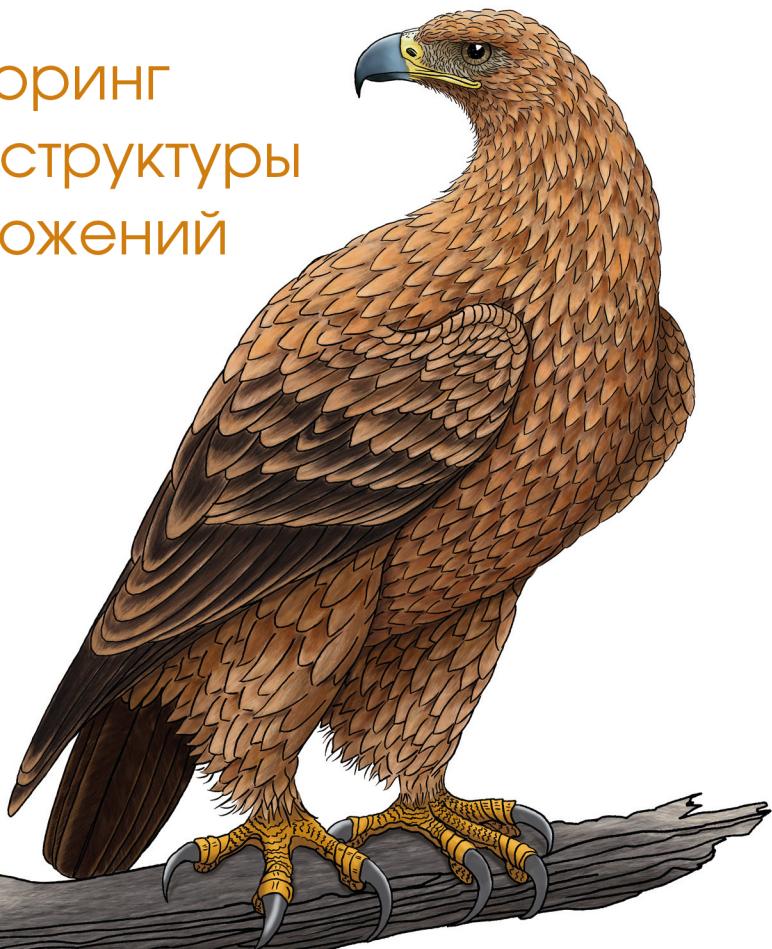


O'REILLY®

Запускаем Prometheus

Мониторинг
инфраструктуры
и приложений



Жюльен Пивотто
Брайан Бразил



Жюльен Пивотто, Брайан Бразил

Запускаем Prometheus

Julien Pivotto and Brian Brazil

Prometheus: Up & Running

Infrastructure and Application
Performance Monitoring

Second Edition



Beijing • Boston • Farnham • Sebastopol • Tokyo

Жюльен Пивотто, Брайан Бразил

Запускаем Prometheus

Мониторинг инфраструктуры
и приложений



**УДК 004.04
ББК 32.372
П32**

Пивотто Ж., Бразил Б.
П32 Запускаем Prometheus / пер. с англ. А. Н. Киселева. – М.: Books.kz, 2023. – 392 с.: ил.

ISBN 978-601-81034-1-4

Возьмите на вооружение Prometheus – систему мониторинга на основе метрик, используемую тысячами организаций. Эта книга расскажет о системе мониторинга на основе метрик Prometheus и познакомит с наиболее важными ее аспектами, включая создание дашбордов и оповещений, прямое инструментирование кода, а также извлечение метрик из сторонних систем с помощью экспортёров.

Авторы покажут, как использовать систему Prometheus для мониторинга приложений и инфраструктуры, как настроить Prometheus, Node Exporter и Alertmanager, как использовать эти инструменты для мониторинга приложений и инфраструктуры.

Издание адресовано инженерам по надежности сайтов, администраторам Kubernetes и разработчикам программного обеспечения.

**УДК 004.04
ББК 32.372**

Authorized Russian translation of the English edition of Prometheus: Up & Running, 2E ISBN 9781098131142 © 2023 Julien Pivotto. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-098-13114-2 (англ.)
ISBN 978-6-01810-341-4 (казах.)

© 2023 Julien Pivotto
© Перевод, оформление, издание,
Books.kz, 2023

Содержание

Вступление	12
Об авторах	15
Колофон	16
Часть I. ВВЕДЕНИЕ	17
Глава 1. Что такое Prometheus?	18
Что такое мониторинг?	19
Краткая и неполная история мониторинга	21
Категории мониторинга	22
Архитектура Prometheus	27
Клиентские библиотеки.....	27
Экспортёры	28
Обнаружение служб.....	29
Извлечение данных	30
Хранилище.....	30
Дашборды	30
Правила записи и оповещения	31
Управление уведомлениями	32
Долгосрочное хранение	32
Чем Prometheus не является.....	32
Глава 2. Настройка и запуск Prometheus	34
Запуск Prometheus	34
Использование браузера выражений.....	38
Запуск экспортёра узла	42
Оповещение.....	46
Часть II. МОНИТОРИНГ ПРИЛОЖЕНИЙ	53
Глава 3. Инструментирование	54
Простая программа.....	54
Счетчики.....	56
Подсчет исключений	58
Подсчет размера	59

Датчики	60
Использование датчиков	61
Обратные вызовы	62
Сводные метрики.....	63
Гистограммы.....	65
Сегменты.....	66
Модульное тестирование метрик	68
Подход к инструментированию.....	69
Что инструментировать?	70
Как широко использовать инструментирование?	72
Какие имена выбирать для метрик?	72
Глава 4. Экспортирование метрик.....	76
Python	77
WSGI	77
Twisted	78
Многозадачность с Gunicorn	79
Go	82
Java.....	83
HTTPServer	83
Сервлеты	85
Pushgateway.....	86
Мосты.....	90
Парсеры	90
Текстовый формат экспорта.....	91
Типы метрик.....	91
Метки	92
Экранирование.....	93
Отметки времени	93
Проверка метрик	94
OpenMetrics	94
Типы метрик.....	95
Метки	96
Отметки времени	96
Глава 5. Метки.....	97
Что такое метки?	97
Инструментированные и целевые метки.....	98
Инструментирование	99
Метрики	100
Несколько меток	101
Дочерние элементы.....	101
Агрегирование.....	103
Шаблоны использования меток.....	104
Метрики-перечисления	104
Информационные метрики	106

Когда использовать метки	108
Кардинальность	109
Глава 6. Создание дашбордов с Grafana	112
Установка.....	113
Источники данных.....	115
Дашборды и панели	116
Как избежать «стены графиков»	117
Панель временных рядов.....	117
Элементы управления временем	119
Панель статистики	121
Панель таблицы.....	123
Панель временной шкалы состояний	125
Переменные шаблона.....	126
Часть III. МОНИТОРИНГ ИНФРАСТРУКТУРЫ.....	130
Глава 7. Node Exporter.....	131
Сборщик информации о процессоре	132
Сборщик информации о файловой системе.....	133
Сборщик дисковой статистики.....	134
Сборщик информации о сетевых устройствах.....	135
Сборщик информации о потреблении памяти	136
Сборщик информации об аппаратной части	136
Сборщик статистики.....	137
Сборщик информации об имени узла	138
Сборщик информации об операционной системе	138
Сборщик информации о средней нагрузке	139
Сборщик информации о давлении	139
Сборщик текстовых файлов	140
Использование сборщика текстовых файлов.....	141
Отметки времени	143
Глава 8. Обнаружение служб.....	144
Механизмы обнаружения служб	145
Статическое обнаружение служб.....	146
Обнаружение служб на основе файла.....	147
Обнаружение служб через HTTP	150
Обнаружение служб с помощью Consul	151
Обнаружение служб с помощью EC2	152
Изменение меток	154
Выбор целей для мониторинга	154
Метки целей.....	157
Как извлекать метрики	165
metric_relabel_configs.....	167
Конфликты меток и honor_labels	169

Глава 9. Контейнеры и Kubernetes	170
Advisor	170
Метрики потребления процессора.....	171
Метрики потребления памяти.....	172
Метки.....	172
Kubernetes	173
Запуск в Kubernetes	173
Обнаружение служб.....	175
kube-state-metrics	184
Альтернативные развертывания.....	185
Глава 10. Популярные экспортеры.....	187
Consul.....	187
MySQLd	189
Grok Exporter	190
Blackbox	193
ICMP	194
TCP	199
HTTP.....	201
DNS.....	203
Настройка Prometheus	205
Глава 11. Работа с другими системами мониторинга.....	209
Другие системы мониторинга	209
InfluxDB.....	211
StatsD.....	212
Глава 12. Разработка экспортеров	215
Consul Telemetry	215
Пользовательские сборщики.....	219
Метки.....	223
Методические рекомендации	224
Часть IV. PROMQL	226
Глава 13. Введение в PromQL	227
Основы агрегирования.....	227
Датчики	227
Счетчики	229
Сводные метрики	230
Гистограммы	231
Селекторы.....	233
Сопоставители.....	234
Мгновенный вектор	235
Вектор диапазона	236

Подзапросы.....	238
Смещение.....	239
Модификатор @.....	240
HTTP API.....	240
query.....	240
query_range.....	243
Глава 14. Операторы агрегирования.....	246
Группировка.....	246
without	247
by	248
Операторы.....	249
sum	249
count.....	250
avg.....	251
group	252
stddev и stdvar.....	252
min и max	253
topk и bottomk	254
quantile	254
count_values	255
Глава 15. Двухместные операторы.....	258
Работа со скалярами	258
Арифметические операторы.....	259
Тригонометрический оператор.....	260
Операторы сравнения.....	260
Сопоставление векторов.....	262
Один к одному.....	263
Многие к одному и group_left	265
Многие ко многим и логические операторы.....	268
Приоритет операторов	272
Глава 16. Функции.....	273
Изменение типа	273
vector	273
scalar	274
Математические функции.....	275
abs.....	275
ln, log2 и log10	276
exp	276
sqrt.....	276
ceil и floor	277
round	277
clamp, clamp_max и clamp_min.....	278
sgn	278

Тригонометрические функции.....	278
Время и дата.....	279
time.....	279
Функции minute, hour, day_of_week, day_of_month, day_of_year, days_in_month, month и year.....	280
timestamp	281
Метки	282
label_replace	282
label_join	283
Отсутствующие серии, absent и absent_over_time	283
Сортировка с помощью sort и sort_desc.....	284
Гистограммы с histogram_quantile.....	285
Счетчики.....	285
rate.....	286
increase	287
irate	288
resets	289
Изменение датчиков	289
changes.....	289
deriv.....	290
predict_linear	290
delta.....	291
idelta.....	291
holt_winters	291
Агрегирование по времени	292
Глава 17. Правила записи.....	294
Использование правил записи.....	294
Когда использовать правила записи	297
Уменьшение кардинальности.....	297
Составление функций для обработки векторов диапазонов	299
Правила для API	300
Как не следует использовать правила	300
Именование правил записи	302
Часть V. ОПОВЕЩЕНИЕ	306
Глава 18. Оповещение	307
Правила оповещения.....	308
for.....	310
Метки оповещений.....	312
Аннотации и шаблоны	315
Что такое хорошее оповещение?.....	317
Настройка диспетчеров уведомлений в Prometheus	318
Внешние метки	319

Глава 19. Alertmanager	321
Конвойер уведомлений	321
Конфигурационный файл	322
Дерево маршрутизации	323
Приемники	330
Подавление	340
Веб-интерфейс Alertmanager.....	341
Часть VI. РАЗВЕРТЫВАНИЕ	344
Глава 20. Безопасность на стороне сервера	345
Функции безопасности в Prometheus	345
Включение TLS	345
Дополнительные параметры TLS	347
Включение базовой аутентификации.....	347
Глава 21. Собираем все вместе	350
Планирование развертывания.....	350
Расширение Prometheus	351
Выход на глобальный уровень с помощью механизма федерации	353
Долгосрочное хранение	356
Эксплуатация Prometheus.....	358
Аппаратное обеспечение.....	358
Управление конфигурацией	360
Сети и аутентификация	362
Планирование сбоев.....	363
Кластеризация диспетчеров уведомлений.....	366
Мета- и перекрестный мониторинг.....	367
Управление производительностью	368
Обнаружение проблем	368
Поиск долгостоящих метрик и целей	369
Снижение нагрузки	370
Горизонтальное сегментирование	371
Управление изменениями	373
Получение помощи.....	373
Предметный указатель.....	375

Вступление

В этой книге подробно описываются приемы работы с системой мониторинга Prometheus, позволяющей наблюдать, строить графики, характеризующие особенности работы приложений и инфраструктуры, и рассыпать оповещения в случае сбоев. Она предназначена для разработчиков приложений, системных администраторов и всех, кто находится между ними.

Увеличение осведомленности

Сопровождая системы, важно знать, что они запущены и работают, но истинная ценность заключается не в этом. Гораздо важнее иметь представление об особенностях работы систем.

Под особенностями работы мы подразумеваем не только производительность – время отклика и уровень нагрузки на центральный процессор (ЦП) для каждого запроса, – но и многие другие показатели. Сколько запросов к базе данных требуется для обработки каждого заказа? Не пора ли приобрести сетевое оборудование с более высокой пропускной способностью? Сколько промахов кеша наблюдается? Достаточно ли много пользователей взаимодействует со сложной функцией, чтобы оправдать ее дальнейшее существование?

Это лишь некоторые из вопросов, на которые система мониторинга, основанная на метриках, может дать ответ и помочь вам понять, почему она дала именно такой ответ. Мы рассматриваем мониторинг как средство получения информации обо всей системе, от обзоров высокого уровня до перечисления мельчайших деталей, полезных для отладки. Полный набор средств мониторинга для отладки и анализа включает не только метрики, но также журналы, трассировки и результаты профилирования, но во главе угла стоят все-таки метрики, и они – первое, к чему вы должны обращаться, ощущив потребность в ответах на вопросы системного уровня.

Prometheus поощряет широкое внедрение своих инструментов в системы, от приложений до «голого железа». Они помогут вам увидеть, как взаимодействуют ваши подсистемы и компоненты, и преобразовать неизвестные в известные.

Эволюция Prometheus

Prometheus перешагнул 10-летний рубеж, и в этом втором издании мы постарались отразить все новые достижения. Prometheus продолжает развиваться и расширяться, с каждым годом предлагая еще больше возможностей для извлечения и хранения данных. Этот прогресс является результатом упорного труда сообщества пользователей и разработчиков, использующих Prometheus в самых разных отраслях для мониторинга самых разных систем и приложений.

Во втором издании этой книги рассказывается о многих новых функциях PromQL, средствах обнаружения служб и приемниках Alertmanager, добавленных со времени выхода первого издания книги.

В новой специальной главе рассматриваются функции обеспечения безопасности на стороне сервера, такие как TLS, добавленные в Prometheus и некоторые экспортеры.

Обозначения и соглашения, принятые в этой книге

В книге действуют следующие типографские соглашения.

Курсив

Используется для обозначения новых терминов, имен файлов и их расширений.

Моноширинный шрифт

Используется для оформления листингов программ, а также в обычном тексте для обозначения элементов программы, таких как имена переменных или функций, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

Моноширинный полужирный шрифт

Используется для выделения команд и другого текста, который должен быть набран самим пользователем.

Моноширинный курсив

Используется для выделения текста, который нужно заменить данными пользователя или значениями, определяемыми контекстом.



Так обозначаются советы.



Так обозначаются примечания общего характера.



Так обозначаются предупреждения и предостережения.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.dmk.ru на странице с описанием соответствующей книги.

Мы высоко ценим, хотя и не требуем ссылки на наши издания. В ссылке обычно указывается имя автора, название книги, издательство и ISBN, например: «Пивотто Ж., Бразил Б. Запускаем Prometheus. М.: ДМК Пресс, 2023. Copyright © 2023 O'Reilly Media, Inc., 978-1-098-13114-2 (англ.), 978-601-81034-1-4 (казах.)».

Если вы полагаете, что планируемое использование кода выходит за рамки изложенной выше лицензии, пожалуйста, обратитесь к нам по адресу dmk-press@gmail.com.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте www.dmkpress.com, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и O'Reilly очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Благодарности

Эта книга едва ли увидела бы свет без работы всей команды Prometheus и сотен участников сообщества. Мы хотели бы особо поблагодарить Джулиуса Фольца (Julius Volz), Ричарда Хартманна (Richard Hartmann), Карла Бергквиста (Carl Bergquist), Эндрю Макмиллана (Andrew McMillan) и Грега Старка (Greg Stark) за отзывы к рукописи первого издания этой книги. Спасибо также Брайану Бразилу (Brian Brazil), Бартломею Плотке (Bartłomiej Płotka), Карлу Бергквисту (Carl Bergquist), Т. Дж. Хоплоку (TJ Hoplock) и Ричарду Хартманну (Richard Hartmann) за отзывы ко второму изданию.

Об авторах

Жюльен Пивотто (Julien Pivotto) – ведущий разработчик сервера Prometheus и экосистемы CNCF с 2017 года. В настоящее время работает главным архитектором программного обеспечения в компании O11y и одновременно является ее соучредителем. Специализируется на поддержке различных инструментов мониторинга, в том числе Prometheus, Cortex, Loki и Jaeger. Обладая многолетним опытом, помогает организациям в развертывании и обслуживании этих инструментов, а также предоставляет индивидуальные решения для разработки.

Брайан Бразил (Brian Brazil) – основатель компании Robust Perception и разработчик ядра Prometheus. Занимается решением проблем с мониторингом для самых разных компаний, начиная от стартапов и заканчивая корпорациями из списка Fortune 500. Хорошо известен в сообществе Prometheus, провел бесчисленное количество презентаций на конференциях и освещает многие аспекты Prometheus и мониторинга в своем блоге на веб-сайте Robust Perception.

Колофон

На обложке «Запускаем Prometheus» изображен каменный орел (*Aquila rapax*) – хищная птица, обитающая в Африке, на Ближнем Востоке и в Индии. Имея длину 1,5–2 м и такой же размах крыльев, каменный орел немного меньше других представителей рода *Aquila*. Имеет коричневый окрас с желтоватым оттенком, наиболее преобладающим в верхней части тела, уступая место более темным перьям в хвосте.

Каменные орлы, как правило, строят свои гнезда на высоких деревьях, где моногамные пары откладывают от одного до трех яиц в год. Предпочитают засушливые области, такие как пустыни, степи и саванны, где питаются падалью, рептилиями и мелкими млекопитающими.

Считается, что из-за широкого ареала обитания каменные орлы не находятся под угрозой вымирания. Однако популяция каменного орла в Западной Африке сокращается из-за расширения обрабатываемых угодий, вторгающихся в их среду обитания.

Многие животные на обложках O'Reilly находятся под угрозой исчезновения; все они важны для мира.

Рисунок для обложки выполнила Карен Монтгомери, взяв за основу стаинную гравюру из энциклопедии «British Birds». Текст на обложке набран шрифтами Gilroy Semibold и Guardian Sans. Текст книги набран шрифтом Adobe Minion Pro; текст заголовков – шрифтом Adobe Myriad Condensed; а фрагменты программного кода – шрифтом Ubuntu Mono, созданным Daltonом Маагом (Dalton Maag).

Часть

ВВЕДЕНИЕ

Этот раздел познакомит вас с мониторингом в целом и системой Prometheus в частности.

В главе 1 вы узнаете о множестве различных вариантов мониторинга и подходов к нему; о метриках, которые использует Prometheus, и об архитектуре Prometheus.

В главе 2 вы запустите Prometheus в простейшей конфигурации и увидите, как система собирает машинные метрики, оценивает запросы и отправляет оповещения.

Глава 1

Что такое Prometheus?

Prometheus – это система мониторинга с открытым исходным кодом, основанная на метриках. Конечно, Prometheus далеко не единственная такая система, но что делает ее примечательной?

Prometheus выполняет только мониторинг – и делает это хорошо. Она имеет простую, но мощную модель данных и язык запросов, позволяющий анализировать производительность приложений и инфраструктуры. Она не пытается решать проблемы, выходящие за рамки метрик, оставляя решение этой задачи другим, более подходящим инструментам.

С той поры (2012 год), когда в SoundCloud работало всего несколько разработчиков, вокруг Prometheus выросло обширное сообщество. Ядро Prometheus написано на Go и распространяется под лицензией Apache 2.0. Однако в проект внесли свой вклад сотни людей, и он не контролируется какой-либо одной компанией. Всегда сложно сказать, сколько пользователей имеет проект с открытым исходным кодом, но, по нашим оценкам, в 2022 году Prometheus использовали сотни тысяч организаций. В 2016 году проект Prometheus стал вторым членом¹ Cloud Native Computing Foundation (CNCF).

Для инструментирования своего кода вы можете использовать клиентские библиотеки, написанные на всех популярных языках, включая Go, Java/JVM, C#/.Net, Python, Ruby, Node.js, Haskell, Erlang и Rust. Многие популярные приложения уже оснащены клиентскими библиотеками Prometheus, и среди них Kubernetes, Docker, Envoy и Vault. Для стороннего программного обеспечения, предоставляющего метрики в формате, отличном от Prometheus, доступны сотни библиотек интеграции. Они называются экспортёрами и обеспечивают поддержку таких продуктов, как HAProxy, MySQL, PostgreSQL, Redis, JMX, SNMP, Consul и Kafka. Друг Брайана даже добавил поддержку мониторинга своих серверов Minecraft, стремясь обеспечить приемлемую частоту обновления экрана.

Простой текстовый формат² упрощает экспортирование метрик Prometheus. Другие системы мониторинга, как с открытым исходным кодом, так

¹ Первым стал фреймворк Kubernetes.

² Наряду с простым текстовым форматом был реализован текстовый формат Prometheus – OpenMetrics, более стандартизированная версия, имеющая некоторые отличия от других форматов.

и коммерческие, добавили поддержку этого формата. Это позволяет всем таким системам мониторинга сосредоточиться на основных функциях, а не тратить время на дублирование поддержки каждой отдельной части программного обеспечения, которую пользователь может пожелать отслеживать.

Модель данных идентифицирует каждый временной ряд не только по имени, но и по неупорядоченному набору пар ключ–значение, называемых метками. Язык запросов PromQL допускает агрегирование по любой из этих меток, что позволяет анализировать данные мониторинга не только по отдельным процессам, но также по центрам обработки данных, службам и любым другим меткам, которые вы определите. Эти данные можно отобразить в виде графиков в системах поддержки дашбордов (информационных панелей), таких как Grafana (<https://oreil.ly/f5uMZ>) и Perses (<https://oreil.ly/YF-xW>).

Оповещения можно определять с использованием того же языка запросов PromQL, что используется для построения графиков. Если параметр можно изобразить на графике, то при его изменении можно послать оповещение. Метки упрощают обслуживание оповещений, позволяя создать одно предупреждение, охватывающее все возможные значения меток. В некоторых других системах мониторинга для этого придется создавать оповещения отдельно для каждой машины/приложения. Кроме того, механизм обнаружения служб может автоматически определять, какие приложения и машины необходимо извлечь из тех или иных источников, таких как Kubernetes, Consul, Amazon Elastic Compute Cloud (EC2), Azure, Google Compute Engine (GCE) и OpenStack.

При всех этих возможностях и преимуществах Prometheus остается простым в использовании. Сервер Prometheus способен обрабатывать миллионы выборок в секунду. Это один статически связанный двоичный файл с файлом конфигурации. Все компоненты Prometheus можно запускать в контейнерах, и они не делают ничего необычного, что могло бы помешать инструментам управления конфигурацией. Он реализован с учетом возможности интеграции в уже имеющуюся инфраструктуру как ее часть, а не как платформа управления.

Теперь, когда вы получили общее представление о Prometheus, вернемся немного назад и поговорим о том, что подразумевается под термином «мониторинг», чтобы заложить основу для дальнейшего обсуждения. После этого мы перечислим основные компоненты Prometheus и поговорим о том, чем Prometheus не является.

