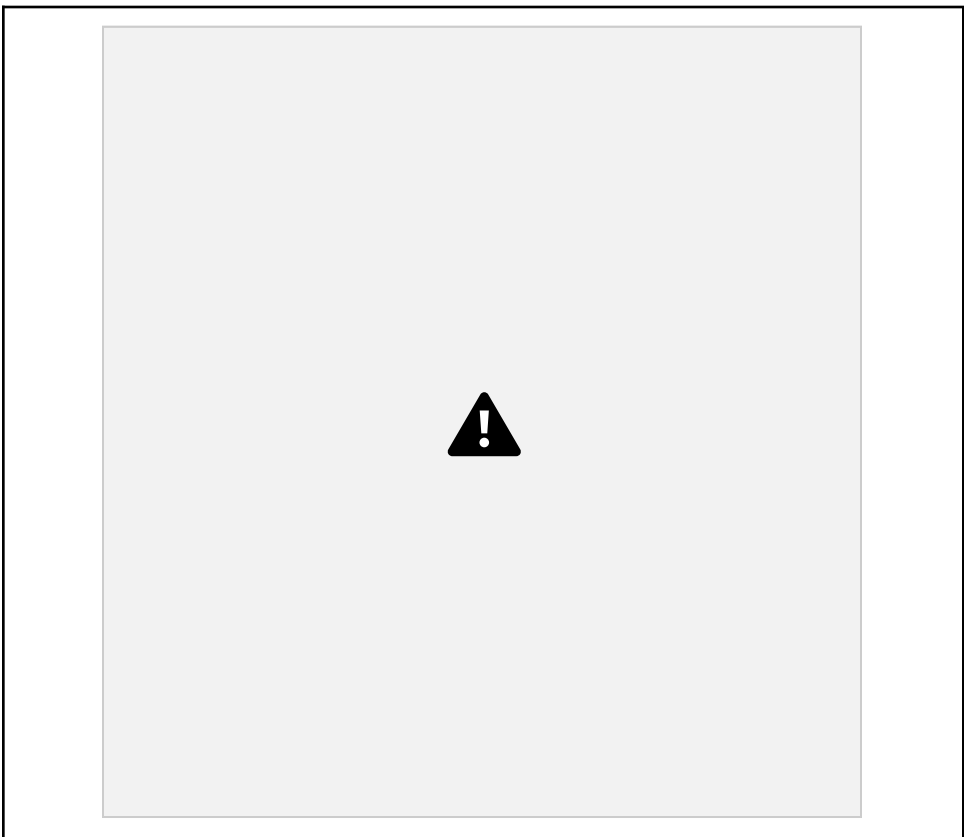


Figura 3-1. Visão geral de alto nível dos componentes do produtor Kafka

Começamos a produzir mensagens para Kafka criando um `ProdutorRecord`, que deve incluir o tópico para o qual queremos enviar o registro e um valor. Opcionalmente, também podemos especificar uma chave, uma partição, um carimbo de data/hora e/ou uma coleção de cabeçalhos. Assim que enviarmos o `ProdutorRecord`, a primeira coisa que o produtor fará é serializar os objetos de chave e valor em matrizes de bytes para que possam ser enviados pela rede.

A seguir, se não especificamos explicitamente uma partição, os dados são enviados para um particionador. O particionador escolherá uma partição para nós, geralmente com base no `ProdutorRecord` chave. Uma vez selecionada uma partição, o produtor sabe para qual tópico e partição o registro irá. Em seguida, adiciona o registro a um lote de registros que também será enviado para o mesmo tópico e partição. Um thread separado é responsável por enviar esses lotes de registros aos corretores Kafka apropriados.

Quando o corretor recebe as mensagens, ele devolve uma resposta. Se as mensagens foram gravadas com sucesso no Kafka, ele retornará um `RegistroMetadados` objeto com o tópico, a partição e o deslocamento do registro dentro da partição. Se o corretor falhou

para escrever as mensagens, retornará um erro. Quando o produtor recebe um erro, ele pode tentar enviar a mensagem mais algumas vezes antes de desistir e retornar um erro.

Construindo um Produtor Kafka

A primeira etapa para escrever mensagens no Kafka é criar um objeto produtor com as propriedades que você deseja passar ao produtor. Um produtor Kafka possui três propriedades obrigatórias:

`bootstrap.servers`

Lista de anfitrião:porta pares de corretores que o produtor usará para estabelecer a conexão inicial com o cluster Kafka. Esta lista não precisa incluir todas as corretoras, pois o produtor obterá mais informações após a conexão inicial. Mas é recomendado incluir pelo menos dois, assim, caso um corretor fique inativo, o produtor ainda poderá se conectar ao cluster.

`chave.serializer`

Nome de uma classe que será usada para serializar as chaves dos registros que produziremos para o Kafka. Os corretores Kafka esperam matrizes de bytes como chaves e valores de mensagens. Porém, a interface do produtor permite, usando tipos parametrizados, qualquer objeto Java ser enviado como chave e valor. Isto torna o código muito legível, mas também significa que o produtor precisa saber como converter esses objetos em matrizes de bytes. `chave.serializer` deve ser definido como o nome de uma classe que implementa o `org.apache.kafka.common.serialization.Serializer` interface. O produtor usará esta classe para serializar o objeto chave em uma matriz de bytes. O pacote do cliente Kafka inclui `ByteArraySerializer` (o que não faz muito), `Serializador de string`, `InteiroSerializadore` muito mais, portanto, se você usar tipos comuns, não será necessário implementar seus próprios serializadores. Contexto `chave.serializer` é obrigatório mesmo que você pretenda enviar apenas valores, mas você pode usar o `Vazio` digite a chave e o `VoidSerializer`.