Что такое мониторинг?

В средней школе один из учителей сказал Брайану, что если спросить 10 экономистов, что означает слово «экономика», то вы получите 11 ответов. То же верно и для мониторинга. В настоящее время нет единого устоявшегося определения этого термина. Когда Брайан рассказывает другим, чем занимается, то люди думают, что его работа включает в себя все: от наблюдения за

температурой на фабриках до выявления сотрудников, общающихся в Facebook в рабочее время, и даже обнаружения злоумышленников в сетях.

Система мониторинга Prometheus не позволяет ничего из перечисленного. Она не предназначена для этого¹ и создавалась с целью помочь разработчикам программного обеспечения и администраторам в работе с промышленными производственными компьютерными системами, такими как приложения, инструменты, базы данных и сети, поддерживающие популярные веб-сайты.

Так что же такое мониторинг в нашем понимании? Давайте сузим этот вид оперативного мониторинга компьютерных систем до четырех вариантов использования.

Оповещение

Рассылка уведомлений, когда что-то идет не так, – как правило, самое важное, для чего предназначен мониторинг. Всегда желательно иметь возможность автоматически известить ответственного сотрудника о сложившейся ситуации.

Отладка

Получив уведомление, сотрудник должен оценить состояние системы, определить причину или причины и, наконец, решить проблему.

Анализ тенденций

Оповещения и отладка обычно характеризуются некоторой протяженностью во времени, от минуты до нескольких часов, и иногда желательно иметь возможность видеть, как системы используются и меняются с течением времени. Тенденции (или тренды) могут учитываться в проектных решениях и процессах, таких как планирование дальнейшего наращивания.

Решение практических задач

Когда у вас в руках оказывается молоток, то все предметы вокруг начинают казаться гвоздями. В конце концов, все системы мониторинга организованы как конвейеры обработки данных. Иногда удобнее выделить часть системы мониторинга для решения некоторой практической задачи, не связанной с мониторингом, чем создавать отдельное решение. Это не совсем мониторинг, но на практике такой вариант применения встречается довольно часто, поэтому мы включили его в список.

В зависимости от специализации и опыта человека, с которым вы разговариваете, он может счесть, что только эти варианты составляют мониторинг. Такое недопонимание приводит к многочисленным дискуссиям на тему мониторинга, которые могут заканчиваться еще большим недопониманием. Чтобы помочь вам лучше понять суть, рассмотрим небольшую часть истории мониторинга.

¹ Мониторинг температуры внутри машин и центров обработки данных на самом деле не редкость. Более того, есть даже пользователи, которые ради забавы используют Prometheus для мониторинга погоды.

Краткая и неполная история мониторинга

За последние несколько лет в мониторинге произошел сдвиг в сторону использования специализированных инструментов, включая Prometheus. Долгое время доминирующим решением была некая комбинация Nagios и Graphite или их вариантов.

Когда мы говорим Nagios, то подразумеваем большое семейство программного обеспечения, такое как Icinga, Zmon и Sensu. Работа такого ПО основана на регулярном выполнении сценариев, называемых *проверками*. Если проверка завершается неудачно, возвращая ненулевой код завершения, то генерируется предупреждение. Первая версия Nagios была создана Итаном Галстадом (Ethan Galstad) в 1996 году как приложение для MS-DOS. Затем в 1999 году оно было выпущено под названием NetSaint и в 2002-м переименовано в Nagios.

История Graphite началась в далеком 1994 году, когда Тобиас Этикер (Tobias Oetiker) написал сценарий на Perl, который в 1995 году превратился в Multi Router Traffic Grapher, или MRTG 1.0. Как следует из названия, он в основном использовался для мониторинга сети через простой протокол управления сетью (Simple Network Management Protocol, SNMP). MRTG также мог получать метрики, запуская сторонние сценарии¹. В 1997 году произошли существенные изменения, связанные с переносом части кода на C и созданием базы данных Round Robin Database (RRD), использовавшейся для хранения метрик. Это привело к заметному улучшению производительности, а RRD стала основой для некоторых других инструментов, включая Smokeping и Graphite.

В проекте Graphite, основанном в 2006 году, для хранения метрик стала использоваться база данных Whisper с архитектурой, похожей на RRD. Приложение Graphite не собирает данные само, а отправляет их с помощью специализированных инструментов, таких как collectd и StatsD, созданных в 2005 и 2010 годах соответственно.

Отсюда можно сделать вывод, что когда-то построение графиков и оповещение были совершенно разными задачами, выполняемыми разными инструментами. Вы можете написать сценарий для оценки запроса в Graphite и генерировать оповещения на этой основе, но большинство проверок, как правило, оказывались в неожиданных состояниях, например, когда процесс не запущен.

Еще один пережиток той эпохи – ручной подход к администрированию компьютерных служб. Службы развертывались на отдельных машинах, и их поддержкой заботливо занимались системные администраторы. Преданные своему делу инженеры оперативно реагировали на предупреждения, которые могли указывать на проблему. С ростом популярности облачных технологий, таких как EC2, Docker и Kubernetes, инженеры пришли к осознанию, что обращение с отдельными машинами и службами как с домашними питомцами, когда каждому уделяется отдельное внимание, плохо масштаб-

¹ У Брайана остались приятные воспоминания о настройке MRTG в начале 2000-х, о написании сценариев для отчетов о температуре и использовании сети на моих домашних компьютерах.

бирается. Вместо этого предпочтительнее относиться к ним как к крупному огненному скоту и управлять ими как группами. Точно так же, как индустрия перешла от ручного управления к использованию таких инструментов, как Chef и Ansible, и постепенно начала использовать такие технологии, как Kubernetes, в мониторинге тоже появилась необходимость совершить подобный переход. Это означает переход от проверок отдельных процессов на отдельных машинах к мониторингу работоспособности службы в целом.

Позднее из двух проектов с открытым исходным кодом, OpenCensus и OpenTracing, появился проект OpenTelemetry. OTEL¹ – это спецификация и набор компонентов для организации встроенной телеметрии в проектах. В отношении метрик он совместим с Prometheus при использовании дополнительного сборщика OpenTelemetry², отвечающего за сбор и передачу метрик серверу Prometheus.

Вы могли заметить, что мы не упомянули журналирование, трассировку и профилирование. Исторически журналы использовались как ручной инструмент, используемый вместе с `tail`, `grep` и `awk`. Возможно, вам приходилось встречаться с инструментом анализа, таким как AWStats, позволяющим создавать почасовые или посutoчные отчеты. В последние годы журналы также использовались в качестве важного компонента мониторинга, например в стеке Elasticsearch, Logstash и Kibana (ELK) и OpenSearch. Трассировка и профилирование обычно выполняются с помощью собственного программного стека; например, для трассировки широко используются Zipkin и Jaeger, а для непрерывного профилирования – Parca и Pyroscope.

Теперь, вкратце познакомившись с графиками и оповещениями, давайте посмотрим, как метрики и журналы вписываются в общий ландшафт. Есть ли еще категории мониторинга, кроме этих двух?

Категории мониторинга

Большая часть мониторинга сводится к одному и тому же: событиям. События могут быть практически любыми, в том числе это:

- получение HTTP-запроса;
- отправка HTTP-ответа 400;
- вход в функцию;
- достижение ветки `else` в операторе `if`;
- выход из функции;
- вход пользователя;
- запись данных на диск;
- чтение данных из сети;
- запрос дополнительной памяти у ядра.

¹ OTEL – неофициальное название OpenTelemetry.

² В то время, когда мы работали над этой книгой, на саммите разработчиков Prometheus было решено добавить в сервер Prometheus поддержку протокола OTEL в будущем, правда, не были установлены четкие временные рамки воплощения этого решения в жизнь.

Все события имеют некоторый контекст. HTTP-запрос будет иметь исходящий и входящий IP-адреса, URL назначения, cookie и идентификатор пользователя, отправившего запрос. В HTTP-ответе будет указана продолжительность обработки запроса, код состояния HTTP и длина тела ответа. События, связанные с функциями, имеют стек вызовов функций, по которому можно отследить путь, приведший к вызову данной функции, и все, что связано с этим вызовом, например HTTP-запрос.

Наличие контекста для всех событий может пригодиться для отладки и изучения особенностей работы системы с технической и с коммерческой точек зрения, но такой объем данных нецелесообразно обрабатывать и хранить. Поэтому на практике используются четыре основных подхода к сокращению этого объема данных до более приемлемого уровня, а именно профилирование, трассировка, журналирование и получение метрик.

Профилирование

В профилировании используется подход, согласно которому для событий сохраняется не весь контекст, а только ограниченная его часть, характеризующая некоторый небольшой интервал времени.

Примером инструмента профилирования может служить Tcpdump. Он фиксирует сетевой трафик на основе заданного фильтра. Это важный инструмент отладки, но его нельзя запустить в постоянную работу, потому что он быстро исчерпает место на диске.

Другой пример – отладочные сборки двоичных файлов, фиксирующие данные профилирования. Они предоставляют множество полезной информации, но процедуры сбора всей этой информации, такой как фиксация времени вызова каждой функции, сильно влияют на производительность, а это означает, что профилирование редко бывает целесообразно запускать в промышленном окружении на постоянной основе.

Механизм расширенных фильтров пакетов Беркли (enhanced Berkeley Packet Filters, eBPF) в ядре Linux позволяет детально исследовать события, возникающие в ядре, от операций с файловой системой до сетевых действий. Этот механизм обеспечивает такой уровень проникновения вглубь происходящего, какой ранее был недоступен. eBPF обладает и другими преимуществами, такими как переносимость и безопасность. Мы настоятельно рекомендуем прочитать работы Брэндана Грегга (Brendan Gregg; <https://oreil.ly/n15mM>) на эту тему.

Профилирование, по сути, является тактическим приемом отладки. Для осуществления профилирования на протяжении долгого времени необходимо предусмотреть сокращение объема данных или перейти на использование методов *непрерывного профилирования*, позволяющих осуществлять сбор и хранение накапливаемых данных.

Главное отличие непрерывного профилирования – снижение частоты сбора метрик для сокращения объема данных и уменьшения накладных расходов. Один из новых инструментов непрерывного профилирования, Parca Agent на основе eBPF (<https://parca.dev/>), использует частоту 19 Гц¹. Как следствие, для

¹ Сравните с частотой 100 Гц в среде выполнения Go или даже 10 000 Гц в Chromium.

накопления статистически значимых объемов данных требуются минуты, а не секунды, но при этом он способен собрать данные, необходимые для понимания, какие участки кода потребляют большую часть процессорного времени, и помочь повысить оптимизировать именно эти участки кода.

Трассировка

Обычно при выполнении трассировки рассматриваются не все события, а только некоторая их доля, например одно из ста, проходящих через исследуемые функции. В ходе трассировки отмечаются функции, находящиеся в стеке вызовов, а также часто время выполнения каждой из этих функций. Из этой информации можно получить представление о том, где программа тратит больше всего времени и какие пути выполнения в коде вызывают наибольшие задержки.

Вместо создания моментальных снимков трассировки стека в интересующих точках некоторые системы трассировки определяют и записывают время работы каждой функции, находящейся в стеке ниже интересующей функции. Например, система может выбирать один из сотни HTTP-запросов и для каждого выбранного запроса фиксировать время, затраченное на взаимодействие с серверными программными компонентами, такими как базы данных и кеша. Эта информация позволяет увидеть, как различаются времена в зависимости от таких факторов, как попадание в кеш и промахи кеша.

Распределенная трассировка делает еще один шаг вперед, позволяя выполнять трассировку между процессами, присваивая уникальные идентификаторы запросам, которые передаются между процессами при вызове удаленных процедур (Remote Procedure Call, RPC), и определяя, является ли этот запрос тем, который следует отслеживать. Затем трассировки различных процессов и машин могут объединяться по идентификатору запроса. Это очень важный инструмент для отладки архитектур распределенных микросервисов. В качестве примеров технологий в этой области можно назвать OpenZipkin и Jaeger.

Выполняя трассировку, важно иметь в виду, что частота выборки и объемы данных могут оказывать существенное влияние на производительность инструментов.

Журналирование

Механизм журналирования получает ограниченный набор событий и фиксирует некоторый контекст для каждого из них. Например, он может получать все входящие HTTP-запросы или все исходящие запросы к базе данных. Чтобы не потреблять слишком много ресурсов, каждая запись в журнале обычно ограничивается примерно сотней полей. Помимо этого, необходимо учитывать пропускную способность и объем хранилища.

Например, для сервера, обрабатывающего 1000 запросов в секунду, когда на каждый запрос создается запись со 100 полями по 10 байт каждый, размер журнала будет увеличиваться на 1 Мбайт в секунду. Это потребует полосы

пропускания сети 100 Мбит и 84 Гбайт пространства в хранилище в день только для журналирования.

Большое преимущество журналирования – отсутствие (обычно) затрат ресурсов на выборку событий, поэтому даже при ограниченном количестве полей журналирование помогает определить влияние медленных запросов на одного конкретного пользователя, обращающегося к одной конкретной конечной точке API.

Подобно мониторингу, журналирование тоже по-разному понимается разными людьми, что может вызвать путаницу. Различные виды журналов имеют разные области применения, долговечность и требования к хранилищу. Всего можно выделить четыре общие категории.

Журналы транзакций

Важные записи, которые должны храниться в безопасности, и, возможно, вечно. К этой категории относится все, что касается денег или важных функций, ориентированных на пользователя.

Журналы запросов

Информация о каждом HTTP-запросе или каждом вызове базы данных помещается в журнал запросов. Информация из этих журналов может использоваться для реализации функций, ориентированных на пользователя, или только для внутренней оптимизации. Обычно нежелательно терять эти журналы, но и потеря некоторых из них не означает конца света.

Журналы приложений

Не вся журналируемая информация относится к запросам; какая-то ее часть характеризует работу самого процесса. Типичными ее представителями являются сообщения, выводимые приложением при запуске, фоновыми задачами и другими механизмами на уровне процесса. Эти журналы часто читаются непосредственно человеком, поэтому старайтесь избегать создания более чем нескольких записей в минуту в обычном режиме работы.

Журналы отладки

Журналы отладки, как правило, очень подробные, поэтому для их поддержки и хранения требуются большие затраты. Обычно они используются только в очень ограниченных по времени периодах отладки и по своим характеристикам близки к профилированию из-за большого объема данных. Требования к надежности и хранению таких журналов, как правило, низкие, и они могут даже не покидать машину, на которой созданы.

Однаковая интерпретация различных типов журналов может поставить вас в неприятное положение, когда большой объем данных в журналах отладки сочетается с экстремальными требованиями к надежности журналов транзакций. Поэтому обязательно постарайтесь спланировать отделение журналов отладки, чтобы их можно было обрабатывать отдельно.

Примерами систем журналирования могут служить стек ELK, OpenSearch, Grafana Loki и Graylog.

Метрики

Метрики редко бывают связаны с контекстом и часто просто представляют совокупность различных типов событий во времени. Чтобы обеспечить потребление ресурсов в разумных пределах, количество отслеживаемых метрик должно быть ограничено: 10 000 на процесс – разумная верхняя граница, которую вы должны иметь в виду.

Примерами метрик могут служить: количество полученных HTTP-запросов; время, потраченное на обработку запросов; и количество запросов, обрабатываемых в настоящий момент. Исключая любую информацию о контексте, можно ограничить в разумных пределах объем данных и время на их обработку.

Однако это не означает, что контекст всегда игнорируется. В отношении HTTP-запросов, например, можно создать отдельную метрику для каждого URL. Но не забывайте о рекомендации в 10 000 метрик, поскольку каждый отдельный URL теперь считается метрикой. Использовать такой контекст, как адрес электронной почты пользователя, было бы неразумно, потому что такие сведения имеют неограниченную кардинальность¹.

Вы можете использовать метрики для определения задержек и объемов данных, обрабатываемых каждой подсистемой в ваших приложениях, чтобы упростить определение причин замедления. Записи в журналах не могут иметь столько полей, но, узнав, какая подсистема виновна в замедлении приложения, вы сможете задействовать журналирование, чтобы выяснить, какие именно запросы пользователей вносят наибольший вклад в задержку.

Именно здесь компромисс между журналами и метриками становится наиболее очевидным. Метрики позволяют собирать информацию о событиях со всего процесса, но, как правило, не более чем с одним или двумя контекстными полями с ограниченной кардинальностью. Журналы позволяют собирать информацию обо всех событиях одного типа, но могут фиксировать только сотню полей контекста с неограниченной кардинальностью. Важно понимать, что такое кардинальность и какие ограничения она накладывает на метрики, о чём мы и поговорим в последующих главах.

Prometheus, как система мониторинга на основе метрик, предназначена для отслеживания общего состояния системы, ее поведения и производительности, а не отдельных событий. Другими словами, Prometheus может сообщить, что за последнюю минуту было получено 15 запросов, на обработку которых ушло 4 с, что привело к 40 обращениям к базе данных, 17 попаданиям в кеш и двум покупкам клиентов. Стоимость и пути выполнения в коде для обработки отдельных вызовов – это сфера профилирования или журналирования.

Теперь, получив представление о том, какое место Prometheus занимает в общем пространстве мониторинга, давайте рассмотрим различные компоненты этой системы.

¹ Адреса электронной почты также часто относятся к категории персональной информации, что влечет за собой проблему соблюдения требований и конфиденциальности, которой лучше избегать при мониторинге.

Архитектура Prometheus

На рис. 1.1 показана общая архитектура Prometheus. Prometheus обнаруживает цели для извлечения информации, используя механизм обнаружения служб. Этими целями могут быть ваши собственные инструментированные приложения или сторонние приложения, сведения о которых вы можете получать с помощью экспортёров. Собранные данные сохраняются и могут отображаться на дашбордах с помощью PromQL или использоваться для рассылки оповещений диспетчером уведомлений Alertmanager, который преобразует данные в страницы, электронные письма и другие формы уведомлений.

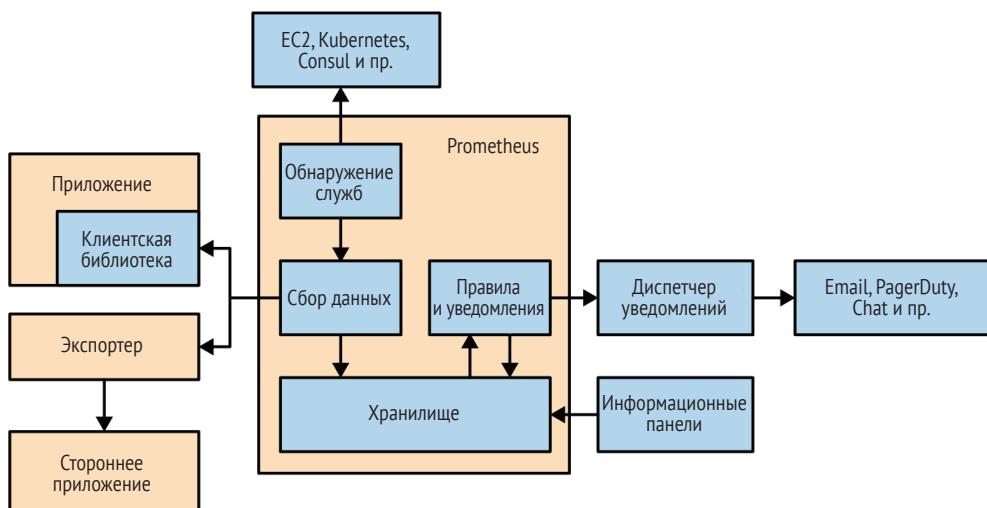


Рис. 1.1 ♦ Архитектура Prometheus

Клиентские библиотеки

Метрики обычно не возникают в приложениях как по волшебству; кто-то должен добавить инструменты для их создания. И для этой цели используются клиентские библиотеки. Добавив всего пару строк кода, вы можете определить метрику и внедрить в приложение нужные инструменты. Это называется *прямым инструментированием*.

Клиентские библиотеки доступны для всех основных языков программирования и операционных систем. Проект Prometheus предоставляет официальные клиентские библиотеки на Go, Python, Java/JVM, Ruby и Rust. Существует также множество сторонних клиентских библиотек, например для C#/.Net, Node.js, Haskell и Erlang.

Официальные и неофициальные библиотеки

Не надо бояться использовать неофициальные или сторонние клиентские библиотеки. Команда проекта Prometheus просто не в состоянии реализовать и поддерживать различные библиотеки интеграции с сотнями приложений и систем, с которыми вам, возможно, придется иметь дело. Поэтому подавляющее большинство таких библиотек в экосистеме являются сторонними. А чтобы обеспечить согласованность, доступны рекомендации по написанию этих библиотек.

Клиентские библиотеки берут на себя заботу обо всех деталях, таких как безопасность потоков, выполнение рутинных задач и создание сообщений в формате Prometheus и/или OpenMetrics в ответ на HTTP-запросы. Как уже отмечалось, мониторинг на основе метрик не связан с отслеживанием отдельных событий, поэтому потребление памяти клиентской библиотекой не увеличивается с увеличением количества событий. Скорее, объем потребляемой памяти зависит от количества имеющихся метрик.

Если одна из библиотек, включенных в ваше приложение, уже инструментирована средствами Prometheus, то они будут задействоваться автоматически. То есть, инструментировав ключевую библиотеку, такую как RPC-клиент, вы сможете получать использовать это преимущество во всех своих приложениях.

Некоторые метрики, такие как потребление процессора и статистика сборки мусора, обычно предоставляются клиентскими библиотеками по умолчанию.

Клиентские библиотеки могут отправлять метрики не только в текстовых форматах Prometheus и OpenMetrics. Prometheus – это открытая экосистема, и те же самые функции, что используются для создания метрик в текстовых форматах, могут применяться для создания метрик в других форматах или для передачи данных другим системам инструментирования. Точно так же есть возможность получать метрики из других систем инструментирования и передавать их клиентской библиотеке Prometheus, если вы еще не заменили их средствами Prometheus.

Экспортеры

Не всякий код, который вы запускаете, доступен вам для изменения, и прямое его инструментирование просто невозможно. Например, едва ли можно ожидать, что ядро операционной системы начнет выводить метрики в формате Prometheus через HTTP.

Однако такое ПО часто имеет некоторый интерфейс, через который можно получить доступ к метрикам. Это может быть специальный формат, требующий специальной обработки, что характерно для многих метрик Linux, или хорошо зарекомендовавший себя стандарт, такой как SNMP.

Экспортер (exporter) – это программный компонент, развертываемый рядом с приложением, из которого требуется получить метрики. Он при-

нимает запросы от Prometheus, извлекает необходимые данные из приложения, преобразует их в правильный формат и возвращает их в Prometheus. Экспортер можно представить себе как маленький прокси-сервер, преобразующий данные из формата метрик приложения в формат представления Prometheus.

В отличие от прямого инструментирования, которое можно применить к своему коду, экспортёры используют другой способ инструментирования, известный как *настраиваемые сборщики* (custom collectors), или *ConstMetrics*¹.

Самое замечательное, что, учитывая многочисленность сообщества Prometheus, нужный вам экспортёр почти наверняка уже существует и может использоваться без особых усилий с вашей стороны. Если у экспортёра отсутствует интересующая вас метрика, то вы всегда сможете послать разработчикам запрос на добавление новой особенности и тем самым поспособствовать совершенствованию экспортёра.

Обнаружение служб

После инструментирования приложения настройки экспортёров Prometheus должен узнать, где они находятся, чтобы знать, что отслеживать и вовремя определять сбои. В динамических средах нельзя просто составить список приложений и экспортёров один раз, так как он быстро устареет. И в таких случаях вам поможет механизм обнаружения служб.

Вероятно, у вас уже есть база данных с информацией о ваших машинах, приложениях и том, что они делают. Эта информация может храниться в базе данных Chef, в файле инвентаризации Ansible, в тегах ваших экземпляров EC2, в метках и аннотациях Kubernetes или, может быть, просто в вашей вики-документации.

Prometheus способна интегрироваться со многими распространёнными механизмами обнаружения сервисов, такими как Kubernetes, EC2 и Consul. Существует также универсальная интеграция для тех, чьи окружения немного отличаются от общепринятых (см. разделы «Файл» и «HTTP» в главе 8).

Тем не менее остается еще одна проблема. Тот факт, что Prometheus имеет список машин и служб, не означает, что мы знаем, как они вписываются в вашу архитектуру. Например, вы можете использовать тег `Name` в EC2², чтобы указать, какое приложение работает на компьютере, тогда как другие могут использовать тег `app`.

Поскольку каждая организация делает это немного по-своему, Prometheus позволяет настроить сопоставление метаданных из службы обнаружения с целями мониторинга и их метками, предлагая возможность изменения меток.

¹ ConstMetric – это разговорный термин и происходит от имени функции `MustNewConstMetric` в клиентской библиотеке Go, используемой для создания метрик экспортёрами, написанными на Go.

² Тег `Name` в EC2 – это имя экземпляра EC2, отображаемое в веб-консоли EC2.

Извлечение данных

Обнаружение служб и поддержка изменения меток позволяют получить список целей для мониторинга. Имея такой список, Prometheus должен как-то получить метрики. Для этого она отправляет HTTP-запрос на *извлечение*, анализирует ответ и сохраняет полученные метрики в хранилище. Дополнительно Prometheus добавляет несколько полезных метрик, например признак удачного получения данных и сколько времени это заняло. Операции извлечения выполняются регулярно, обычно каждые 10-60 с для каждой цели.

Получение по запросу или пассивный прием

Prometheus – это система, основанная на получении метрик по запросу (pull). Она сама решает, когда и что извлекать, основываясь на своей конфигурации. Существуют также системы, предполагающие пассивный прием (push), когда цели мониторинга сами решают, когда и как часто они будут передавать свои метрики.

В интернете ведутся бурные споры об этих двух подходах, которые во многом напоминают дебаты – что лучше, Vim или EMACS. Достаточно сказать, что оба подхода имеют свои плюсы и минусы, и в целом выбор того или иного не имеет большого значения.

Как пользователь Prometheus, вы должны понимать, что получение по запросу укоренилось в ядре Prometheus, и пытаться заставить эту систему принимать данные, посылаемые по инициативе целей мониторинга, в лучшем случае неразумно.

Хранилище

Prometheus хранит данные локально в собственной базе данных. Распределенные системы сложно сделать надежными, поэтому Prometheus не пытается использовать какую-либо кластеризацию. Кроме надежности этот аспект также упрощает запуск Prometheus.

За прошедшие годы технологии хранения претерпели ряд изменений, и система хранения в Prometheus 2.0 стала третьей итерацией. Она может обрабатывать миллионы выборок в секунду, что позволяет контролировать тысячи машин с помощью одного сервера Prometheus. Используемый алгоритм сжатия может достигать 1.3 байта на выборку для реальных данных. В качестве основы рекомендуется использовать твердотельные накопители, такие как SSD, но это не является обязательным условием.

Дашборды

Prometheus предлагает ряд HTTP API, позволяющих получать исходные данные и обрабатывать запросы PromQL. Их можно использовать для создания

графиков и дашбордов. По умолчанию Prometheus предоставляет *браузер выражений* (expression browser). Он использует эти API и подходит для обработки специальных запросов и исследования данных, но это не типичная система дашбордов.

Для создания дашбордов лучше использовать Grafana. Она имеет широкий спектр функций, включая официальную поддержку Prometheus как источника данных, может создавать самые разнообразные дашборды, такие как на рис. 1.2. Grafana поддерживает возможность взаимодействия с несколькими серверами Prometheus даже в рамках одного дашборда.



Рис. 1.2 ♦ Дашборд Grafana

Правила записи и оповещения

PromQL и поддержка хранилища – мощные и эффективные механизмы, но оперативное агрегирование метрик с тысяч машин в процессе визуализации графиков может немного отставать. Правила записи позволяют вычислять выражения PromQL через заданные интервалы времени и загружать их результаты в механизм хранения.

Правила оповещения – еще одна форма правил записи. Они также регулярно вычисляют выражения PromQL, и любые результаты этих выражений становятся уведомлениями. Уведомления отправляются в диспетчер уведомлений Alertmanager.

Управление уведомлениями

Диспетчер уведомлений Alertmanager получает сообщения от серверов Prometheus и превращает их в уведомления. Уведомления могут включать адрес электронной почты, приложения чата, такие как Slack, и службы, такие как PagerDuty.

Alertmanager не просто слепо превращает сообщения в уведомления по принципу «один к одному». Связанные уведомления могут объединяться в одно уведомление для предотвращения шторма уведомлений¹, и для каждой из команд сотрудников можно настроить разные маршруты рассылки уведомлений. Также есть возможность отключить уведомления, например чтобы отложить проблему, решение которой запланировано в ближайший период технического обслуживания.

На отправке уведомлений роль Alertmanager заканчивается; для управления реакцией людей на инциденты следует использовать такие службы, как PagerDuty и системы регистрации инцидентов.

- ✓ Уведомления и соответствующие пороговые значения настраиваются в Prometheus, а не в Alertmanager.

Долгосрочное хранение

Поскольку Prometheus хранит данные только на локальной машине, вы ограничены объемом дискового пространства на этой машине². Обычно интерес представляют только самые свежие данные, полученные, например, за последние сутки или чуть больше, но для перспективного планирования развития желателен более длительный срок хранения.

Prometheus не предлагает кластерных решений для хранения данных на нескольких машинах, но имеет API для чтения и записи в удаленные хранилища, что позволяет перекладывать роль хранилищ на другие системы и прозрачно выполнять запросы PromQL как к локальным, так и к удаленным данным.

Чем Prometheus не является

Теперь, получив представление о том, какое место Prometheus занимает в более широком ландшафте мониторинга и его основных компонентах,

¹ Уведомление сообщает дежурному инженеру о необходимости незамедлительно изучить проблему и устраниить ее. Уведомления могут рассыпаться на традиционные радиопейджеры, но в наши дни, скорее всего, рассылка будет осуществляться на мобильные телефоны в виде SMS, push-уведомлений или телефонных звонков. Шторм уведомлений возникает, когда рассыпается большое количество уведомлений за короткий промежуток времени.

² Современные машины могут хранить огромные объемы данных локально, поэтому вам может не понадобиться отдельная кластерная система хранения.

рассмотрим некоторые случаи, когда Prometheus оказывается не лучшим выбором.

Как система, основанная на метриках, Prometheus не подходит для хранения журналов событий или отдельных событий. Она также не лучший выбор для данных с высокой кардинальностью, таких как адреса электронной почты или имена пользователей.

Prometheus предназначена для оперативного мониторинга, когда небольшие неточности и состояния гонки из-за таких факторов, как планирование выполнения процессов в ядре и неудачные попытки сбора информации, являются обычным делом. Prometheus идет на компромисс и предпочитает предоставлять данные, которые на 99.9 % верны, вместо остановки мониторинга в ожидании идеальных данных. Таким образом, в приложениях, связанных с финансами или выставлением счетов, Prometheus следует использовать с большой осторожностью.

В следующей главе мы покажем вам, как запустить Prometheus и настроить простой мониторинг.

Глава 2

Настройка и запуск Prometheus

В этой главе вы настроите и запустите Prometheus, Node Exporter и Alertmanager. В этом простом примере мы реализуем мониторинг одной машины, и вы получите некоторое представление о том, как выглядит процесс развертывания Prometheus. В следующих главах мы подробно рассмотрим каждый аспект настройки.

Для опробования примеров из этой главы вам потребуется машина с более или менее свежей версией Linux. Подойдет как физическая, так и виртуальная машина. Вы будете использовать командную строку и получать доступ к службам на машине с помощью веб-браузера. Для простоты предположим, что все работы выполняются на локальном хосте, но если это не так, то просто измените URL-адреса соответствующим образом.



Базовая конфигурация, аналогичная используемой в этой главе, доступна на демонстрационном сайте Prometheus (<https://oreil.ly/KHxZC>).

Запуск Prometheus

Готовые версии Prometheus и других компонентов доступны на сайте проекта Prometheus (https://oreil.ly/e_S6d). Перейдите на этот сайт и загрузите последнюю версию Prometheus для ОС Linux с аппаратной архитектурой amd64; страница загрузки будет выглядеть примерно так, как показано на рис. 2.1.

В книге мы используем Prometheus 2.37.0, поэтому соответствующий файл дистрибутива имеет имя *prometheus-2.37.0.linux-amd64.tar.gz*.

prometheus				
The Prometheus monitoring system and time series database. prometheus/prometheus				
2.37.0 / 2022-07-14 <small>LTS</small> Release notes				
File name	OS	Arch	Size	SHA256 Checksum
prometheus-2.37.0.darwin-amd64.tar.gz	darwin	amd64	80.01 MiB	ba760de15e016680d2952e103aeef2f2164c1b7bdc8dd76214e099f3cbf4045416
prometheus-2.37.0.linux-amd64.tar.gz	linux	amd64	79.90 MiB	ca70f5a261fd545da0b9893c42a73547a94ebd5111ef2b6b9f8742c5dbb73522
prometheus-2.37.0.windows-amd64.zip	windows	amd64	81.47 MiB	99a2f58cc66d7a6cf06db865d59daf291b1055e6c4485b97e8704db7842a4b61f

Рис. 2.1 ♦ Фрагмент страницы загрузки Prometheus:
версия для Linux/amd64 доступна по средней ссылке

Долгосрочная поддержка

Младшие версии Prometheus выпускаются с периодичностью раз в шесть недель. Обновлять установленную версию с такой частотой может оказаться сложной задачей, поэтому некоторые версии определяются как *версии с долгосрочной поддержкой* (Long Term Support, LTS). Версии LTS поддерживаются в течение более длительного периода времени, чем обычные версии: вместо шести недель выпуски LTS обновляются исправлениями ошибок и безопасности в течение одного года. Полное описание обновлений можно найти на сайте Prometheus (<https://oreil.ly/ZxU-S>).



Обновления Prometheus предназначены для обеспечения безопасности между младшими версиями, такими как 2.0.0 до 2.0.1, 2.1.0 или 2.3.1. Однако, как и в случае с любым другим программным обеспечением, мы рекомендуем читать журнал изменений.

Для этой главы подойдет любая версия Prometheus 2.xx.

Распакуйте архив в командной строке и перейдите в распакованный каталог¹:

```
hostname $ tar -xzf prometheus-*.linux-amd64.tar.gz
hostname $ cd prometheus-*.linux-amd64/
```

Теперь измените содержимое файла *prometheus.yml*, как показано ниже:

```
global:
  scrape_interval: 10s
scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets:
          - localhost:9090
```

¹ В командах ниже используются шаблонные символы на месте версии на тот случай, если вы используете версию, отличную от нашей. Символ звездочки соответствует любому тексту.

! В экосистеме Prometheus файлы конфигурации оформляются в формате YAML (YAML Ain't Markup Language – YAML не язык разметки), так как текст в этом формате легко читается людьми и может обрабатываться программными инструментами. Однако этот формат чувствителен к пробелам, поэтому копируйте примеры с максимальной точностью и используйте пробелы, а не табуляции¹.

По умолчанию Prometheus использует TCP-порт 9090, а конфигурация выше предписывает извлекать данные каждые 10 с. Теперь вы можете запустить двоичный файл Prometheus, выполнив команду `./prometheus`:

```
hostname $ ./prometheus
level=info ... msg="No time or size retention was set so using the default
time retention" duration=15d
level=info ... msg="Starting Prometheus" version="(version=2.37.0, branch=HEAD,
revision=b41e0750abf5cc18d8233161560731de05199330)"
level=info ... build_context="(go=go1.18.4, user=root@0ebb6827e27f,
date=20220714-15:13:18)"
level=info ... host_details="(Linux 5.18.12 #1-NixOS SMP PREEMPT..."
level=info ... fd_limits="(soft=1024, hard=1048576)"
level=info ... msg="Start listening for connections" address=0.0.0.0:9090
level=info ... msg="Starting TSDB ..."
level=info ... msg="TSDB started"
level=info ... component=web msg="TLS is disabled." http2=false
level=info ... msg="Loading configuration file" filename=prometheus.yml
level=info ... msg="Server is ready to receive web requests."
```

Как видите, в процессе запуска Prometheus сообщает различную полезную информацию, включая точную версию и сведения о машине, на которой он запущен. Теперь можно получить доступ к пользовательскому интерфейсу Prometheus в браузере по адресу `http://localhost:9090/`. Он выглядит, как показано на рис. 2.2.

Это *браузер выражений* (expression browser), в котором можно запускать запросы PromQL. В пользовательском интерфейсе также есть еще несколько страниц, которые помогут вам понять, что делает Prometheus, например страница **Targets** (Цели) на вкладке **Status** (Состояние), которая выглядит, как показано на рис. 2.3.

В списке на этой странице присутствует только один сервер Prometheus, находящийся в состоянии UP (работает), которое означает, что последняя попытка извлечения информации прошла успешно. Если бы возникла какая-то проблема при извлечении метрик, то в поле **Error** (Ошибка) появилось бы сообщение.

Еще одна страница, на которую следует обратить внимание, – страница `/metrics`. Тот факт, что сама система Prometheus инструментирована и подготовлена для сбора метрик, не должен вызывать удивления. Метрики доступны по адресу `http://localhost:9090/metrics` в удобочитаемом виде, как показано на рис. 2.4.

¹ Возможно, вам интересно, почему Prometheus не использует формат JSON. Дело в том, что формату JSON присущи свои проблемы, такие как строгость к запятым, и, в отличие от YAML, он не поддерживает комментарии. Поскольку JSON является подмножеством YAML, вы можете использовать JSON, если хотите.

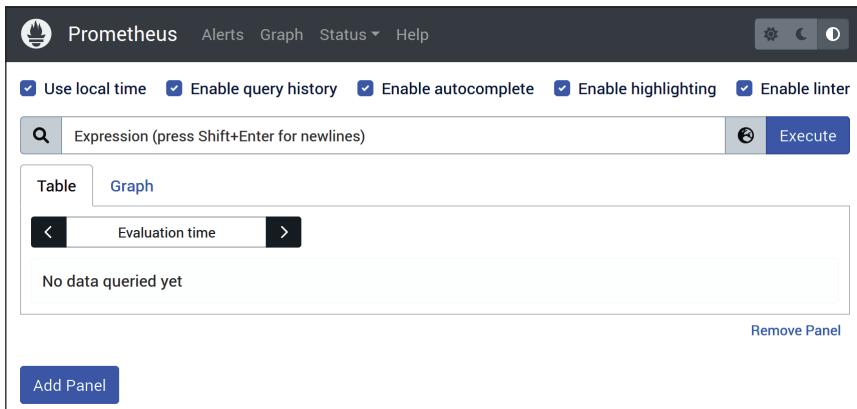


Рис. 2.2 ♦ Браузер выражений в Prometheus

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9090/metrics	UP	instance=localhost:9090 job=prometheus	4.572s ago	5.789ms	

Рис. 2.3 ♦ Страница состояния целей

```
# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 2.8243e-05
go_gc_duration_seconds{quantile="0.25"} 4.2601e-05
go_gc_duration_seconds{quantile="0.5"} 4.3442e-05
go_gc_duration_seconds{quantile="0.75"} 0.000197572
go_gc_duration_seconds{quantile="1"} 0.000208714
go_gc_duration_seconds_sum 0.000520572
go_gc_duration_seconds_count 5
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 27
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.18.4"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 1.3531248e+07
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 2.0167168e+07
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.459769e+06
```

Рис. 2.4 ♦ Первая часть страницы /metrics

Обратите внимание, что имеются метрики не только в коде самой Prometheus, но также в среде выполнения Go и в процессе.

Использование браузера выражений

Браузер выражений удобно использовать для выполнения специальных запросов, разработки выражений PromQL и отладки как PromQL, так и данных внутри Prometheus.

Для начала убедитесь, что вы находитесь в представлении консоли, введите выражение `up` и щелкните по кнопке **Execute** (Выполнить).

Как показано на рис. 2.5, этот запрос возвращает единственный результат со значением 1 и именем `up{instance="localhost:9090",job="prometheus"}`. `up` – это специальная метрика, добавляемая Prometheus перед началом сбора данных; 1 означает, что сбор прошел успешно. `instance` – это метка, идентифицирующая цель, откуда были извлечены данные. В данном случае она сообщает, что целью был сам сервер Prometheus.

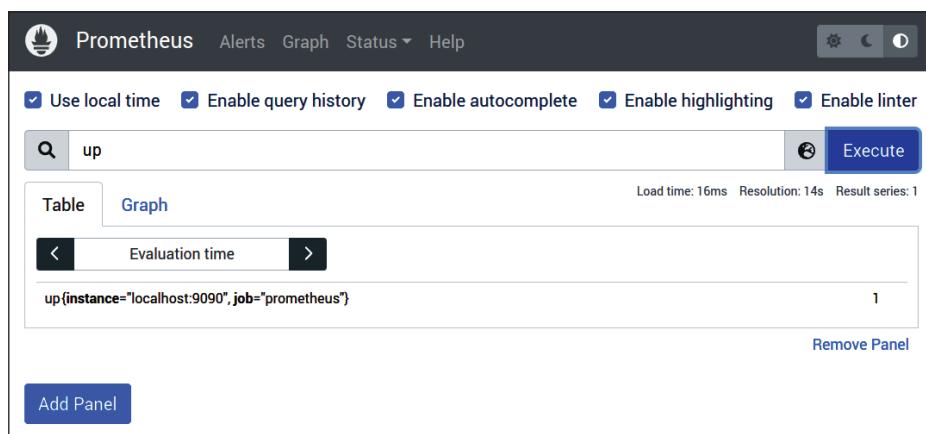


Рис. 2.5 ♦ Результат выполнения запроса «`up`» в браузере выражений

Метка `job` соответствует параметру `job_name` в `prometheus.yml`. Prometheus не знает, что извлекает данные из сервера Prometheus и поэтому использует метку `job` со значением `prometheus`. Это общепринятое соглашение, которое требует, чтобы настройка была выполнена пользователем. Метка `job` указывает тип приложения.

Далее выполним запрос `process_resident_memory_bytes`, как показано на рис. 2.6.

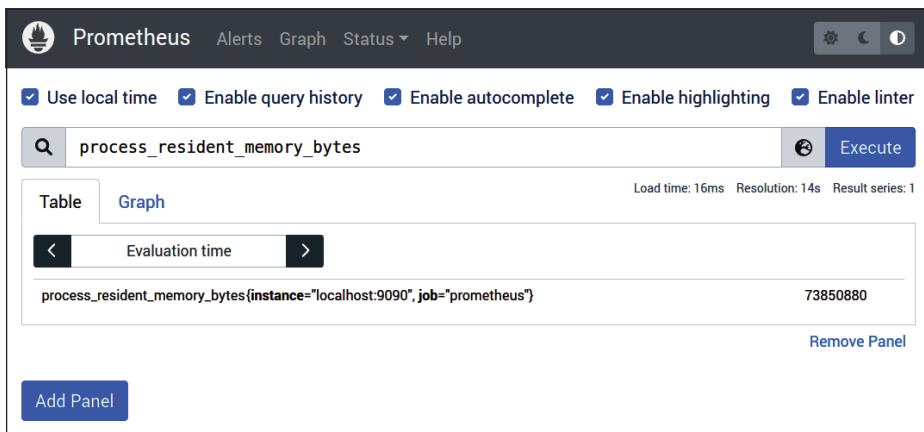


Рис. 2.6 ♦ Результат выполнения запроса `process_resident_memory_bytes` в браузере выражений

Наш сервер Prometheus использует около 73 Мбайт памяти. Возможно, вам интересно, почему эта метрика представлена в байтах, а не в мегабайтах или гигабайтах, что было бы удобнее. Ответ прост: удобство чтения во многом зависит от контекста, и даже один и тот же двоичный файл в разных окружениях может давать значения, различающиеся на несколько порядков¹. Простой запрос RPC может выполняться за микросекунды, тогда как опрос долгоживущего процесса длительный может занять часы или даже дни. Поэтому в Prometheus принято соглашение об использовании по умолчанию базовых единиц, таких как байты и секунды, и оставлять преобразование величин в более удобочитаемый формат на долю таких интерфейсных инструментов, как Grafana².

Знать текущее потребление памяти – это здорово и все такое, но еще лучше было бы видеть, как оно меняется с течением времени. Для этого щелкните по кнопке **Graph** (График), чтобы переключиться в представление графика, как показано на рис. 2.7.

¹ Число в мегабайтах можно получить, выполнив запрос вида `process_resident_memory_bytes / (1024*1024)`.

² По этой же логике даты и время обычно лучше хранить в формате UTC, а преобразования часовых поясов применять только перед отображением на экране.

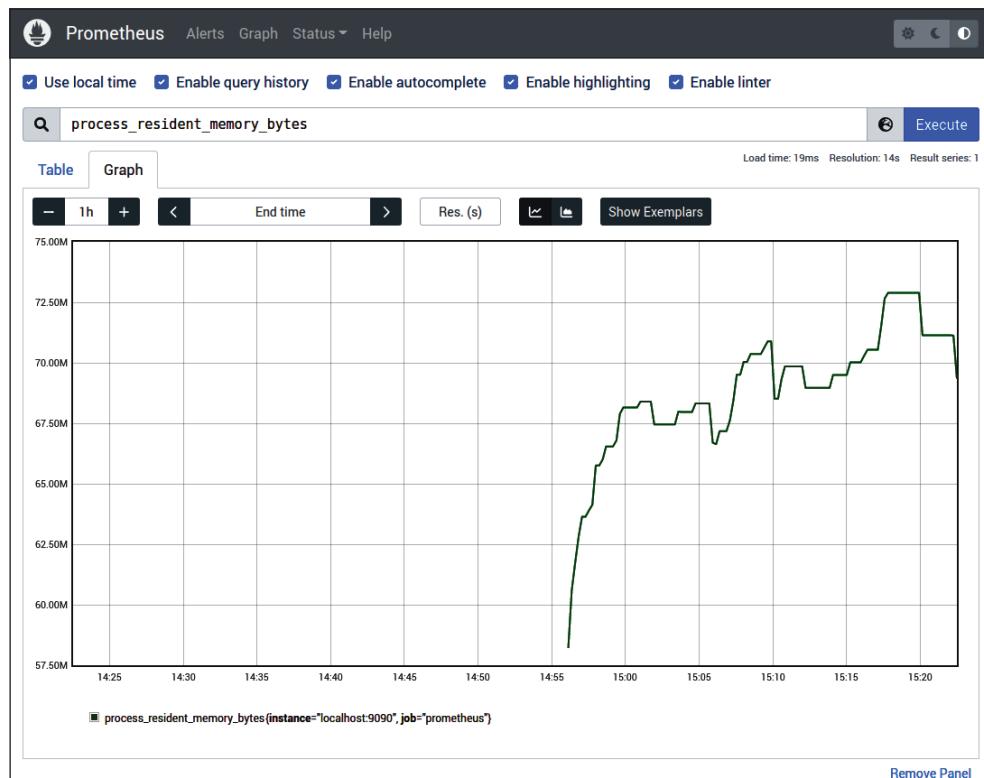


Рис. 2.7 ❖ График изменения метрики `process_resident_memory_bytes` в браузере выражений

Такие метрики, как `process_resident_memory_bytes`, называются *датчиками*. Для датчиков важно текущее абсолютное значение. Но есть еще один основной тип метрик – *счетчики*. Счетчики накапливают количество произошедших событий или общий размер данных, связанных с событиями. Давайте рассмотрим пример счетчика, построив график счетчика `prometheus_tsdb_head_samples_appended_total`, который хранит количество выборок, произведенных Prometheus (рис. 2.8).