`valor.serializer`

Nome de uma classe que será usada para serializar os valores dos registros que produziremos para o Kafka. Da mesma forma que você definiu `chave.serializer` para o nome de uma classe que serializará o objeto-chave da mensagem em uma matriz de bytes, você define `valor.serializer` para uma classe que serializará o objeto de valor da mensagem.

O trecho de código a seguir mostra como criar um novo produtor definindo apenas os parâmetros obrigatórios e usando padrões para todo o resto:

```
Propriedades kafkaProps = new Propriedades();
```

```
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");

kafkaProps.put("key.serializer",
```

50 | Capítulo 3: Produtores Kafka: Escrevendo mensagens para Kafka

```
"org.apache.kafka.common.serialization.StringSerializer");
kafkaProps.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
```

```
produtor = new KafkaProducer<String, String>(kafkaProps);
```

Começamos com um `Properties` objeto.

Como planejamos usar strings para chave e valor da mensagem, usamos o método integrado `StringSerializer`.

Aqui criamos um novo produtor definindo os tipos de chave e valor apropriados e passando o `Properties` objeto.

Com uma interface tão simples, fica claro que a maior parte do controle sobre o comportamento do produtor é feita através da definição das propriedades de configuração corretas. A documentação do Apache Kafka cobre todos os **opções de configuração**, e abordaremos os mais importantes posteriormente neste capítulo.

Depois de instanciarmos um produtor, é hora de começar a enviar mensagens. Existem três métodos principais de envio de mensagens:

Dispare e esqueça

Enviamos uma mensagem para o servidor e não nos importamos se ela chega com sucesso ou não. Na maioria das vezes, ele chegará com sucesso, pois o Kafka está altamente disponível e o produtor tentará enviar novamente as mensagens automaticamente. Porém, em caso de erros não recuperáveis ou de tempo limite, as mensagens serão perdidas e a aplicação não obterá nenhuma informação ou exceção sobre isso.

Envio síncrono

Tecnicamente, o produtor Kafka é sempre assíncrono – enviamos uma mensagem e o `enviar()` método retorna um `Futuro` objeto. No entanto, usamos `get()` esperar no `Futuro` e veja se o `enviar()` foi bem-sucedido ou não antes de enviar o próximo registro.

Envio assíncrono

Chamamos o `enviar()` método com uma função de retorno de chamada, que é acionada quando recebe uma resposta do corretor Kafka.

Nos exemplos a seguir veremos como enviar mensagens usando esses métodos e como lidar com os diferentes tipos de erros que podem ocorrer.

Embora todos os exemplos neste capítulo sejam de thread único, um objeto produtor pode ser usado por vários threads para enviar mensagens.

Enviando uma mensagem para Kafka

A maneira mais simples de enviar uma mensagem é a seguinte:

```
ProdutorRecord<String, String> registro =
    new ProducerRecord<>("CustomerCountry", "Produtos de Precisão",
        "França");
tentar {
    produtor.send(registro);
} catch (Exceção e) {
    e.printStackTrace();
}
```

O produtor aceita `ProdutorRecord` objetos, então começamos criando um. `ProdutorRecord` tem vários construtores, que discutiremos mais tarde. Aqui usamos um que requer o nome do tópico para o qual estamos enviando os dados, que é sempre uma string, e a chave e o valor que estamos enviando para o Kafka, que neste caso também são strings. Os tipos de chave e valor devem corresponder aos nossos serializador de chave e serializador de valor objetos.

Usamos o objeto produtor `enviar()` método para enviar o `ProdutorRecord`. Como vimos no diagrama da arquitetura do produtor em **Figura 3-1**, a mensagem será colocada em um buffer e enviada ao corretor em um thread separado. O `enviar()` método retorna um **Java Futuro objeto** com `RegistroMetadados`, mas como simplesmente ignoramos o valor retornado, não temos como saber se a mensagem foi enviada com sucesso ou não. Este método de envio de mensagens pode ser usado quando descartar uma mensagem silenciosamente for aceitável. Normalmente, esse não é o caso em aplicativos de produção.

Embora ignoremos erros que podem ocorrer durante o envio de mensagens aos corretores Kafka ou nos próprios corretores, ainda podemos obter uma exceção se o produtor encontrar erros antes de enviar a mensagem ao Kafka. Podem ser, por exemplo, um `SerializaçãoException` quando falha ao serializar a mensagem, um `Buffer esgotadoException` ou `TimeoutException` se o buffer estiver cheio ou um `InterruptedException` se o thread de envio foi interrompido.

Enviando uma mensagem de forma síncrona

Enviar uma mensagem de forma síncrona é simples, mas ainda permite que o produtor capture exceções quando Kafka responde à solicitação de produção com um erro ou quando as novas tentativas de envio se esgotam. A principal compensação envolvida é o desempenho. Dependendo de quão ocupado o cluster Kafka está, os corretores podem levar de 2 ms a alguns segundos para responder às solicitações de produção. Se você enviar

mensagens de forma síncrona, o thread de envio gastará esse tempo esperando e não fazendo mais nada, nem mesmo enviando mensagens adicionais. Isso leva a um desempenho muito ruim e, como resultado,

52 | Capítulo 3: Produtores Kafka: Escrevendo mensagens para Kafka