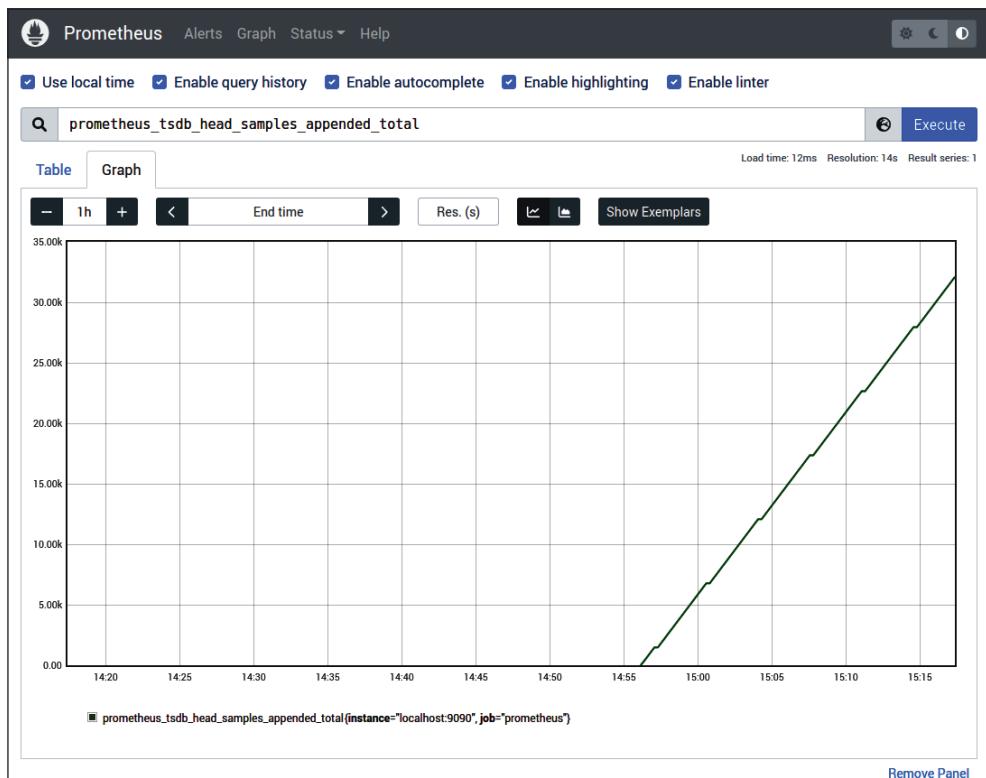


Рис. 2.8 ♦ График изменения метрики `prometheus_tsdb_head_samples_appended_total` в браузере выражений

Счетчики всегда только увеличиваются, поэтому их графики имеют вид линий, всегда стремящихся вверх и вправо. Но значения счетчиков сами по себе не очень полезны. Намного полезнее знать, как быстро увеличивается значение счетчика, и для этого можно использовать функцию скорости `rate`. Функция `rate` вычисляет, насколько быстро увеличивается значение счетчика в секунду. Измените выражение на `rate(prometheus_tsdb_head_samples_appended_total[1m])` в браузере. Оно подсчитает количество метрик, обрабатываемых в секунду, усредненное за одну минуту, и даст результат, подобный изображенному на рис. 2.9.

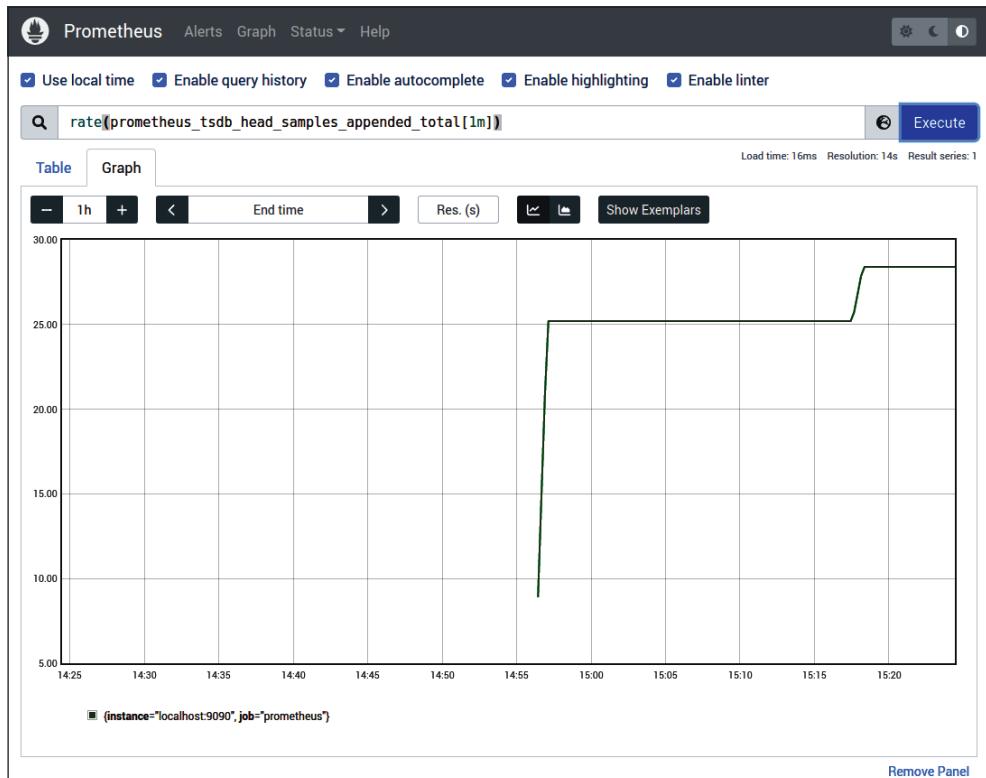


Рис. 2.9 ♦ График изменения значения `rate(prometheus_tsdb_head_samples_appended_total[1m])` в браузере выражений

Судя по этому графику, каждую секунду Prometheus принимает в среднем около 28 метрик. Функция `rate` автоматически обрабатывает сброс счетчиков при перезапуске процессов и неточность усреднения метрик¹.

Запуск экспортера узла

Экспортер узла (Node Exporter) предоставляет метрики уровня ядра систем Unix, таких как Linux². Он предоставляет все стандартные метрики, такие как нагрузка на процессор, объем памяти и дискового пространства, интенсивность дискового ввода-вывода и пропускная способность сети, а также множество дополнительных метрик, предоставляемых ядром, от средней нагрузки до температуры материнской платы.

¹ Это может привести к тому, что скорость изменения целочисленных значений будет определяться как нецелочисленная, но в среднем результаты будут верными. Дополнительные сведения см. в разделе «`rate`» главы 16.

² Пользователям Windows следует использовать экспортер для Windows (Windows Exporter; <https://oreil.ly/dB6ZZ>) вместо Node Exporter.

Однако Node Exporter не предоставляет метрик для отдельных процессов или метрик из других экспортёров или приложений – в архитектуре Prometheus мониторинг приложений и служб выполняется отдельно.

Вы можете загрузить готовую версию Node Exporter со страницы загрузки Prometheus (<https://oreil.ly/Bc4js>). Перейдите на эту страницу и загрузите последнюю версию Node Exporter для ОС Linux с аппаратной архитектурой amd64.

Архив необходимо распаковать, но файл конфигурации не требуется, поэтому экспортёр узла можно запустить сразу же после распаковки:

```
hostname $ tar -xzf node_exporter-*linux-amd64.tar.gz
hostname $ cd node_exporter-*linux-amd64/
hostname $ ./node_exporter
level=info ... msg="Starting node_exporter" version="(version=1.3.1,
branch=HEAD, revision=a2321e7b940ddcff26873612bccdf7cd4c42b6b6)"
level=info ... msg="Build context" build_context="(go=go1.17.3,
user=root@243aaafa5525c, date=20211205-11:09:49)"
level=info ... msg="Enabled collectors"
level=info ... collector=arp
level=info ... collector=bcache
level=info ... collector=bonding
...
various other collectors
...
level=info ... msg="Listening on" address=:9100
level=info ... msg="TLS is disabled." http2=false
```

Теперь можно обратиться к Node Exporter, открыв браузере страницу `http://localhost:9100/` и перейдя к конечной точке `/metrics`.

Чтобы заставить Prometheus выбирать метрики из Node Exporter, нужно добавить в `prometheus.yml` дополнительную конфигурацию выборки данных:

```
global:
  scrape_interval: 10s
scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets:
          - localhost:9090
  - job_name: node ①
    static_configs:
      - targets:
          - localhost:9100
```

① Задание для выборки данных из Node Exporter.

Перезапустите Prometheus, чтобы ввести в действие новую конфигурацию. Для этого нажмите комбинацию **Ctrl+C** в консоли, где был запущен сервер Prometheus, а затем снова запустите его¹. Если теперь открыть страницу `Tar-`

¹ Есть возможность заставить Prometheus перезагрузить файл конфигурации без перезапуска, послав ему сигнал SIGHUP.

gets (Цели), то можно увидеть, что теперь имеются две цели, обе в состоянии UP, как показано на рис. 2.10.

The screenshot shows the Prometheus Targets page. At the top, there are buttons for All, Unhealthy, and Expand All, followed by a search bar labeled "Filter by endpoint or labels". Below this, there are two sections: "node (1/1 up)" and "prometheus (1/1 up)".

node (1/1 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9100/metrics	UP	instance="localhost:9100" job="node"	13.968s ago	66.746ms	

prometheus (1/1 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9090/metrics	UP	instance="localhost:9090" job="prometheus"	791.000ms ago	4.768ms	

Рис. 2.10 ❖ Node Exporter появился в списке целей на странице состояния

Если теперь выполнить запрос **up** в представлении **Console** (Консоль) браузера выражений, то вы увидите две записи, как показано на рис. 2.11.

The screenshot shows the Prometheus Console page. At the top, there are several checkboxes: "Use local time", "Enable query history", "Enable autocomplete", "Enable highlighting", and "Enable linter". Below this is a search bar with a magnifying glass icon and the word "up", followed by an "Execute" button.

Under the search bar, there are tabs for "Table" and "Graph". The "Table" tab is selected. Below the tabs, there is a date range selector with "Evaluation time" and arrows, and a status message "Load time: 15ms Resolution: 14s Result series: 2".

The main area displays the results of the "up" query:

```
up{instance="localhost:9090",job="prometheus"} 1
up{instance="localhost:9100",job="node"} 1
```

Рис. 2.11 ❖ Теперь запрос up вернул два результата

По мере добавления новых заданий может сложиться ситуация, когда два или более разных заданий будут возвращать одну и ту же метрику. Едва ли вам захочется видеть суммарное значение одних и тех же метрик из разных заданий. Например, потребление памяти Prometheus и Node Exporter сильно различается, и их суммирование может усложнить анализ и отладку. Чтобы избежать подобной неприятности, можно построить график потребления памяти только экспортёрами узлов с помощью выражения `process_resident_memory_bytes{job="node"}`. job="node" называется *сопоставителем меток* и разграничивает возвращаемые метрики, как показано на рис. 2.12.

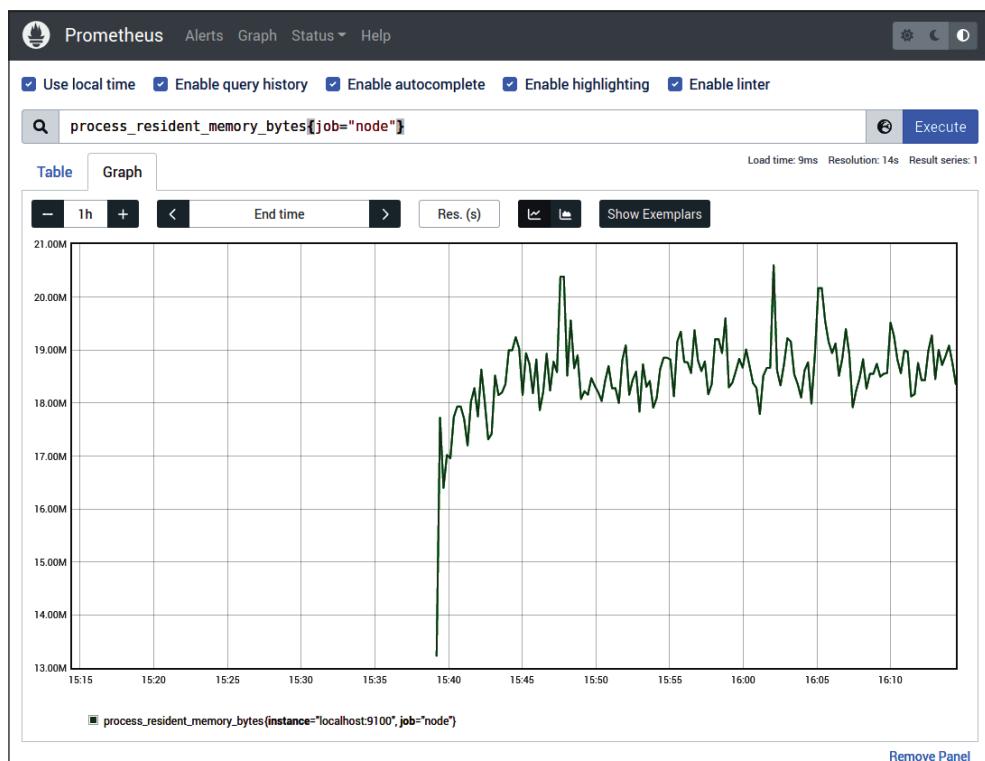


Рис. 2.12 ✦ График потребления памяти только экспортёром узла Node Exporter

Здесь `process_resident_memory_bytes` – это объем памяти, используемый самим процессом Node Exporter (на что намекает префикс `process`), а не всей системой в целом. Знать, сколько ресурсов потребляет Node Exporter, конечно, интересно, но это не то, для чего вы его запускаете.

В качестве последнего примера выполним запрос `rate(node_network_receive_bytes_total[1m])` в представлении **Graph** (График), который выведет график, подобный изображенному на рис. 2.13.

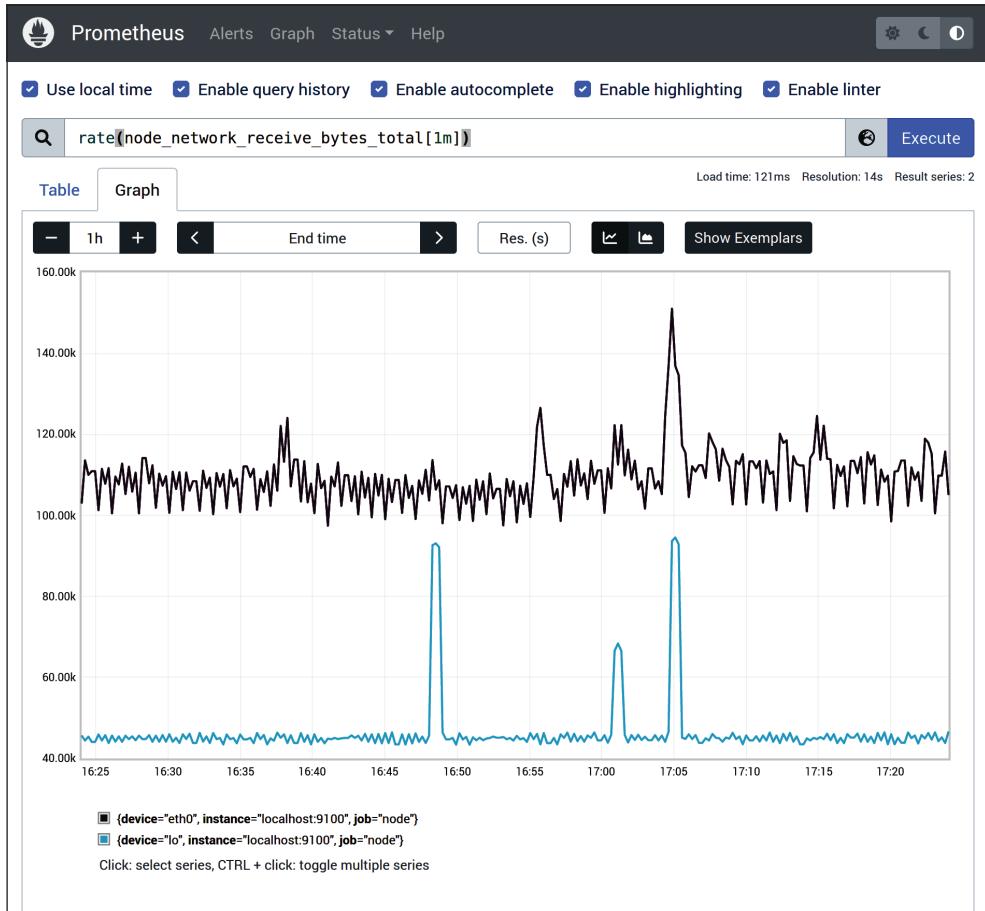


Рис. 2.13 ♦ График изменения входящего сетевого трафика по нескольким интерфейсам

`node_network_receive_bytes_total` – это счетчик байтов, полученных сетевыми интерфейсами. Node Exporter автоматически подхватывает все сетевые интерфейсы, и с ними можно работать в PromQL как с группой. Это очень удобно для оповещения, поскольку метки позволяют избежать необходимости создавать исчерпывающий список всего, о чем вы хотели бы предупредить.

Оповещение

Настройка и рассылка оповещений выполняется в два этапа. Сначала добавляются в Prometheus правила оповещения с логикой, определяющей необходимость отправки оповещения. На втором этапе диспетчер уведомлений

Alertmanager преобразует оповещения в уведомления, такие как электронные письма, SMS и сообщения в чате.

Начнем с определения условия, о выполнении которого вы хотели бы предупредить. Остановите Node Exporter комбинацией **Ctrl+C**. После следующего цикла извлечения метрик на странице **Targets** (Цели) напротив Node Exporter будет отображаться состояние DOWN, как показано на рис. 2.14, с сообщением об ошибке *connection refused* (отказано в соединении), так как соответствующий TCP-порт не обслуживает никакой процесс, и HTTP-запрос отклоняется¹.

- ✓ Prometheus не фиксирует в журналах приложений неудачные попытки извлечь данные, потому что это вполне ожидаемое событие, которое не указывает на какие-либо проблемы в самом сервере Prometheus. Помимо страницы **Targets** (Цели), ошибки извлечения данных также доступны в журналах отладки Prometheus, поддержку которых можно включить, передав в командной строке параметр `--log.level debug`.

node (0/1 up)					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9100/metrics	DOWN	instance='localhost:9100' job='node'	2.489s ago	1.761ms	Get "http://localhost:9100/metrics": dial tcp [::1]:9100: connect: connection refused

prometheus (1/1 up)					
Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9090/metrics	UP	instance='localhost:9090' job='prometheus'	4.312s ago	6.314ms	

Рис. 2.14 ♦ На странице состояния целей Node Exporter отображается как недоступный

Вручную просматривать страницу **Targets** (Цели) в поисках неработающих экземпляров – не лучшая тратка времени. К счастью, у нас имеется метрика `up`, и при оценке выражения `up` в представлении **Console** (Консоль) браузера выражений вы увидите, что теперь для Node Exporter она имеет значение 0, как показано на рис. 2.15.

¹ Еще одна распространенная ошибка – *context deadline exceeded* (превышен крайний срок контекста). Она указывает на тайм-аут, что нередко случается из-за слишком медленной работы запрашиваемой стороны или из-за потери пакетов в сети.

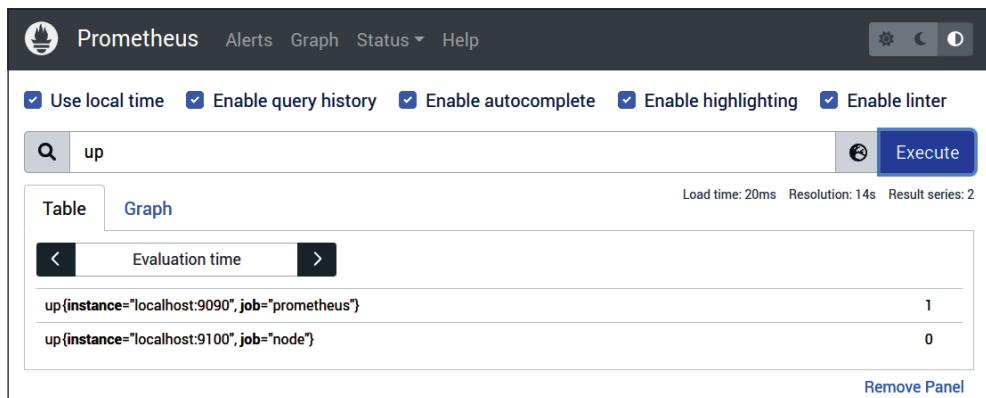


Рис. 2.15 ♦ Теперь метрика up для Node Exporter имеет значение 0

Чтобы определить правило оповещения, нужно задать выражение PromQL, возвращающее только те результаты, о которых вы хотели бы уведомить. В данном случае это легко сделать с помощью оператора `==`. Он отфильтрует¹ все временные ряды, значения которых не совпадают. Если выполнить запрос `up == 0` в обозревателе выражений, то он вернет только недоступный экземпляр, как показано на рис. 2.16.

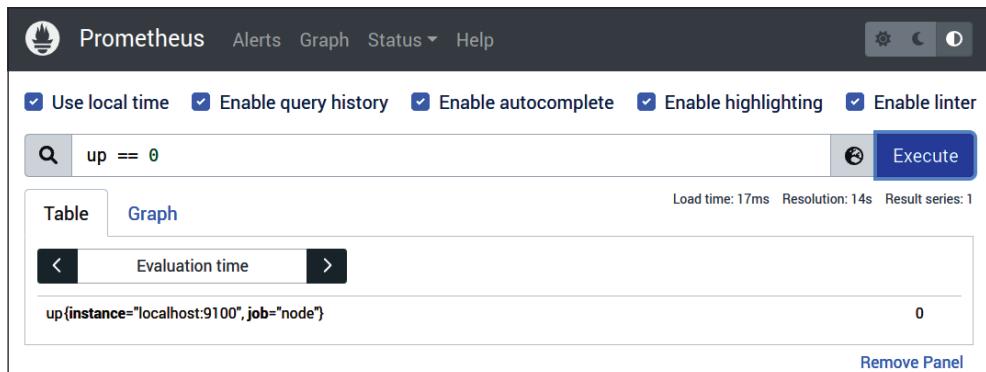


Рис. 2.16 ♦ Возвращаются только экземпляры, в которых метрика up имеет значение 0

Затем нужно добавить это выражение в правило оповещения в Prometheus. Мы также немного забежим вперед и укажем серверу Prometheus, с каким Alertmanager он будет взаимодействовать. Для этого дополним содержимое `prometheus.yml`, как показано в примере 2.1.

¹ Существует также режим `bool` без фильтрации (см. описание в разделе «Модификатор `bool`» главы 15).

Пример 2.1 ♦ `prometheus.yml` с настройками для получения данных из двух целей, загрузки файла с правилами оповещения и используемым Alertmanager

```
global:
  scrape_interval: 10s
  evaluation_interval: 10s
rule_files: ①
  - rules.yml
alerting: ②
  alertmanagers:
    - static_configs:
      - targets:
          - localhost:9093
scrape_configs: ③
  - job_name: prometheus
    static_configs:
      - targets:
          - localhost:9090
  - job_name: node
    static_configs:
      - targets:
          - localhost:9100
```

① Файлы с правилами.

② Настройки оповещения.

③ Задания для извлечения информации.

Теперь создадим новый файл `rules.yml` с содержимым, показанным в примере 2.2, а затем перезапустим Prometheus.

Пример 2.2 ♦ `rules.yml` с одним правилом оповещения

```
groups:
  - name: example
    rules:
      - alert: InstanceDown
        expr: up == 0
        for: 1m
```

Условие отправки оповещения `InstanceDown` будет оцениваться каждые 10 с, как определено параметром `evaluation_interval`. Если серия непрерывно возвращается в течение хотя бы одной минуты¹ (`for`), то условие будет считаться выполненным. Пока требуемая минута не истечет, оповещение будет пребывать в состоянии **ожидания**. На странице **Alerts** (Оповещения) можно щелкнуть на этом оповещении и просмотреть дополнительные сведения, включая метки, как показано на рис. 2.17.

¹ Обычно рекомендуется не менее 5 мин, чтобы уменьшить влияние шума и различных состояний гонки, присущих мониторингу. Мы используем одну минуту, только чтобы вам не пришлось слишком долго ждать результата.

The screenshot shows the Prometheus interface with the 'Alerts' tab selected. At the top, there are filters for 'Inactive (0)', 'Pending (0)', and 'Firing (1)'. A search bar and a 'Show annotations' checkbox are also present. Below the filters, a list of alerts is shown, starting with 'rules.yml > example'. Underneath this, a specific alert is expanded: 'InstanceDown (1 active)'. The alert details are as follows:

```

name: InstanceDown
expr: up == 0
for: 1m
  
```

A table below shows the alert's state and active since information:

Labels	State	Active Since	Value
alarmname=InstanceDown, instance=localhost:9100, job=node	FIRING	2022-07-18T14:27:59.456891295Z	0

Рис. 2.17 ❖ Оповещение на странице **Alerts** (Оповещения)

Теперь, имея оповещение, нам нужен Alertmanager, чтобы он обслужил это оповещение. Загрузите последнюю версию Alertmanager для ОС Linux с аппаратной архитектурой amd64. Разархивируйте Alertmanager и перейдите в каталог, куда он был распакован:

```
hostname $ tar -xzf alertmanager-*.linux-amd64.tar.gz
hostname $ cd alertmanager-*.linux-amd64/
```

Теперь настроим Alertmanager. Существует множество способов, которыми Alertmanager может уведомить нас о сложившейся ситуации, но большинство из тех, что работают «из коробки», используют коммерческих провайдеров и имеют инструкции по настройке, которые со временем меняются. Поэтому предположим, что у вас есть открытый смарт-хост SMTP¹. Создайте файл *alertmanager.yml*, используя в качестве основы код из примера 2.3 и настроив *smtp_smarthost*, *smtp_from* и *to* в соответствии с вашими настройками и адресом электронной почты.

Пример 2.3 ❖ *alertmanager.yml* отправляет все оповещения на электронную почту

```

global:
  smtp_smarthost: 'localhost:25'
  smtp_from: 'yourprometheus@example.org' ❶
route:
  receiver: example-email
  group_by: [alertname]
receivers:
  - name: example-email
  
```

¹ Учитывая, как развивалась безопасность электронной почты за последнее десятилетие, это не очень хорошее предположение, но у вашего интернет-провайдера есть такой хост.

```
email_configs:
- to: 'youraddress@example.org' ②
```

- ① Адрес электронной почты, который будет использоваться в поле «От» («From»).
 ② Адрес электронной почты, куда будут отправляться электронные письма.

Теперь запустите Alertmanager командой `./alertmanager`:

```
hostname $ ./alertmanager
level=info ... msg="Starting Alertmanager" version="(version=0.24.0,
branch=HEAD, revision=f484b17fa3c583ed1b2c8bbcec20ba1db2aa5f11)"
level=info ... build_context="(go=go1.17.8, user=root@265f14f5c6fc,
date=20220325-09:31:33)"
level=info ... component=cluster msg="setting advertise address
explicitly" addr=192.168.10.52 port=9094
level=info ... component=cluster msg="Waiting for gossip to settle..."
interval=2s
level=info ... component=configuration msg="Loading configuration file"
file=alertmanager.yml
level=info ... component=configuration msg="Completed loading of
configuration file" file=alertmanager.yml
level=info ... msg=Listening address=:9093
level=info ... msg="TLS is disabled." http2=false
level=info component=cluster ... msg="gossip not settled" polls=0 before=0
now=1 elapsed=2.00004715s
level=info component=cluster ... msg="gossip settled; proceeding"
elapsed=10.001771352s
polls=0 before=0 now=1 elapsed=2.00011639s
```

Теперь откройте в своем браузере страницу `http://localhost:9093/`, чтобы получить доступ к Alertmanager. На этой странице вы увидите оповещение, как показано на рис. 2.18.

The screenshot shows the Alertmanager web interface. At the top, there's a navigation bar with tabs for 'Alerts', 'Silences', 'Status', and 'Help', and a 'New Silence' button. Below the navigation bar, there are two tabs: 'Filter' (selected) and 'Group'. To the right of these tabs are filter options: 'Receiver: All', 'Silenced' (unchecked), and 'Inhibited' (unchecked). A search input field contains the placeholder 'Custom matcher, e.g. env="production"'. Below the search field is a '+' button and a 'Silence' button with a crossed-out icon. Underneath the search area, there's a link '+ Expand all groups'. The main content area displays a single alert entry:

- alertname="InstanceDown"** **+ 1 alert**
- 2023-03-18T16:29:43.691Z** **✗ Silence**
- instance="localhost:9100"** **+** **job="node"** **+**

Рис. 2.18 ♦ Оповещение InstanceDown в Alertmanager

Если все настроено и работает правильно, то через минуту или две вы получите уведомление от Alertmanager на свою электронную почту, которое будет выглядеть, как показано на рис. 2.19.

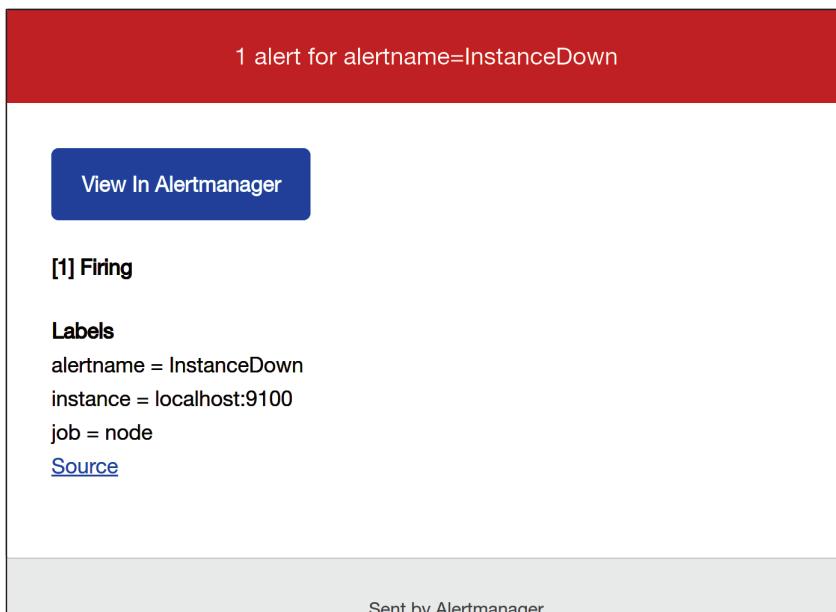


Рис. 2.19 ♦ Электронное письмо с уведомлением InstanceDown

Эта базовая конфигурация помогла вам получить общее представление о возможностях Prometheus. Вы можете попробовать добавить другие цели в *prometheus.yml*, и ваше оповещение будет автоматически отправляться и для них.

В следующей главе мы сосредоточимся на конкретном аспекте использования Prometheus – инструментировании ваших собственных приложений.

Часть



МОНИТОРИНГ ПРИЛОЖЕНИЙ

Теперь вы знаете, какие преимущества дает Prometheus при наличии простого доступа к метрикам, которые вы добавите в свои приложения. В этом разделе мы посмотрим, как это сделать.

В главе 3 вы узнаете, как осуществить базовое инструментирование и какие метрики желательно получать.

В главе 4 мы покажем, как сделать метрики приложения доступными для Prometheus.

В главе 5 вы познакомитесь с одной из самых мощных функций Prometheus и особенностях ее использования.

После получения метрик приложения в Prometheus в главе 6 мы покажем, как создавать дашборды, группирующие связанные графики вместе.

Глава 3

Инструментирование

Самая большая выгода, которую дает Prometheus, – это возможность *прямого инструментирования* ваших собственных приложений с применением клиентской библиотеки. Клиентские библиотеки доступны для разных языков программирования, в том числе официальные библиотеки для Go, Python, Java, Rust и Ruby.

В качестве примера мы используем Python 3, но те же самые принципы применимы к другим языкам и средам выполнения, хотя синтаксис и служебные методы будут различаться.

Большинство современных операционных систем поставляются с Python 3 в их составе. В том маловероятном случае, если у вас еще нет Python 3, загрузите и установите его (<https://oreil.ly/6sAX9>).

Вам также потребуется установить последнюю версию клиентской библиотеки для Python. Сделать это можно командой `pip install prometheus_client`. Примеры инструментирования можно найти на GitHub (<https://oreil.ly/-IbFJ>).

Простая программа

Для начала мы написали простой HTTP-сервер, показанный в примере 3.1. Запустив его под управлением интерпретатора Python 3, а затем открыв страницу `http://localhost:8001/` в браузере, вы получите ответ «Hello World».

Пример 3.1 ♦ Простая программа «Hello World», которая экспортирует метрики Prometheus.

```
import http.server
from prometheus_client import start_http_server

class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello World")
```

```
if __name__ == "__main__":
    start_http_server(8000)
    server = http.server.HTTPServer(('localhost', 8001), MyHandler)
    server.serve_forever()
```

Вызов `start_http_server(8000)` запускает HTTP-сервер на порту 8000, способный передавать метрики Prometheus. Эти метрики можно просмотреть на странице `http://localhost:8000/`, как показано на рис. 3.1. Какие метрики возвращаются по умолчанию, зависит от платформы, однако платформы Linux, как правило, имеют наибольшее количество метрик.

```
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 131.0
python_gc_objects_collected_total{generation="1"} 244.0
python_gc_objects_collected_total{generation="2"} 0.0
# HELP python_gc_objects_uncollectable_total Uncollectable object found during GC
# TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total{generation="0"} 0.0
python_gc_objects_uncollectable_total{generation="1"} 0.0
python_gc_objects_uncollectable_total{generation="2"} 0.0
# HELP python_gc_collections_total Number of times this generation was collected
# TYPE python_gc_collections_total counter
python_gc_collections_total{generation="0"} 38.0
python_gc_collections_total{generation="1"} 3.0
python_gc_collections_total{generation="2"} 0.0
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation="CPython", major="3", minor="9", patchlevel="13", version="3.9.13"} 1.0
```

Рис. 3.1 ♦ Страница /metrics,
доступная после запуска простой программы в Linux под управлением CPython

Страницу `/metrics` можно просматривать вручную, но в действительности нам нужно получить их в Prometheus. Для этого настройте сервер Prometheus с конфигурацией из примера 3.2 и запустите его.

Пример 3.2 ♦ Конфигурация prometheus.yml для получения метрик из `http://localhost:8000/metrics`

```
global:
  scrape_interval: 10s
scrape_configs:
  - job_name: example
    static_configs:
      - targets:
          - localhost:8000
```

Если открыть страницу `http://localhost:9090/` браузера выражений и ввести выражение PromQL `python_info`, то вы получите результат, показанный на рис. 3.2.

В оставшейся части этой главы мы будем предполагать, что сервер Prometheus запущен и выполняет мониторинг примера приложения. Для анализа созданных вами метрик вы будете использовать браузер выражений.

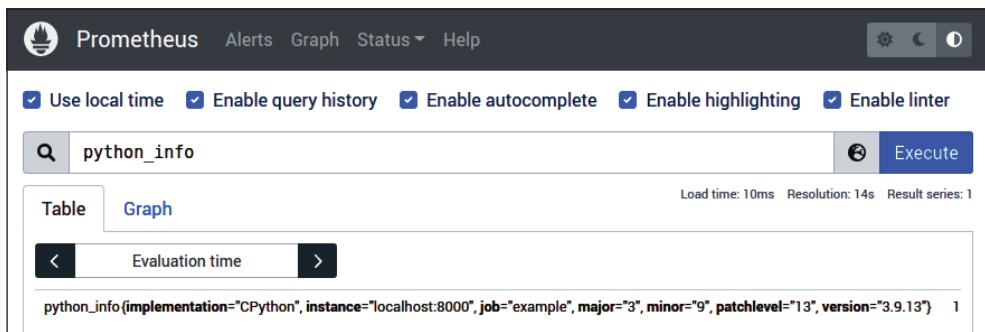


Рис. 3.2 ♦ Запрос python_info возвращает единственный результат

Счетчики

Счетчики – это метрики, которые вы, вероятно, будете создавать чаще всего. Счетчики отслеживают количества или размеры событий. В основном они используются для определения, как часто выполняется конкретный путь в коде.

Усовершенствуйте пример 3.1, добавив новую метрику, которая суммирует количество запросов, обработанных приложением «Hello World», как показано в примере 3.3.

Пример 3.3 ♦ REQUESTS суммирует количество сообщений, возвращенных приложением «Hello World»

```
from prometheus_client import Counter

REQUESTS = Counter('hello_worlds_total',
                    'Hello Worlds requested.')

class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        REQUESTS.inc()
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello World")
```

Код делится на три основные части – импорт библиотек, определение метрики и инструментирование.

Import

Инструкции `import` в языке Python импортируют функции и классы из других модулей, делая их доступными для использования. В данном случае вы должны импортировать класс `Counter` из библиотеки `prometheus_client`.

Определение

Метрики Prometheus должны быть определены до их использования. Здесь мы определяем счетчик с именем `hello_worlds_total`. В определении при-

существует строка справки «Hello Worlds requested.», которая появится на странице `/metrics` и поможет вам понять, что означает эта метрика.

Клиентская библиотека автоматически регистрирует метрики в *реестре по умолчанию*¹. Реестр – это место, где регистрируются метрики, которые должны быть доступны для мониторинга. Реестр по умолчанию – это реестр, в котором по умолчанию выполняется поиск метрик при получении запроса `/metrics`. В некоторых случаях может быть удобно использовать пользовательский реестр, например когда разрабатывается библиотека для подключения к другому программному обеспечению.

В библиотеке Java, например, требуется выполнить дополнительный вызов функции, а в библиотеке Go, в зависимости от особенностей ее использования, может потребоваться явно зарегистрировать метрики. Нет необходимости возвращать метрику обратно в вызов `start_http_server`; на самом деле реализация полностью отделена от определения. Если у вас есть промежуточная зависимость, осуществляющая инструментирование, то метрики, создаваемые ею, автоматически появятся на странице `/metrics`. Метрики должны иметь уникальные имена, а клиентские библиотеки должны сообщать об ошибке при попытке дважды зарегистрировать одну и ту же метрику. Чтобы избежать возможных проблем, определяйте метрики на уровне файла, а не на уровне класса, функции или метода. Альтернативным решением является использование своих реестров и локальных определений.

Инструментирование

Определив объект метрики, вы можете его использовать. Метод `inc` увеличивает значение счетчика на единицу.

Обо всех тонкостях позаботятся клиентские библиотеки Prometheus, поэтому от вас требуется реализовать только это.

После запуска программы новая метрика появится на странице `/metrics`. Она получит начальное нулевое значение и будет увеличиваться на единицу² каждый раз, когда вы попытаетесь открыть основной URL приложения. Убедиться в этом можно, открыв браузер выражений и использовав запрос PromQL `rate(hello_worlds_total[1m])`, чтобы узнать, сколько запросов в секунду обрабатывает «Hello World», как показано на рис. 3.3.

Всего двумя строками можно добавить счетчик в любую библиотеку или приложение. Эти счетчики помогут вам узнать, сколько раз возникают ошибки и непредвиденные ситуации. Возможно, вам не требуется выдавать предупреждение о каждой ошибке, но знать, как частота ошибок меняется с течением времени, может очень пригодиться при отладке. Однако счетчики применяются не только для подсчета ошибок. С их помощью также можно узнать, как часто используются наиболее популярные особенности вашего приложения, что позволит вам оптимизировать распределение усилий по разработке.

¹ К сожалению, по разным техническим причинам не все клиентские библиотеки делают это автоматически.

² Оно может увеличиться на два из-за того, что веб-браузер также использует конечную точку `/favicon.ico`.

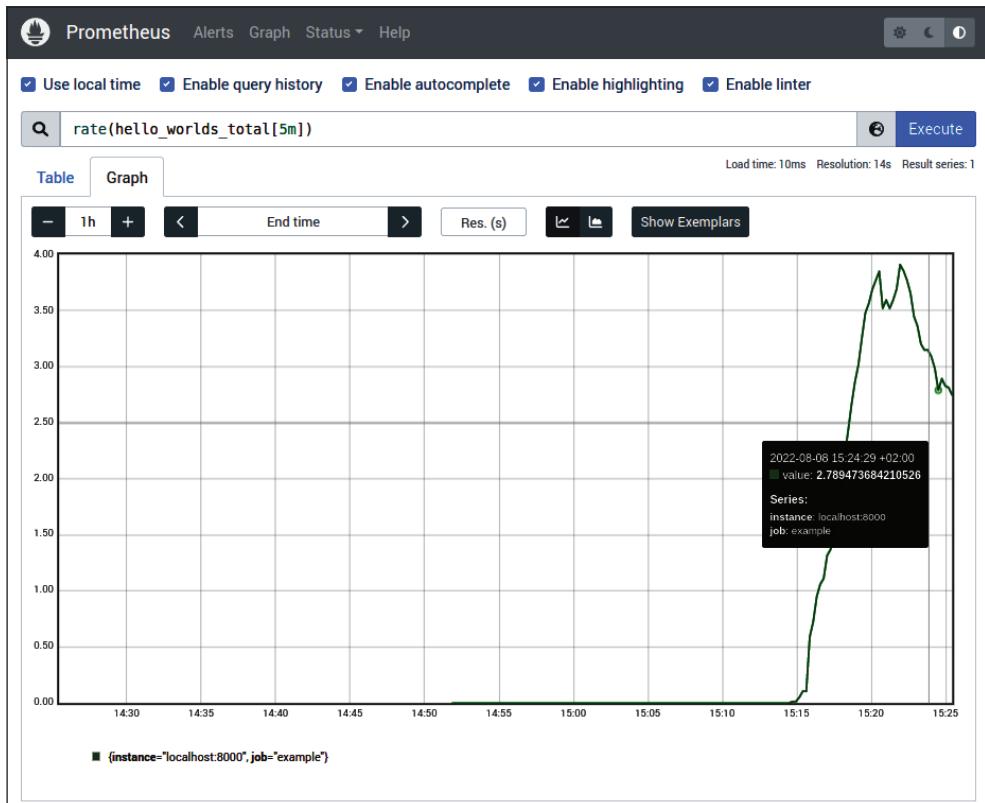


Рис. 3.3 ❖ График количества обращений в секунду к приложению «Hello Worlds»

Подсчет исключений

Клиентские библиотеки предоставляют не только основные возможности, но также утилиты и методы для типичных случаев использования. Один из них в Python – возможность подсчета исключений. Вам не нужно писать свой инструментирующий код, используя `try...except`, достаточно воспользоваться диспетчером контекста `count_exceptions` и декоратором, как показано в примере 3.4.

Пример 3.4 ❖ `EXCEPTIONS` подсчитывает количество исключений с помощью диспетчера контекста

```
import random
from prometheus_client import Counter

REQUESTS = Counter('hello_worlds_total',
    'Hello Worlds requested.')
EXCEPTIONS = Counter('hello_world_exceptions_total',
```

```
'Exceptions serving Hello World.')
```

```
class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        REQUESTS.inc()
        with EXCEPTIONS.count_exceptions():
            if random.random() < 0.2:
                raise Exception
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello World")
```

`count_exceptions` позаботится о передаче исключения вверх, возбудив его, поэтому оно не помешает логике приложения. Увидеть частоту исключений можно с помощью выражения `rate(hello_world_exceptions_total[1m])`. Количество исключений само по себе мало полезно, если не знать количества обработанных запросов. Вы можете рассчитать более полезное соотношение исключений, выполнив в браузере выражений запрос:

```
rate(hello_world_exceptions_total[1m])
/
rate(hello_worlds_total[1m])
```

Именно так обычно экспортят соотношения: создают два счетчика, а затем получают их и делят друг на друга в PromQL.

i Вы можете заметить пустые участки на графике соотношения исключений в периоды, когда приложение не получает запросов. Это объясняется делением на ноль, что в математике с плавающей запятой дает значение *NaN* (Not a Number – не число). Возвращать нуль было бы неправильно, потому что соотношение исключений не равно нулю, а просто не определено.

Также `count_exceptions` можно использовать в качестве декоратора:

```
EXCEPTIONS = Counter('hello_world_exceptions_total',
    'Exceptions serving Hello World.')

class MyHandler(http.server.BaseHTTPRequestHandler):
    @EXCEPTIONS.count_exceptions()
    def do_GET(self):
        ...
```

Подсчет размера

Для представления чисел с плавающей точкой Prometheus использует 64-битный формат, поэтому счетчики можно увеличивать не только на целочисленные значения, но и на любое неотрицательное число. Это позволяет отслеживать количество обработанных записей и байтов или продаж в евро, как показано в примере 3.5.

Пример 3.5 ♦ SALES отслеживает сумму продаж в евро

```
import random
from prometheus_client import Counter

REQUESTS = Counter('hello_worlds_total',
    'Hello Worlds requested.')
SALES = Counter('hello_world_sales_euro_total',
    'Euros made serving Hello World.')

class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        REQUESTS.inc()
        euros = random.random()
        SALES.inc(euros)
        self.send_response(200)
        self.end_headers()
        self.wfile.write("Hello World for {} euros.".format(euros).encode())
```

Увидеть, как меняется объем продаж в евро в секунду, можно в браузере выражений, введя выражение `rate(hello_world_sales_euro_total[1m])`.

! Попытка увеличить счетчик на отрицательное число считается ошибкой и вызывает исключение.

Для PromQL важно, чтобы счетчики только увеличивались, и функция `rate` и другие подобные ей функции не интерпретировали по ошибке уменьшение счетчика как сброс до нуля при перезапуске приложения. Это также означает, что нет необходимости сохранять состояние счетчика в процессе выполнения приложения или сбрасывать счетчики при извлечении очередной порции данных. Это позволяет нескольким серверам Prometheus опрашивать одно и то же приложение, не мешая друг другу.

Датчики

Датчики – это моментальный снимок некоторого текущего состояния. Если при интерпретации счетчиков нас чаще интересует скорость их изменения, то при интерпретации наибольший интерес представляют их фактические значения. Соответственно, значения могут не только увеличиваться, но и уменьшаться.

Вот несколько примеров датчиков:

- количество элементов в очереди;
- потребление памяти кешем;
- количество активных потоков;
- время обработки последней записи;
- среднее количество запросов в секунду за последнюю минуту¹.

¹ Это значение – датчик, но его лучше экспортовать с помощью счетчика. Счетчик запросов, изменяющийся с течением времени, можно преобразовать в датчик с помощью PromQL-функции `rate`.

Использование датчиков

Датчики имеют три основных метода: `inc`¹, `dec` и `set`. Подобно методам счетчиков, `inc` и `dec` по умолчанию изменяют значение датчика на единицу. При необходимости этим методам можно передать значение, на которое следует увеличить или уменьшить датчик. В примере 3.6 показано, как использовать датчики для отслеживания количества выполняемых вызовов и определения времени завершения последнего из них.

Пример 3.6 ♦ INPROGRESS и LAST отслеживают количество текущих запросов и время завершения последнего из них

```
import time
from prometheus_client import Gauge

INPROGRESS = Gauge('hello_worlds_inprogress',
    'Number of Hello Worlds in progress.')
LAST = Gauge('hello_world_last_time_seconds',
    'The last time a Hello World was served.')

class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        INPROGRESS.inc()
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello World")
        LAST.set(time.time())
        INPROGRESS.dec()
```

Эти метрики можно использовать в браузере выражений непосредственно, без применения дополнительных функций. Например, выполнив выражение `hello_world_last_time_seconds`, можно узнать, когда приложение «Hello World» обслужило последний запрос. Подобные метрики обычно используются, чтобы определить, сколько времени прошло с момента обработки последнего запроса. PromQL-выражение `time() - hello_world_last_time_seconds` сообщит, сколько секунд прошло с момента последнего запроса.

Это очень распространенные варианты использования, поэтому для них предусмотрены специальные функции, демонстрируемые в примере 3.7. Преимущество `track_inprogress` в том, что ее имя короче и она берет на себя хлопоты по правильной обработке исключений. Функция `set_to_current_time` не так полезна в Python, потому что `time.time()` возвращает время Unix в секундах², но в клиентских библиотеках других языков эквиваленты `set_to_current_time` делают использование более простым и понятным.

¹ В отличие от счетчиков датчики могут уменьшаться, поэтому в вызов метод `inc` датчика можно передавать отрицательные числа.

² Секунды – это базовая единица времени, поэтому в Prometheus они предпочтительнее других единиц, таких как минуты, часы, дни, милли-, микро- и наносекунды.

Пример 3.7 ♦ Этот код делает то же, что и код в примере 3.6, но использует специализированные функции датчика

```
from prometheus_client import Gauge

INPROGRESS = Gauge('hello_worlds_inprogress',
                   'Number of Hello Worlds in progress.')
LAST = Gauge('hello_world_last_time_seconds',
             'The last time a Hello World was served.')

class MyHandler(http.server.BaseHTTPRequestHandler):
    @INPROGRESS.track_inprogress()
    def do_GET(self):
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello World")
        LAST.set_to_current_time()
```

Суффиксы в именах метрик

Возможно, вы заметили, что имена всех метрик в примерах со счетчиками заканчиваются на `_total`, тогда как в примерах с датчиками такого суффикса нет. Это общепринятое соглашение Prometheus, помогающее определить тип метрики, с которой вы работаете.

В OpenMetrics этот суффикс обязателен. Библиотека `prometheus_client` в Python является эталонной реализацией OpenMetrics, поэтому, если вы не добавите суффикс сами, то библиотека добавит его за вас.

Помимо `_total`, также поддерживаются суффиксы `_count`, `_sum` и `_bucket`, имеющие особый смысл, и они не должны использоваться в качестве суффиксов в именах метрик, чтобы избежать путаницы.

Также настоятельно рекомендуется указывать единицу измерения в конце имени. Например, счетчику обработанных байтов можно присвоить имя `myapp_requests_processed_bytes_total`.

Обратные вызовы

Для отслеживания размеров или количеств элементов в кеше обычно следует добавлять вызовы `inc` и `dec` во все функции, добавляющие или удаляющие элементы из кеша. Это может вызывать трудности по мере развития прикладного кода. К счастью, клиентские библиотеки предлагают быстрый способ реализовать все это без использования интерфейсов, которые необходимы для написания экспортёров.

В Python датчики имеют метод `set_function`, которому можно передать функцию для вызова в извлечения данных сервером. Она должна возвращать значение метрики с плавающей точкой, как показано в примере 3.8. Однако этот прием выходит за рамки прямого инструментирования, поэтому вам придется учитывать безопасность потоков и, возможно, использовать мьютексы.

Пример 3.8 ♦ Тривиальный пример `set_function`, для возврата метрики с текущим временем¹

```
import time
from prometheus_client import Gauge

TIME = Gauge('time_seconds',
             'The current time.')
TIME.set_function(lambda: time.time())
```

Сводные метрики

Информация о задержке, т. е. о том, сколько времени потребовалось приложению для обработки запроса, является ценной метрикой, особенно на этапе исследования особенностей работы систем. Многие системы инструментирования предлагают метрику Timer в том или ином виде, но Prometheus использует более обобщенный подход. Учитывая, что счетчики могут увеличиваться на значения, отличные от единицы, с их помощью можно отслеживать не только временные задержки, но и, например, размеры получаемых ответов.

Основным обобщенным методом является `observe`, которому можно передать размер события. Это должно быть неотрицательное значение. В примере 3.9 показано, как с помощью `time.time()` можно отслеживать задержки.

Пример 3.9 ♦ LATENCY отслеживает, сколько времени требуется обработчику «Hello World» для запуска

```
import time
from prometheus_client import Summary

LATENCY = Summary('hello_world_latency_seconds',
                  'Time for a request Hello World.')

class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        start = time.time()
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello World")
        LATENCY.observe(time.time() - start)
```

Если теперь открыть страницу `/metrics`, то на ней можно увидеть, что метрика `hello_world_latency_seconds` имеет два временных ряда: `hello_world_latency_seconds_count` и `hello_world_latency_seconds_sum`.

`hello_world_latency_seconds_count` отсчитывает количество вызовов `observe`, поэтому `rate(hello_world_latency_seconds_count[1m])` в браузере выражений вернет частоту запросов к «Hello World» в секунду.

¹ На практике в такой метрике нет особой необходимости – ее можно получить с помощью функции `timestamp` в PromQL, а с помощью функции `time` можно узнать время обработки запроса.

`hello_world_latency_seconds_sum` – это сумма значений, переданных в вызов `observe`, поэтому `rate(hello_world_latency_seconds_sum[1m])` вернет количество времени, расходуемое каждой секундой на обработку запросов.

Разделив второе выражение на первое, вы получите среднюю задержку за последнюю минуту. Полное выражение для средней задержки будет выглядеть так:

```
rate(hello_world_latency_seconds_sum[1m])
/
rate(hello_world_latency_seconds_count[1m])
```

Рассмотрим простой пример. Допустим, за последнюю минуту приложение получило три запроса, на обработку которых ушло 2, 4 и 9 с. Количество запросов будет равно 3, а суммарное время обработки составит 15 с, соответственно, средняя задержка будет равна 5 с. Функция `rate` вычисляет частоту в секунду, а не в минуту, поэтому обе части нужно разделить на 60, но в данном случае этого не требуется, потому что единицы измерения в числителе и знаменателе компенсируют друг друга.

i Несмотря на то что в соответствии с соглашениями Prometheus метрика `hello_world_latency_seconds` использует секунды как базовую единицу измерения времени, это не означает, что она может иметь только секундную точность. 64-битные значения с плавающей точкой в Prometheus могут представлять метрики времени в диапазоне от нескольких дней до наносекунд.

Поскольку сводные данные обычно используются для отслеживания задержек, для этой цели был разработан диспетчер контекста и декоратор `time`, который упрощает эту задачу, как показано в примере 3.10. Он также автоматически обрабатывает исключения и значения времени¹.

Пример 3.10 ♦ LATENCY отслеживание задержки с помощью декоратора time

```
from prometheus_client import Summary

LATENCY = Summary('hello_world_latency_seconds',
                  'Time for a request Hello World.')

class MyHandler(http.server.BaseHTTPRequestHandler):
    @LATENCY.time()
    def do_GET(self):
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello World")
```

Сводные метрики могут также включать квантили, однако в настоящее время клиентская библиотека для Python не поддерживает квантили. В общем случае их следует избегать, потому что квантили не позволяют выполнять такие математические операции, как усреднение, и агрегировать

¹ Системное время можно вернуть, если вручную установить дату в ядре или если демон синхронизирует время по протоколу NTP.

квантили на стороне клиента по экземплярам службы. Кроме того, анализ квантилей на стороне клиента требует дополнительных затрат (квантили могут обрабатываться в сто раз медленнее). Хотя преимущества инструментирования, как правило, оправдывают затраты ресурсов, но это редко относится к квантилям.

Гистограммы

Сводные метрики покажут вам среднюю задержку, но как быть, если желательно знать квантиль? Квантиль сообщает, какую долю от общего числа составили события с размером ниже заданного значения. Например, квантиль 0.95, равный 300 мс, означает, что на обработку каждого из 95 % запросов уходило меньше 300 мс.

Квантили могут пригодиться, когда речь заходит о фактическом опыте конечного пользователя. Если веб-браузер пользователя посыпает приложению сразу 20 запросов, то задержка, ощущаемая пользователем, будет определяться самым медленным из них. В этом случае 95-й процентиль зафиксирует эту задержку.

- ✓ 95-й процентиль – это квантиль 0.95. Поскольку Prometheus отдает предпочтение базовым единицам измерения, он всегда использует квантили, потому что отношения предпочтительнее процентов.

Инструментирование для создания гистограмм выполняется точно так же, как и для получения сводных метрик. Метод `observe` позволяет извлекать наблюдения вручную, а диспетчер контекста и декоратор `time` упрощает синхронизацию, как показано в примере 3.11.

Пример 3.11 ♦ LATENCY гистограмма задержек, полученная с помощью декоратора `time`

```
from prometheus_client import Histogram

LATENCY = Histogram('hello_world_latency_seconds',
    'Time for a request Hello World.')

class MyHandler(http.server.BaseHTTPRequestHandler):
    @LATENCY.time()
    def do_GET(self):
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello World")
```

Этот код создаст набор временных рядов с именем `hello_world_latency_seconds_bucket`, представленных наборами счетчиков. Гистограмма имеет набор сегментов, таких как 1 мс, 10 мс и 25 мс, в каждом из которых запоминается количество событий, попавших в соответствующий сегмент. Вычислить кван-

тили на основе сегментов можно с помощью PromQL-функции `histogram_quantile`. Например, квантиль 0.95 (95-й процентиль) можно получить так:

```
histogram_quantile(0.95, rate(hello_world_latency_seconds_bucket[1m]))
```

Функция `rate` необходима здесь по той простой причине, что временные ряды сегментов являются счетчиками.

Сегменты

По умолчанию сегменты гистограмм охватывают диапазоны от 1 мс до 10 с, которые считаются наиболее типичными диапазонами задержек для веб-приложений. Однако есть возможность задать свои протяженности диапазонов при определении метрик, если значения по умолчанию не подходят для вашего варианта использования или требуется явно задать сегменты для квантилей задержки, упомянутых в ваших соглашениях об уровне обслуживания (Service-Level Agreement, SLA). Чтобы помочь вам обнаружить опечатки, предоставленные сегменты должны быть отсортированы:

```
LATENCY = Histogram('hello_world_latency_seconds',
    'Time for a request Hello World.',
    buckets=[0.0001, 0.0002, 0.0005, 0.001, 0.01, 0.1])
```

Если вам нужны линейные или экспоненциальные сегменты, то в Python можно использовать генераторы списков. Клиентские библиотеки для языков, не имеющих эквивалента генераторов списков, могут предусматривать вспомогательные функции для этой цели:

```
buckets=[0.1 * x for x in range(1, 10)] # Линейные
buckets=[0.1 * 2**x for x in range(1, 10)] # Экспоненциальные
```

Кумулятивные гистограммы

Рассматривая гистограмму на странице `/metrics`, вы могли заметить, что сегменты – это не просто счетчики событий, попадающих в эти сегменты. Они также включают количество событий из всех меньших сегментов, вплоть до сегмента `+Inf`, который содержит суммарное количество всех событий. Такие гистограммы называют кумулятивными гистограммами, и по этой причине метка сегмента называется `le`, т. е. «less than or equal to» – меньше или равно.

Кроме того, что сегменты являются счетчиками, гистограммы Prometheus еще и суммируются двумя разными способами.

Причина, почему они реализованы как кумулятивные, заключается в том, что при большом количестве сегментов, когда они превращаются в проблему производительности, есть возможность удалить некоторые лишние сегменты¹ с помощью `metric_relabel_configs` (см. раздел «`metric_relabel_configs`» главы 8), при этом позволяя вычислять квантили. Одно из возможных решений показано в примере 8.27.

¹ Кроме сегмента `+Inf`, который должен присутствовать всегда.

Возможно, вам интересно узнать, сколько сегментов должно быть, чтобы обеспечить достаточную точность. Мы рекомендуем создавать порядка 10 сегментов. Это число может показаться небольшим, но сегменты потребляют ресурсы, потому что каждый сегмент – это дополнительный временной ряд, который необходимо хранить¹. Системы на основе метрик, такие как Prometheus, в принципе не могут обеспечить 100 % точность квантилей. Чтобы получить абсолютную точность, вам придется рассчитать квантили на основе журналов. Но для большинства задач оповещения и отладки той точности, что дает Prometheus, вполне достаточно.

Важно понимать, что сегменты (и метрики вообще) не всегда идеальны, но обычно они дают достаточно информации, чтобы выбрать правильное направление при отладке. Так, например, если Prometheus указывает, что квантиль 0.95 подскочил с 300 до 350 мс, но на самом деле он увеличился с 305 до 355 мс, то эта разница не имеет большого значения. Вы видите, что произошел большой скачок, и направление расследования от увеличения точности не изменится.

i На момент написания этой книги в Prometheus и некоторых клиентских библиотеках появилась новая экспериментальная функция Native Histograms. Она использует динамические сегменты и устраняет большинство проблем «старых» гистограмм.

Ганеш Вернекар (Ganesh Vernekar) и Бьорн Рабенштейн (Björn Rabenstein) рассказали об этой экспериментальной функции на PromCon 2022:

- доклад о новой функции Native Histograms в Prometheus (<https://oreil.ly/uLAxm>);
- доклад о поддержке Native Histograms в PromQL (<https://oreil.ly/8CNJx>).

SLA и квантили

Задержки в SLA часто выражаются как *задержка 95-го процентиля не более 500 мс*. Здесь есть неочевидная ловушка выбора неправильного числа.

Точное вычисление 95-го процентиля – сложная задача, требующая значительных вычислительных ресурсов. Однако подсчитать долю запросов, на обработку которых ушло больше 500 мс, несложно – для этого достаточно двух счетчиков: один для подсчета общего числа запросов и другой – для запросов, обработанных менее чем за 500 мс.

Имея в гистограмме сегмент 500 мс, можно точно рассчитать долю запросов, обрабатывавшихся дольше 500 мс:

```
my_latency_seconds_bucket{le="0.5"}  
/ ignoring(le)  
my_latency_seconds_bucket{le="+Inf"}
```

и тем самым определить, соблюдается ли соглашение об уровне обслуживания (SLA). Остальные сегменты все еще будут давать вам хорошую оценку 95-го процента задержек.

Такие инструменты, как Pyrra (<https://pyrra.dev/>), могут помочь в управлении вашими целями и задачами обслуживания, расчете бюджета ошибок и создании правил записи и оповещения.

¹ Особенno если гистограмма имеет метки.

Квантили накладывают свои ограничения – после их вычисления вы не сможете выполнять дальнейшие математические операции. Например, статистически некорректно складывать, вычитать или усреднять их. Это влияет не только на то, что можно попробовать получить в PromQL, но и на рассуждения о системе во время ее отладки. Внешний интерфейс может сообщить об увеличении задержки в квантиле 0.95, но вызывающая эту задержку серверная реализация может не показать такого увеличения (и даже показывать уменьшение!).

Такое положение вещей может показаться нелогичным, особенно если вас разбудили посреди ночи и сообщили о проблеме. Средние значения, напротив, не страдают этой проблемой; их можно складывать и вычитать¹. Например, если вы видите увеличение задержки на 20 мс во внешнем интерфейсе, обусловленное увеличением задержки в одном из серверных компонентов, то вы увидите соответствующее увеличение задержки примерно на 20 мс и в серверной части. Но с квантилями такая взаимосвязь не гарантируется. Поэтому, хотя квантили хороши для фиксации того, что наблюдает конечный пользователь, отладка с их использованием намного сложнее.

Мы рекомендуем устранять проблемы с задержкой, опираясь в первую очередь на средние значения, а не на квантили. Средние значения соответствуют привычному пониманию, и, определив подсистему, ответственную за увеличение задержки, используя средние значения, вы сможете вернуться к квантилям, если это уместно. С этой целью гистограмма также включает временные ряды `_sum` и `_count`. Как и в случае со сводными данными, вы можете рассчитать среднюю задержку с помощью:

```
rate(hello_world_latency_seconds_sum[1m])
/
rate(hello_world_latency_seconds_count[1m])
```

Модульное тестирование метрик

Модульные тесты помогают своевременно обнаружить нарушения в работе вашего кода, который меняется с течением времени. Модульное тестирование инструментированного кода следует осуществлять так же, как тестирование журналирования. Маловероятно, что вы решите тестировать инструкции журналирования, выполняющиеся на уровне отладки, и точно так же вы не должны тестировать большинство метрик, разбросанных по всему коду.

Обычно в системе журналирования тестируются только инструкции, выполняющие запись в журналы транзакций и иногда в журналы запросов².

¹ Однако было бы неправильно усреднять набор средних значений. Например, если у вас было три события со средним значением 5 и четыре события со средним значением 6, то общее среднее будет не $(5 + 6)/2 = 5.5$, а $(3 \times 5 + 4 \times 6)/(3 + 4) = 5.57$.

² Категории журналов были перечислены в разделе «Журналирование» главы 1.

Точно так же обычно имеет смысл тестировать метрики, являющиеся ключевой частью вашего приложения или библиотеки. Например, если вы пишете библиотеку RPC, то имеет смысл предусмотреть хотя бы несколько простых тестов, чтобы убедиться, что метрики для оценки ключевых запросов, задержек и ошибок работают как должно.

Без тестов некоторые некритические метрики, используемые для отладки, могут перестать работать, и, по нашему опыту, это верно примерно для 5 % таких метрик. Требование, обязывающее тестировать все метрики, усложнит инструментирование, поэтому вместо 20 метрик, из которых 19 пригодны для использования, вы, скорее всего, оставите только пять протестированных метрик, потому что невозможно будет добавлять новые метрики всего двумя строками кода. Однако, когда метрики используются для отладки и глубокого анализа производительности, всегда полезно иметь более широкий набор метрик.

Клиентская библиотека для Python предлагает функцию `get_sample_value`, которая эффективно опрашивает реестр и ищет временные ряды. Вы можете использовать `get_sample_value`, как показано в примере 3.12, для тестирования счетчиков. Поскольку основной интерес представляет увеличение счетчика, тест должен сравнивать значение счетчика до и после события, а не его абсолютное значение. Такой тест будет успешно выполняться, даже если другие тесты тоже вызовут увеличение счетчика.

Пример 3.12 ♦ Модульное тестирование счетчика в Python

```
import unittest
from prometheus_client import Counter, REGISTRY

FOOS = Counter('foos_total', 'The number of foo calls.')

def foo():
    FOOS.inc()

class TestFoo(unittest.TestCase):
    def test_counter_inc(self):
        before = REGISTRY.get_sample_value('foos_total')
        foo()
        after = REGISTRY.get_sample_value('foos_total')
        self.assertEqual(1, after - before)
```

Подход к инструментированию

Теперь, когда вы знаете, как инструментировать свой код (добавлять метрики), перейдем к другому вопросу и поговорим о том, где и в какой степени их применять.

Что инструментировать?

Обычно инструментированию подвергаются службы или библиотеки.

Инструментирование служб

Вообще говоря, существует три типа служб, каждый из которых предлагает свои ключевые метрики: системы онлайн-обслуживания, системы автономного обслуживания и пакетные задания.

Системы онлайн-обслуживания – это системы, от которых человек или другая служба ожидает ответа. К ним относятся веб-серверы и базы данных. Ключевыми метриками, которыми желательно снабдить такие службы, являются частота запросов, величина задержек и частота ошибок. Добавление метрик, показывающих частоту запросов, величину задержек и частоту ошибок, иногда называют методом RED (Rate, Errors, Duration – частота, ошибки и продолжительность). Эти метрики полезны не только на стороне сервера, но и на стороне клиента. Если вы заметили, что клиент наблюдает большую задержку, чем сервер, то, возможно, у вас проблемы с сетью или большой нагрузкой на клиента.

-  Инструментируя метрики продолжительности, старайтесь не исключать сбои. Если метрика будет сообщать только продолжительность обработки успешных запросов, то вы не заметите увеличение задержек, вызванное множеством медленных запросов, завершившихся неудачей.

Системы автономного обслуживания не вынуждают никого ждать ответа. Обычно они группируют задания и передают их по этапам в конвейере с очередями между ними. Примером системы автономного обслуживания может служить система обработки журналов. Каждый этап должен генерировать свои метрики, сообщающие количество заданий в очереди, объем незавершенной работы, скорость обработки элементов и частоту возникающих ошибок. Эти метрики также известны как метод USE (Utilization, Saturation, Errors – потребление, насыщение, ошибки). Потребление показывает уровень нагрузки на службу, насыщение – это объем работы в очереди, а ошибки не требуют пояснений. Если вы объединяете задания в пакеты, то полезно иметь метрики как для пакетов, так и для отдельных заданий.

Пакетные задания – это третий тип служб. Они похожи на автономные системы обслуживания. Однако пакетные задания выполняются по регулярному расписанию, в то время как системы автономного обслуживания работают непрерывно. Поскольку пакетные задания выполняются не всегда, сбор метрик из них работает не слишком хорошо, поэтому при работе с ними применяются такие методы, как Pushgateway (обсуждается в разделе «Pushgateway» главы 4) и сборщик текстовых файлов Node Exporter (обсуждается в разделе «Сборщик текстовых файлов» главы 7). Завершив обработку пакетного задания, необходимо зафиксировать, сколько времени потребовалось для его выполнения, сколько времени занял каждый этап и время последнего успешного выполнения задания. Вы можете добавить оповещение, если за последнее время

ни одно задание не было выполнено успешно, что позволит вам допустить возможность сбоев при выполнении отдельных пакетных заданий.

Идемпотентность пакетных заданий

Идемпотентность – это свойство, гарантирующее получение одного и того же результата от операции или функции независимо от того, сколько раз она запускается. Это полезное свойство для пакетных заданий, поскольку в случае неудачи достаточно просто повторить попытку, и можно не сильно беспокоиться об отдельных сбоях. Чтобы обеспечить идемпотентность, следует избегать передачи конкретных элементов (например, данных за предыдущий день), которые должно обработать пакетное задание. Вместо этого пакетное задание должно само определять, с какого места начать.

Такой подход дает дополнительное преимущество – возможность многократного запуска пакетных заданий. Например, вы можете запускать ежедневное пакетное задание несколько раз в день, чтобы даже в случае временного сбоя задание могло выполниться при следующем запуске. Пороговые количества ошибок для рассылки предупреждений можно соответственно увеличить, тогда вам придется реже вмешиваться вручную.

Инструментирование библиотек

Службы – это то, что находится на верхнем уровне. Каждая из ваших служб использует какие-то библиотеки, которые можно рассматривать как мини-службы. Большинство из них будут играть роль подсистем онлайн-обслуживания, т. е. обрабатывать вызовы синхронно, и, соответственно, вы сможете пользоваться всеми преимуществами наличия одних и тех же метрик с частотой запросов, величиной задержек и частотой ошибок. Для мониторинга кеша вам понадобятся эти метрики как для общего числа обращений к кешу, так и для промахов кеша, информацию о которых необходимо вычислить или запросить у сервера.



Имея метрики с количеством отказов и общим количеством запросов, легко рассчитать коэффициент отказов простым делением. С успехами и неудачами дело обстоит сложнее¹, так как сначала нужно подсчитать итог.

Точно так же для кешей желательно иметь количество попаданий и общее количество запросов или количество промахов и общее количество запросов. Ничуть не помешает иметь все три метрики, показывающие количества попаданий, промахов и всего запросов.

Не ленитесь добавлять метрики для любых возникающих ошибок и везде, где есть журналирование. Учитывая большой объем журналов отладки, их хранение может быть ограничено несколькими днями, но даже в этом случае метрики помогут вам получить представление о частоте появления в журнале той или иной строки.

¹ Не пытайтесь делить неудачи на успехи.

Пулы рабочих потоков должны инструментироваться подобно системам автономного обслуживания. Желательно снабжать их метриками, сообщающими размер очереди, количество активных потоков, любые ограничения на количество потоков и обнаруженные ошибки.

Задачи фонового обслуживания, которые запускаются не чаще нескольких раз в час, фактически являются пакетными заданиями, и они должны снабжаться аналогичными метриками.

Как широко использовать инструментирование?

Система Prometheus чрезвычайно эффективна, однако и она не может обрабатывать слишком большое количество метрик. В какой-то момент наладные расходы начнут перевешивать преимущества инструментирования.

Однако в большинстве практических случаев об этом можно не беспокоиться. Допустим, у вас есть сервер Prometheus, способный обрабатывать 10 млн метрик¹ и 1000 экземпляров приложений. Одна новая метрика в каждом из этих экземпляров будет использовать 0.01 % ресурсов, что делает ее фактически бесплатной². Это означает, что вы можете добавлять новые метрики везде, где это принесет пользу.

Единственное место, где следует проявлять осторожность, – промышленное окружение. Если вы по привычке добавляете метрику, измеряющую продолжительность выполнения каждой функции, то суммарное потребление ресурсов может оказаться огромным (в конце концов, это классическое профилирование). Если у вас есть метрики, разбитые по типам запросов и URL³, то при большом количестве возможных комбинаций потребление ресурсов тоже может значительно вырасти. Сегменты гистограмм тоже требуют наладных расходов. Метрика с кардинальностью под сотню в каждом экземпляре будет потреблять 1 % ресурсов вашего сервера Prometheus, и выгоды от такой метрики становятся не так очевидны. Мы поговорим об этом в разделе «Кардинальность» главы 5.

Обычно 10 самых больших метрик в экземпляре Prometheus потребляют более половины его ресурсов. Оптимизировав потребление ресурсов десятью самыми важными метриками, вы получите наибольшую отдачу.

Как правило, простая служба, такая как кеш, может иметь под сотню метрик, а сложная и тщательно инструментированная служба – под тысячу.

Какие имена выбирать для метрик?

Именование метрик – это скорее искусство, чем наука. Есть несколько простых правил, которые помогут избежать наиболее очевидных ошибок, а также общие рекомендации по выбору имен для метрик.

¹ Это примерный предел производительности Prometheus 1.x.

² Этот расчет действителен для счетчиков и датчиков без меток.

³ В главе 5 рассматриваются метки – мощная особенность Prometheus, делающая это возможным.



Переименование метрик может затруднить отслеживание и анализ данных в будущем, потому что нарушит работу существующих запросов и дашбордов. Для поддержания целостности ваших данных при переименовании метрик может потребоваться выполнить сложные манипуляции, такие как редактирование запросов PromQL.

В общем случае для имен метрик рекомендуется использовать такую структуру: `библиотека_имя_модуль_суффикс`.

Символы

Имена метрик Prometheus должны начинаться с буквы, за которой может следовать любое количество букв, цифр и знаков подчеркивания.

Для проверки действительности имен метрик можно использовать регулярное выражение `[a-zA-Z_-][a-zA-Z0-9_-]*`, но есть несколько вполне допустимых имен, которых следует избегать. Вы не должны использовать двоеточия в именах, поскольку они зарезервированы для определения правил записи, как описано в разделе «Именование правил записи» главы 17 на стр. 294. Символы подчеркивания в начале имен метрик зарезервированы для внутреннего использования Prometheus.

Змеиная нотация

В Prometheus действует соглашение об использовании змеиной нотации для имен метрик; т. е. компоненты имени должны записываться строчными буквами и разделяться символом подчеркивания.

Суффиксы в именах метрик

Суффиксы `_total`, `_count`, `_sum` и `_bucket` принято использовать в именах метрик-счетчиков, сводных метрик и гистограмм. Имена счетчиков всегда должны оканчиваться суффиксом `_total`, но не добавляйте перечисленные суффиксы в имена датчиков, чтобы избежать путаницы.

Единицы измерения

Отдавайте предпочтение базовым единицам измерения, таким как секунды, байты и отношения¹. Это связано с тем, что Prometheus использует секунды в таких функциях, как `time`, и избегает таких уродливых единиц, как киломикросекунды.

Явное использование единиц измерения поможет избежать путаницы в отношении того, является ли эта конкретная метрика секундами или миллисекундами², поэтому всегда включайте единицы измерения в имена метрик. Например, для счетчика, измеряющего время в секундах, можно выбрать имя `mymetric_seconds_total`.

¹ Величина отношений, как правило, изменяется в диапазоне от 0 до 1, а проценты – от 0 до 100.

² В течение некоторого периода времени сама система Prometheus использовала секунды, миллисекунды, микросекунды и наносекунды для разных метрик.

Однако для некоторых метрик не существует очевидной единицы измерения, поэтому не переживайте, если в имени метрики не получится отразить ее единицу измерения. Страйтесь избегать использовать `count` в роли единицы, так как, помимо конфликта с именами сводных метрик и гистограмм, это слово не несет большой смысловой нагрузки, потому что большинство метрик являются счетчиками чего-либо. То же относится и к `total`.

Имя

Основа имени метрики – это..., гм..., имя. Имя метрики должно давать хорошее представление о ее назначении. Имя `requests` не очень информативно, имя `http_requests` лучше, а `http_requests_authenticated` – еще лучше. Описание метрики может содержать более подробное объяснение, но часто пользователь видит только имя метрики.

Как было показано в предыдущих примерах, имя метрики может состоять из нескольких компонентов, разделенных подчеркиванием. Страйтесь использовать одинаковый префикс для родственных метрик, чтобы было легче понять их взаимосвязь. Имена `queue_size` и `queue_limit` лучше подчеркивают родственность, чем `size_queue` и `limit_queue`. Можно даже использовать имена `items` и `items_limit`. Обычно компоненты имен следуют в порядке от менее конкретных к более конкретным.

Не добавляйте в имена метрик слова, которые играют роль меток (метки описываются в главе 5). Реализуя прямое инструментирование, никогда не генерируйте метрики или их имена процедурно.

 Избегайте включения меток в имена метрик, потому что это может породить ошибки при агрегировании меток с помощью PromQL.

Библиотека

Поскольку имена метрик фактически определяются в глобальном пространстве имен, важно не только избегать конфликтов между библиотеками, но и указывать, откуда берется метрика. Имя метрики в конечном счете указывает на конкретную строку кода в конкретном файле в конкретной библиотеке. Библиотека может быть стереотипной библиотекой, подключаемой как зависимость, играющей роль подсистемы в приложении или даже реализующей основную функциональность этого приложения.

Компонент имени, определяющий библиотеку, должен быть достаточно однозначным, чтобы избежать путаницы, и совершенно необязательно включать полные имена организаций и пути в системе управления версиями. Найдите баланс между краткостью и полнотой квалификации.

Например, `Cassandra` – широко известное приложение, поэтому вполне уместно использовать только слово `cassandra` в роли компонента имени, определяющего библиотеку. С другой стороны, использование слова `db` для обозначения принадлежности к внутренней библиотеке управления пулом соединений с базой данных было бы неразумно из-за большого разнообразия библиотек баз данных и пулов соединений. Более того, в вашем приложении

может использоваться несколько таких библиотек. Намного предпочтительнее выглядят такие компоненты, как `robustreception_db_pool` и `rp_db_pool`.

Некоторые имена библиотек уже хорошо известны. Библиотека `process` предоставляет метрики уровня процесса, такие как потребление процессорного времени и памяти, и стандартизирована. По этой причине не следует создавать свои метрики с этим префиксом. Клиентские библиотеки также предоставляют метрики, относящиеся к среде выполнения. Метрики Python используют префикс `python`, метрики виртуальной машины Java (JVM) используют префикс `jvm`, а Go – префикс `go`.

Объединение всех вышеперечисленных шагов дает имена метрик, такие как `go_memstats_heap_inuse_bytes`. Эта метрика определена в библиотеке `go_memstats` и отражает статистику потребления памяти в среде выполнения Go. `heap_inuse` подсказывает, что метрика связана с объемом используемой памяти в куче, а `bytes` – что этот объем измеряется в байтах. Только по имени метрики уже можно сказать, что эта метрика сообщает объем памяти в куче¹, которую Go использует в настоящее время. Смысл метрики не всегда может быть очевидным из ее названия, но к этому нужно стремиться.



Необязательно добавлять в имена всех метрик префикс с именем приложения. `process_cpu_seconds_total` – это `process_cpu_seconds_total`, независимо от того, какое приложение предоставляет эту метрику. Отличить метрики из разных приложений можно с помощью целевых меток, см. раздел «Метки целей» главы 8.

Теперь, закончив инструментирование приложения, давайте посмотрим, как передать эти метрики в Prometheus.

¹ Куча – это область динамически выделяемой памяти для вашего процесса. Она используется для выделения памяти такими функциями, как `malloc`.

Глава 4

Экспортирование метрик

В главе 3 мы в основном сосредоточились на добавлении метрик в код. Но любые метрики бесполезны, если они не попадают в систему мониторинга. Процесс передачи метрик в Prometheus известен как **экспортирование**.

Экспортирование метрик в Prometheus осуществляется через HTTP. Обычно метрики экспортируются через URL `/metrics`, а запросы обрабатываются клиентской библиотекой. Prometheus поддерживает два удобочитаемых текстовых формата: Prometheus и OpenMetrics. Также есть возможность определить свой формат, и в этом случае проще взять за основу текстовый формат Prometheus как менее строгий. Этот шаг может понадобиться, если нет подходящей библиотеки для вашего языка, но мы все же рекомендуем стараться использовать готовые библиотеки, так как они заботятся обо всех мелких деталях, таких как правильное экранирование. Большинство библиотек также предоставляют возможность создавать метрики, используя оба текстовых формата, OpenMetrics и Prometheus.

Экспортирование обычно выполняется либо в функции `main`, либо в другой функции верхнего уровня, и ее необходимо настроить только один раз для каждого приложения.

Метрики обычно регистрируются в *реестре по умолчанию* в процессе создания. Если одна из библиотек, от которых вы зависите, имеет готовые метрики Prometheus, то они будут помещены в реестр по умолчанию, и вы получите дополнительные преимущества, не приложив никаких усилий. Некоторые пользователи предпочитают явно задавать реестр в функции `main`, поэтому вам придется полагаться на то, что каждая библиотека между функцией `main` вашего приложения и метриками Prometheus поддерживает инstrumentирование. Это предполагает, что каждая библиотека в цепочке зависимостей сама позаботится об инструментировании своего кода и соглашается с выбором библиотек инструментирования.

Это решение позволяет инструментировать метрики Prometheus вообще без экспортования¹. В этом случае, помимо мизерных затрат, инструментирование никак не повлияет на ваше приложение. Если вы пишете библиотеку, то можете добавить метрики Prometheus для своих пользователей, чтобы избавить их от дополнительных усилий. Для лучшей поддержки этого варианта использования клиентские библиотеки стараются свести к минимуму свои зависимости.

Давайте посмотрим, как осуществляется экспортование в некоторых популярных клиентских библиотеках. Здесь мы предполагаем, что вы знаете, как установить клиентские библиотеки и любые другие необходимые зависимости.

Python

Вы уже видели функцию `start_http_server` в главе 3. Она запускает фоновый поток с HTTP-сервером, который обслуживает только метрики Prometheus, как показано ниже:

```
from prometheus_client import start_http_server

if __name__ == '__main__':
    start_http_server(8000)
    // Далее следует ваш код.
```

`start_http_server` очень удобна. Но может так сложиться, что в вашем приложении уже есть HTTP-сервер, и вам хотелось бы, чтобы метрики обслуживал он.

В Python это можно сделать различными способами, в зависимости от используемых фреймворков.

WSGI

Интерфейс шлюза веб-сервера (Web Server Gateway Interface, WSGI; <https://oreil.ly/5B1tz>) – это стандарт для веб-приложений на Python. Клиент на Python предоставляет приложение WSGI, которое можно использовать с существующим кодом WSGI. В примере 4.1 функция `my_app` делегирует `metrics_app`, если запрашивается путь `/metrics`; в противном случае она выполняет свою обычную логику. Объединив приложения WSGI в цепочку, можно добавить промежуточное ПО, например для аутентификации, обычно не поддерживающей клиентскими библиотеками.

¹ Отсутствие экспортования означает, что метрики не будут анализироваться сервером Prometheus.

Пример 4.1 ♦ Экспортирования в Python с использованием WSGI

```
from prometheus_client import make_wsgi_app
from wsgiref.simple_server import make_server

metrics_app = make_wsgi_app()

def my_app(environ, start_fn):
    if environ['PATH_INFO'] == '/metrics':
        return metrics_app(environ, start_fn)
    start_fn('200 OK', [])
    return [b'Hello World']

if __name__ == '__main__':
    httpd = make_server('', 8000, my_app)
    httpd.serve_forever()
```

Путь /metrics обязателен?

/metrics – это HTTP-путь по умолчанию, по которому обслуживаются метрики Prometheus, но это всего лишь соглашение, поэтому вы можете экспортировать свои метрики по другому пути. Например, если путь */metrics* уже используется в вашем приложении или вы хотите разместить административные конечные точки с префиксом */admin/*.

На такую конечную точку, даже если она находится на другом пути, по-прежнему принято ссылаться как на путь */metrics*.

Twisted

Twisted (<https://twisted.org/>) – сетевой движок Python, управляемый событиями. Он поддерживает WSGI, что позволяет подключить `make_wsgi_app`, как показано в примере 4.2.

Пример 4.2 ♦ Экспортирования в Python с использованием Twisted

```
from prometheus_client import make_wsgi_app
from twisted.web.server import Site
from twisted.web.wsgi import WSGIResource
from twisted.web.resource import Resource
from twisted.internet import reactor

metrics_resource = WSGIResource(
    reactor, reactor.getThreadPool(), make_wsgi_app())

class HelloWorld(Resource):
    isLeaf = False
    def render_GET(self, request):
        return b"Hello World"

root = HelloWorld()
```

```
root.putChild(b'metrics', metrics_resource)
reactor.listenTCP(8000, Site(root))
reactor.run()
```

Многозадачность с Gunicorn

Prometheus предполагает, что приложения, за которыми он следит, являются долгоживущими и многопоточными. Но это неверно для некоторых окружений выполнения, таких как CPython¹. Интерпретатор CPython фактически ограничен одним ядром процессора из-за глобальной блокировки интерпретатора (Global Interpreter Lock, GIL). Чтобы обойти это ограничение, некоторые пользователи распределяют рабочую нагрузку между несколькими процессами с помощью такого инструмента, как Gunicorn (<https://gunicorn.org/>).

Если использовать клиентскую библиотеку для Python обычным образом, то каждый рабочий процесс экспортировал бы свои собственные метрики. Каждый раз, сканируя приложение, Prometheus случайным образом будет получать метрики только от одного из рабочих процессов, содержащих лишь часть общей информации, и сталкиваться с такими проблемами, как счетчики, которые на первый взгляд считают в обратную сторону. Кроме того, рабочие процессы могут быть относительно недолговечными.

Решение, предлагаемое клиентом Python, состоит в том, чтобы оставить за каждым рабочим процессом экспортацию их собственных метрик как есть, но во время опроса объединять все метрики всех рабочих процессов, чтобы обеспечить семантику, которую вы получили бы от многопоточного приложения. Это решение имеет некоторые ограничения: метрики `process` и пользовательские сборщики будут недоступны, а метод `Pushgateway` использовать нельзя².

При использовании Gunicorn вы должны сообщить клиентской библиотеке, когда рабочий процесс завершается³. Эта настройка производится в файле конфигурации, подобном тому, что показан в примере 4.3.

Пример 4.3 ♦ Конфигурационный файл config.py для Gunicorn, обрабатывающий завершение рабочих процессов

```
from prometheus_client import multiprocess

def child_exit(server, worker):
    multiprocess.mark_process_dead(worker.pid)
```

¹ CPython – это официальное название стандартной реализации Python. Не путайте его с реализацией Cython, с помощью которой можно писать расширения на С для Python.

² Pushgateway не подходит для этого варианта использования, поэтому на практике это не проблема.

³ В версию Gunicorn 19.7, вышедшую в марте 2017 года, для этой цели была добавлена функция `child_exit`.

Вам также понадобится приложение для обслуживания метрик. Gunicorn использует WSGI, поэтому вы можете использовать `make_wsgi_app`. Вы также должны создать *свой реестр*, содержащий только экземпляр `MultiProcessCollector`, выполняющий экспортацию, чтобы он не включал метрики от множества процессов и метрики из локального реестра по умолчанию (пример 4.4).

Пример 4.4 ♦ Приложение Gunicorn в `app.py`

```
from prometheus_client import multiprocess, make_wsgi_app, CollectorRegistry
from prometheus_client import Counter, Gauge

REQUESTS = Counter("http_requests_total", "HTTP requests")
IN_PROGRESS = Gauge("http_requests_inprogress", "Inprogress HTTP requests",
    multiprocess_mode='livesum')

@IN_PROGRESS.track_inprogress()
def app(environ, start_fn):
    REQUESTS.inc()
    if environ['PATH_INFO'] == '/metrics':
        registry = CollectorRegistry()
        multiprocess.MultiProcessCollector(registry)
        metrics_app = make_wsgi_app(registry)
        return metrics_app(environ, start_fn)
    start_fn('200 OK', [])
    return [b'Hello World']
```

Как показано в примере 4.4, счетчики работают нормально, так же как сводные метрики и гистограммы. Для датчиков имеется дополнительная необязательная конфигурация в виде параметра `multiprocess_mode`. Вы можете настроить датчик в зависимости от того, как собираетесь его использовать.

`all`

Значение по умолчанию. В этом случае временной ряд возвращается для каждого процесса независимо от того, активен он или остановлен. Это позволяет вам агрегировать временные ряды в PromQL по своему усмотрению. Они будут различаться меткой `pid`.

`liveall`

Возвращает временной ряд для каждого активного процесса.

`livesum`

Возвращает один временной ряд с суммами значений из всех активных процессов. Это значение можно использовать, например, для подсчета незавершенных запросов или потребления ресурсов всеми процессами. Процесс может прервать свое выполнение с ненулевым значением, поэтому прерванные процессы исключаются.

`max`

Возвращает один временной ряд, содержащий максимальные значения метрик среди всех активных или остановленных процессов. Эта возможность может пригодиться для оценки ситуации в последний момент перед происшествием, например, чтобы определить, какой запрос обрабатывался в процессе, который сейчас остановлен.

min

Возвращает один временной ряд, содержащий минимальные значения метрик среди всех активных или остановленных процессов.

Перед запуском Gunicorn необходимо выполнить некоторые настройки, как показано в примере 4.5. В частности, установить переменную окружения с именем `PROMETHEUS_MULTIPROC_DIR`. Она должна ссылаться на пустой каталог, который клиентская библиотека будет использовать для отслеживания метрик. Этот каталог всегда должен очищаться перед запуском приложения, чтобы обеспечить поддержку любых возможных изменений в инструментировании.

Пример 4.5 ♦ Подготовка окружения перед запуском Gunicorn с двумя рабочими процессами

```
hostname $ export PROMETHEUS_MULTIPROC_DIR=$PWD/multiproc
hostname $ rm -rf $PROMETHEUS_MULTIPROC_DIR
hostname $ mkdir -p $PROMETHEUS_MULTIPROC_DIR
hostname $ gunicorn -w 2 -c config.py app:app
[2018-01-07 19:05:30 +0000] [9634] [INFO] Starting gunicorn 19.7.1
[2018-01-07 19:05:30 +0000] [9634] [INFO] Listening at: http://127.0.0.1:8000 (9634)
[2018-01-07 19:05:30 +0000] [9634] [INFO] Using worker: sync
[2018-01-07 19:05:30 +0000] [9639] [INFO] Booting worker with pid: 9639
[2018-01-07 19:05:30 +0000] [9640] [INFO] Booting worker with pid: 9640
```

Запросив путь `/metrics`, вы увидите две метрики, определенные в приложении, но `python_info` и `process_metrics` среди них не будет.

Многозадачный режим за кулисами

Производительность жизненно важна для клиентских библиотек. Архитектуры, в которых рабочие процессы отправляют UDP-пакеты и вообще используют сеть, не могут обеспечить высокой производительности из-за больших накладных расходов на системные вызовы. Нужно что-то иное, такое же быстрое, как обычное инструментирование, действующее в локальной памяти процесса, но доступное для других процессов.

Общепринятый подход заключается в использовании `mmap`. У каждого процесса есть свой набор `mmap`-файлов, в которых он хранит свои метрики. На этапе опроса клиентская библиотека читает все файлы и объединяет метрики. Операции записи значений в файлы средствами инструментирования и чтения из файлов во время опроса не требуется блокировать друг от друга, чтобы гарантировать получение целостных данных, а при добавлении нового временного ряда используется двухэтапная запись.

Счетчики (включая сводные метрики и гистограммы) не должны сбрасываться, поэтому файлы, относящиеся к счетчикам, сохраняются после завершения рабочего процесса. Необходимость подобного решения для датчика зависит от особенностей его использования. Например, метрика, отражающая количество незавершенных запросов, имеет смысл только для активных процессов, тогда как информация о последнем обрабатывавшемся запросе нужна не только для активных, но и для оставшихся процессов. Такое поведение можно настроить отдельно для каждого датчика.

 Каждый процесс создает в `prometheus_multiproc_dir` несколько файлов, которые необходимо прочитать во время опроса. Если ваши рабочие процессы часто останавливаются и запускаются, то это может замедлить опрос, если в каталоге появятся тысячи файлов.

Удалять отдельные файлы небезопасно, так как это может привести к неправильному сбросу счетчиков, но вы можете попытаться смягчить проблему (увеличив или удалив ограничение на количество запросов, обрабатываемых рабочими процессами перед завершением¹) либо регулярно перезапускать приложение и удалять накопившиеся файлы.

Описанные шаги применяются при использовании Gunicorn. Но тот же подход можно использовать и с механизмами поддержки многозадачности в Python, такими как модуль `multiprocessing`.

Поддержка OpenMetrics

Клиентская библиотека для Python изначально экспортирует метрики в формате OpenMetrics. Prometheus всегда предпочитает этот формат, когда он доступен. Чтобы сообщить, что он поддерживает анализ метрик в формате OpenMetrics, Prometheus использует HTTP-заголовок `Accept`. Вы можете имитировать это поведение с помощью параметра `-H` утилиты curl:

```
curl -v -H 'Accept: application/openmetrics-text; version=1.0.0;
charset=utf-8' http://127.0.0.1:8000/metrics
```

Go

В Go стандартным интерфейсом для реализации обработчиков HTTP служит `http.Handler`, а для клиентской библиотеки на Go этот интерфейс реализован как `promhttp.Handler`. Чтобы увидеть, как он работает, поместите код из примера 4.6 в файл с именем `example.go`.

Пример 4.6 ♦ Простая программа на Go, демонстрирующая порядок инструментирования и экспортирования метрик

```
package main

import (
    "log"
    "net/http"

    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/promauto"
    "github.com/prometheus/client_golang/promhttp"
)

var (
```

¹ Флаг `--max-requests` в Gunicorn – один из примеров такого ограничения.

```

    requests = promauto.NewCounter(
        prometheus.CounterOpts{
            Name: "hello_worlds_total",
            Help: "Hello Worlds requested.",
        })
    )

func handler(w http.ResponseWriter, r *http.Request) {
    requests.Inc()
    w.Write([]byte("Hello World"))
}

func main() {
    http.HandleFunc("/", handler)
    http.Handle("/metrics", promhttp.Handler())
    log.Fatal(http.ListenAndServe(":8000", nil))
}

```

Получить зависимости и запустить этот код можно обычным способом:

```

hostname $ go get -d -u github.com/prometheus/client_golang/prometheus
hostname $ go run example.go

```

В этом примере используется модуль `rgomauto`, который автоматически зарегистрирует вашу метрику в реестре по умолчанию. Если этого не требуется, то счетчик можно создать вызовом `rgometheus.NewCounter`, а затем использовать `MustRegister` в функции `init`:

```

func init() {
    prometheus.MustRegister(requests)
}

```

Однако, применяя такое решение, легко допустить ошибку: создать и использовать метрику, но забыть вызвать `MustRegister`.

Java

Клиентская библиотека для Java также известна как *simpleclient*. Она заменила *оригинальную клиентскую библиотеку*, разработанную до того, как были выработаны современные приемы и методы разработки клиентских библиотек. Клиентскую библиотеку для Java следует использовать для инструментирования кода на любом языке, выполняющегося под управлением виртуальной машины Java (Java Virtual Machine, JVM).

HTTPServer

Подобно `start_http_server` в Python, класс `HTTPServer` в клиенте для Java позволяет быстро начать использовать метрики (пример 4.7).

Пример 4.7 ♦ Простая программа на Java, демонстрирующая порядок инструментирования и экспортирования метрик

```
import io.prometheus.client.Counter;
import io.prometheus.client.hotspot.DefaultExports;
import io.prometheus.client.exporter.HTTPServer;

public class Example {
    private static final Counter myCounter = Counter.build()
        .name("my_counter_total")
        .help("An example counter.").register();

    public static void main(String[] args) throws Exception {
        DefaultExports.initialize();
        HTTPServer server = new HTTPServer(8000);
        while (true) {
            myCounter.inc();
            Thread.sleep(1000);
        }
    }
}
```

Как правило, метрики в Java определяются в виде статических полей классов, чтобы они регистрировались только один раз.

Вызов `DefaultExports.initialize` необходим, чтобы сделать доступными различные метрики `process` и `jvm`. Обычно этот метод достаточно вызывать только один раз в Java-приложении, например в функции `main`. Однако метод `DefaultExports.initialize` идемпотентный и потокобезопасный, поэтому дополнительные вызовы не принесут вреда.

Чтобы запустить код из примера 4.7, вам понадобятся зависимости `simple-client`. Для тех, кто использует Maven, в примере 4.8 показано, как должны выглядеть определения зависимостей в файле `pom.xml`.

Пример 4.8 ♦ Зависимости в `pom.xml` для примера 4.7

```
<dependencies>
    <dependency>
        <groupId>io.prometheus</groupId>
        <artifactId>simpleclient</artifactId>
        <version>0.16.0</version>
    </dependency>
    <dependency>
        <groupId>io.prometheus</groupId>
        <artifactId>simpleclient_hotspot</artifactId>
        <version>0.16.0</version>
    </dependency>
    <dependency>
        <groupId>io.prometheus</groupId>
        <artifactId>simpleclient_httpserver</artifactId>
        <version>0.16.0</version>
    </dependency>
</dependencies>
```

Сервлеты

Многие платформы Java и JVM поддерживают использование подклассов *HttpServlet* в своих HTTP-серверах и промежуточном программном обеспечении. Jetty – один из таких серверов, и вы можете увидеть в примере 4.9, как использовать класс *MetricsServlet* из клиентской библиотеки для Java.

Пример 4.9 ♦ Программа на Java, демонстрирующая использование *MetricsServlet* и Jetty

```
import io.prometheus.client.Counter;
import io.prometheus.client.exporter.MetricsServlet;
import io.prometheus.client.hotspot.DefaultExports;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.servlet.ServletContextHandler;
import org.eclipse.jetty.servlet.ServletHolder;
import java.io.IOException;

public class Example {
    static class ExampleServlet extends HttpServlet {
        private static final Counter requests = Counter.build()
            .name("hello_worlds_total")
            .help("Hello Worlds requested.").register();

        @Override
        protected void doGet(final HttpServletRequest req,
            final HttpServletResponse resp)
            throws ServletException, IOException {
            requests.inc();
            resp.getWriter().println("Hello World");
        }
    }

    public static void main(String[] args) throws Exception {
        DefaultExports.initialize();

        Server server = new Server(8000);
        ServletContextHandler context = new ServletContextHandler();
        context.setContextPath("/");
        server.setHandler(context);
        context.addServlet(new ServletHolder(new ExampleServlet()), "/");
        context.addServlet(new ServletHolder(new MetricsServlet()), "/metrics");

        server.start();
        server.join();
    }
}
```

Вам также потребуется добавить Java-клиента в зависимости. В примере 4.10 показано, как это сделать при использовании Maven.

Пример 4.10 ♦ Зависимости в pom.xml для примера 4.9

```
<dependencies>
  <dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>simpleclient</artifactId>
    <version>0.16.0</version>
  </dependency>
  <dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>simpleclient_hotspot</artifactId>
    <version>0.16.0</version>
  </dependency>
  <dependency>
    <groupId>io.prometheus</groupId>
    <artifactId>simpleclient_servlet</artifactId>
    <version>0.16.0</version>
  </dependency>
  <dependency>
    <groupId>org.eclipse.jetty</groupId>
    <artifactId>jetty-servlet</artifactId>
    <version>11.0.11</version>
  </dependency>
</dependencies>
```

Pushgateway

Пакетные задания обычно выполняются по регулярному расписанию, например ежечасно или ежедневно. Они запускаются, выполняют какую-то работу и завершаются. Поскольку они работают не постоянно, Prometheus может не успеть извлечь из них метрики¹. Решить эту проблему поможет Pushgateway.

Pushgateway² – это кеш метрик для пакетных заданий на уровне служб. Его архитектура показана на рис. 4.1. Для каждого пакетного задания он сохраняет последнюю порцию метрик, которую вы передали. Этот прием используется, когда пакетные задания отправляют свои метрики непосредственно перед завершением. Prometheus выбирает эти метрики из кеша Pushgateway, после чего появляется возможность проанализировать их, вывести на графики или разослать оповещения. Обычно Pushgateway запускается на одном сервере с Prometheus.

¹ Хотя для пакетных заданий, выполняющихся больше нескольких минут, имеет смысл оставить сбор метрик в обычном режиме через HTTP, чтобы избежать возможных проблем с производительностью.

² В неформальном контексте его также часто называют *pgw*.

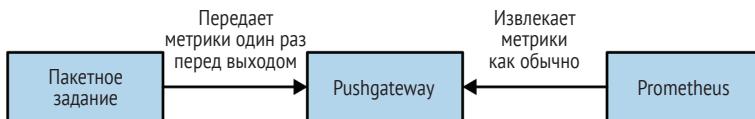


Рис. 4.1 ♦ Архитектура Pushgateway

Пакетное задание уровня службы – это задание, к которому на самом деле не применяется метка `instance`. То есть это все службы, не привязанные к конкретному компьютеру или экземпляру процесса¹. Если вам не важно, где выполняется пакетное задание, но важен сам факт его выполнения (даже если в настоящее время оно настроено для запуска через cron на одной машине), то это пакетное задание уровня службы. Примеры включают пакетное задание для каждого центра обработки данных для проверки наличия неисправных машин или задание, которое выполняет сборку мусора для всей службы.

i Pushgateway – это не способ перевести Prometheus из режима `pull` (режим активного опроса) в режим `push` (режим пассивного приема). Если, например, между двумя циклами опроса со стороны Prometheus пакетное задание успело несколько раз отправить свои метрики, то Pushgateway вернет только последнюю отправку для этого пакетного задания. Подробнее этот вопрос обсуждается в разделе «Сети и аутентификация» главы 21.

Скачать Pushgateway можно со страницы загрузки на сайте проекта Prometheus (<https://oreil.ly/hoXpK>). Это экспортер, который по умолчанию прослушивает порт 9091, и Prometheus должен быть настроен на извлечение метрик из него. Также необходимо указать параметр `honor_labels: true` в конфигурационном файле с настройками опроса, как показано в примере 4.11. Это обусловлено тем, что метрики, которые отправляются в Pushgateway, не должны иметь метки `instance`, и точно так же собственная целевая метка `instance` кеша Pushgateway не должна попасть в метрики, когда Prometheus попытается извлечь их². Подробнее параметр `honor_labels` обсуждается в разделе «Конфликты меток и `honor_labels`» главы 8.

Пример 4.11 ♦ Настройки опроса локального кеша Pushgateway в `prometheus.yml`

```

scrape_configs:
  - job_name: pushgateway
    honor_labels: true
    static_configs:
      - targets:
          - localhost:9091
  
```

¹ Для пакетных заданий, таких как резервное копирование базы данных, привязанных к жизненному циклу машины, лучше использовать сборщик текстовых файлов Node Exporter. Это обсуждается в разделе «Сборщик текстовых файлов» главы 7.

² Pushgateway явно экспортирует пустые метки `instance` для метрик, не имеющих ее. В сочетании с `honor_labels: true` это приводит к тому, что Prometheus не добавляет метку `instance` к этим метрикам. Обычно пустые и отсутствующие метки в Prometheus – это одно и то же, но данный случай является исключением.

Отправлять метрики в Pushgateway можно с помощью клиентских библиотек. В примере 4.12 показано, как это сделать в пакетном задании на Python. Пользовательский реестр создается так, что отправляются только определенные метрики. Продолжительность выполнения пакетного задания передается всегда¹, а время его окончания отправляется, только если задание выполнено успешно.

Отправить метрики в Pushgateway можно тремя разными способами. В Python для этого имеются функции `push_to_gateway`, `pushadd_to_gateway` и `delete_from_gateway`.

`push_to_gateway`

Все существующие метрики, полученные из этого задания, удаляются, и добавляются отправленные метрики. Отправка осуществляется HTTP-методом PUT.

`pushadd_to_gateway`

Передаваемые метрики переопределяют существующие метрики с теми же именами для этого задания. Любые метрики с другими именами, существовавшие ранее, остаются без изменений. Отправка осуществляется HTTP-методом POST.

`delete_from_gateway`

Метрики для этого задания удаляются. Удаление осуществляется HTTP-методом DELETE.

Поскольку в примере 4.12 используется функция `pushadd_to_gateway`, значение `my_job_duration_seconds` будет заменяться всегда, а значение `my_job_last_success_seconds#` будет заменяться, только если не возникло исключений.

Пример 4.12 ♦ Инструментирование пакетного задания и отправка его метрик в Pushgateway

```
from prometheus_client import CollectorRegistry, Gauge, pushadd_to_gateway

registry = CollectorRegistry()
duration = Gauge('my_job_duration_seconds',
                 'Duration of my batch job in seconds', registry=registry)
try:
    with duration.time():
        # Здесь находится ваш код.
        pass

    # Этот код выполняется, только если не возникло никаких исключений.
    g = Gauge('my_job_last_success_seconds',
              'Last time my batch job successfully finished', registry=registry)
```

¹ Подобно сводным метрикам и гистограммам, для датчиков имеется декоратор и менеджер контекста `time`. Он предназначен только для использования в пакетных заданиях.

```
g.set_to_current_time()
finally:
    pushadd_to_gateway('localhost:9091', job='batch', registry=registry)
```

Вы можете увидеть отправленные данные на странице состояния, как показано на рис. 4.2. Здесь Pushgateway добавил дополнительную метрику `push_time_seconds`, потому что Prometheus всегда использует время извлечения данных как отметку времени для метрик Pushgateway. `push_time_seconds` позволяет узнать фактическое время последней отправки данных. Также добавилась еще одна метрика, `push_failure_time_seconds`, с временемем, когда не удалось обновить эту группу в Pushgateway.

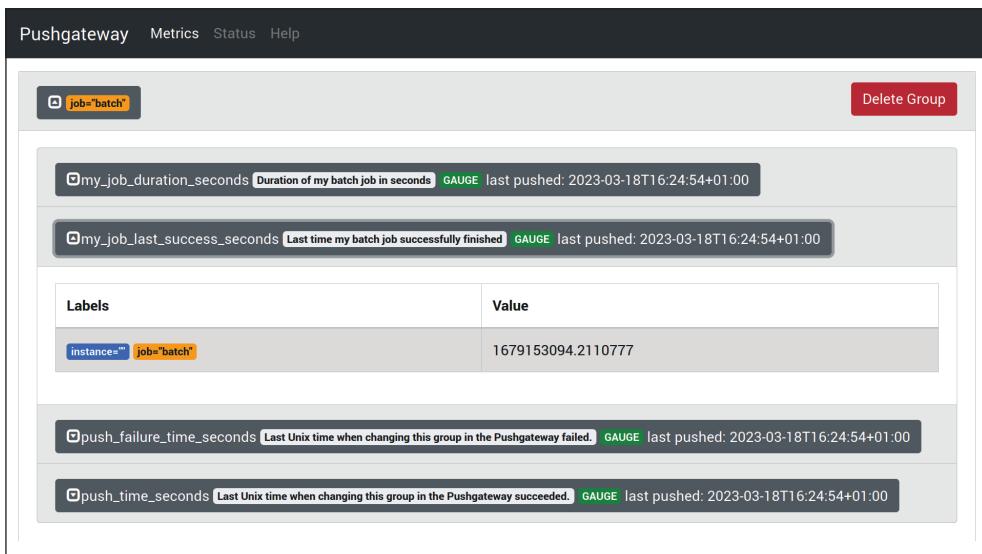


Рис. 4.2 ♦ Страница состояния Pushgateway содержит метрики, отправленные пакетным заданием

Как можно заметить на рис. 4.2, отправленный набор метрик называется *группой* (group). В дополнение к метке `job` вы можете указать дополнительные метки при отправке. Эти метки называют *ключом группировки*. В Python такие метки можно задать в именованном аргументе `grouping_key`. Дополнительные метки обычно используются, если пакетное задание каким-то образом сегментировано. Например, если в вашей базе данных имеется 30 сегментов и каждый обрабатывается отдельным пакетным заданием, то для их различия вы можете добавить метку `shard`.

- ✓ После отправки группы навсегда остаются в Pushgateway. Поэтому старайтесь избегать использования ключей группировки, меняющихся от одного запуска пакетного задания к другому, так как это затруднит работу с метриками и вызовет проблемы с производительностью. При выводе пакетного задания из эксплуатации не забудьте удалить его метрики из Pushgateway.

Мосты

Клиентские библиотеки Prometheus не ограничиваются выводом метрик в формате Prometheus. Между инструментированием и экспортацией существует разделение задач, благодаря чему вы можете обрабатывать метрики любым удобным для вас способом.

Например, клиенты для Go, Python и Java включают *мост Graphite*. Мост берет метрики из реестра клиентской библиотеки и отправляет их куда-то еще, кроме Prometheus. При этом мост Graphite bridge преобразует метрики в формат, понятный Graphite¹, и записывает их в Graphite, как показано в примере 4.13.

Пример 4.13 ♦ Использование GraphiteBridge в Python для отправки метрик в Graphite каждые 10 с

```
import time
from prometheus_client.bridge.graphite import GraphiteBridge

gb = GraphiteBridge(['graphite.your.org', 2003])
gb.start(10)
while True:
    time.sleep(1)
```

В своей работе мост опирается на метод реестра, позволяющий получить моментальный снимок всех текущих метрик. В Python это `CollectorRegistry.collect`, в Java – `CollectorRegistry.metricFamilySamples` и в Go – `Registry.Gather`. Это метод используется механизмом экспортации метрик через HTTP, и вы тоже можете его использовать. Например, этот метод можно использовать для передачи данных другой библиотеке инструментирования, не связанной с Prometheus².

Если вам потребуется обрабатывать метрики, созданные прямым инструментированием, то вместо этого используйте метрики, выдаваемые реестром. С точки зрения системы мониторинга, основанной на метриках, нет смысла обрабатывать каждое приращение счетчика. Однако количество приращений уже предоставляется методом `CollectorRegistry.collect` и может использоваться в пользовательских сборщиках.

Парсеры

Помимо реестра клиентской библиотеки, позволяющего получить выходные метрики, клиенты для Go³ и Python также имеют парсер (синтаксический

¹ Метки упрощаются до имен метрик. Поддержка тегов (т. е. меток) была добавлена в Graphite совсем недавно, в версии 1.1.0.

² Этот прием работает в обе стороны. Метрики, определяемые с помощью других библиотек инструментирования, можно передавать в клиентскую библиотеку Prometheus с помощью аналогичной функции. Этот вопрос обсуждается в разделе «Пользовательские сборщики» главы 12.

³ Парсер клиента для Go считается эталонной реализацией.

анализатор) для форматов Prometheus и OpenMetrics. Код в примере 4.14 просто выводит образцы, но вы можете передавать метрики Prometheus в другие системы мониторинга или в свои локальные инструменты.

Пример 4.14 ♦ Анализ текстового формата Prometheus с помощью клиента на Python

```
from prometheus_client.parser import text_string_to_metric_families

for family in text_string_to_metric_families(u"counter_total 1.0\n"):
    for sample in family.samples:
        print("Name: {0} Labels: {1} Value: {2}".format(*sample))
```

DataDog, InfluxDB, Sensu и Metricbeat¹ – это некоторые примеры систем мониторинга, имеющих компоненты, которые способные анализировать текстовый формат. При использовании любой из этих систем мониторинга можно воспользоваться преимуществами экосистемы Prometheus, даже не запуская сервер Prometheus. Мы считаем, что это хорошо, так как современные системы мониторинга дублируют функциональность друг друга. Каждая из них требует писать аналогичный код для поддержки множества пользовательских метрик, предоставляемых наиболее часто используемым программным обеспечением.

Текстовый формат экспорта

Текстовый формат экспорта Prometheus относительно прост. В большинстве случаев генерирование данных в этом формате можно переложить на клиентскую библиотеку, но иногда, например при создании сборщика текстовых файлов, такого как Node Exporter (обсуждается в разделе «Сборщик текстовых файлов» главы 7), может понадобиться форматировать данные вручную.

Мы покажем вам версию 0.0.4 текстового формата, которая имеет такой заголовок типа содержимого:

```
Content-Type: text/plain; version=0.0.4; charset=utf-8
```

В простейших случаях текстовый формат содержит лишь имя метрики и 64-битное число с плавающей точкой. Каждая строка завершается символом перевода строки (\n):

```
my_counter_total 14
a_small_gauge 8.3e-96
```

Типы метрик

Более полный текстовый формат Prometheus будет включать теги HELP и TYPE метрик, как показано в примере 4.15. HELP содержит описание метрики, и со-

¹ Часть стека Elasticsearch.

держимое этого тега обычно не должно меняться между циклами выборки метрик. TYPE определяет тип метрики: counter, gauge, summary, histogram или untyped. Тип untyped указывается, когда тип метрики неизвестен, и используется по умолчанию, если тип не указан явно. Дублирование метрик недопустимо, поэтому убедитесь, что все временные ряды, относящиеся к метрике, сгруппированы вместе.

Пример 4.15 ♦ Формат представления для датчика (gauge), счетчика (counter), сводной метрики (summary) и гистограммы (histogram)

```
# HELP example_gauge Пример датчика
# TYPE example_gauge gauge
example_gauge -0.7
# HELP my_counter_total Пример счетчика
# TYPE my_counter_total counter
my_counter_total 14
# HELP my_summary Пример сводной метрики
# TYPE my_summary summary
my_summary_sum 0.6
my_summary_count 19
# HELP latency_seconds Пример гистограммы
# TYPE latency_seconds histogram
latency_seconds_bucket{le="0.1"} 7 ❶
latency_seconds_bucket{le="0.2"} 18
latency_seconds_bucket{le="0.4"} 24
latency_seconds_bucket{le="0.8"} 28
latency_seconds_bucket{le="+Inf"} 29
latency_seconds_sum 0.6
latency_seconds_count 29 ❷
```

- ❶ Для гистограмм метки le имеют значения с плавающей точкой и должны быть отсортированы. Обратите внимание, что сегменты гистограммы являются кумулятивными (накопительными), потому что le означает *less than or equal to* – меньше или равно.
- ❷ Значение _count должно соответствовать сегменту +Inf, причем сегмент +Inf должен присутствовать всегда. Сегменты не должны меняться между циклами выборки метрик, так как это вызовет проблемы в PromQL-функции histogram_quantile.

Метки

Гистограмма в предыдущем примере также показывает, как оформляются метки в текстовом формате. Вы можете перечислить несколько меток через запятую, и допускается оставлять завершающую запятую перед закрывающей фигурной скобкой.

Порядок меток не имеет значения, но желательно, чтобы порядок оставался неизменным между циклами выборки метрик. Это упростит написание модульных тестов, а сохранение одного и того же порядка обеспечит наилучшую производительность приема метрик в Prometheus.

Вот пример сводной метрики в текстовом формате:

```
# HELP my_summary Пример сводной метрики
# TYPE my_summary summary
my_summary_sum{foo="bar",baz="quu"} 1.8
my_summary_count{foo="bar",baz="quu"} 453
my_summary_sum{foo="bla",baz=""} 0
my_summary_count{foo="bla",baz="quu"} 0
```

Можно определить метрику без временных рядов, если дочерние элементы не были инициализированы, как описано в разделе «Дочерние элементы» главы 5:

```
# HELP a_counter_total Пример счетчика
# TYPE a_counter_total counter
```

Экранирование

В формате текстового представления используется кодировка UTF-8, поэтому в значениях HELP и в метках допускается использовать любые символы UTF-8¹, а для экранирования символов, которые могут вызвать проблемы, нужно использовать обратный слеш. В значениях HELP такие проблемы могут вызвать символы перевода строки и сам обратный слеш. В значениях меток – перевод строки, обратный слеш и двойные кавычки². Избыточные пробелы игнорируются.

Вот пример, демонстрирующий экранирование в текстовом формате:

```
# HELP escaping Перевод строки \\n и обратный слеш \\ должны экранироваться
# TYPE escaping gauge
escaping{foo="newline \\n backslash \\ double quote \" \"} 1
```

Отметки времени

Для временного ряда можно указать отметку времени после значения – целочисленное значение времени в миллисекундах, прошедшее от начала эпохи Unix³. Но в общем случае нежелательно указывать отметки времени в формате экспорта, потому что они применимы только в определенных ограниченных случаях. Отметки времени автоматически добавляются системой Prometheus во время извлечения для метрик. Поведение системы не определено для случая, когда в массиве извлеченных данных будет присутствовать несколько строк с одинаковыми именами и метками, но с разными отметками времени.

¹ Нулевой байт является допустимым символом UTF-8.

² Да, в текстовом формате действуют два разных набора правил экранирования. В OpenMetrics они объединены в одно правило, потому что двойные кавычки в HELP всегда должны экранироваться.

³ Полночь 1 января 1970 года по всемирному координированному времени.

Следующий датчик имеет отметку времени:

```
# HELP foo Я попал в ловушку в клиентской библиотеке
# TYPE foo gauge
foo 1 15100992000000
```

! В текстовом формате Prometheus отметки времени выражаются в миллисекундах, прошедших от начала эпохи, тогда как в OpenMetrics – в секундах.

Проверка метрик

Для большей эффективности Prometheus 2.0 использует свой парсер. Поэтому возможность извлечения метрик из конечной точки `/metrics` не означает, что метрики соответствуют формату.

Проверить правильность формата представления выходных данных метрик можно с помощью `promtool` – утилиты, входящей в состав Prometheus:

```
curl http://localhost:8000/metrics | promtool check metrics
```

Среди самых распространенных ошибок, допускаемых при формировании данных для извлечения, можно назвать отсутствие перевода строки в последней строке, использование возврата каретки и перевода строки вместо одного только перевода строки¹, а также использование недопустимых имен метрик или меток. Напомним еще раз, что имена метрик и меток не могут содержать дефисы и не могут начинаться с цифры.

Теперь вы знаете о текстовом формате практически все, что нужно на практике. Полную спецификацию можно найти в официальной документации Prometheus (<https://oreil.ly/20X3R>).

OpenMetrics

Формат OpenMetrics схож с текстовым форматом Prometheus, но имеет несколько несовместимых с Prometheus отличий. Даже если обе формы представления метрик выглядят одинаково, результаты, которые они генерируют, обычно будут отличаться.

Мы покажем вам версию 1.0.0 формата OpenMetrics, которая имеет такой заголовок типа содержимого:

```
Content-Type: application/openmetrics-text; version=1.0.0; charset=utf-8
```

В простейших случаях текстовый формат содержит лишь имя метрики и 64-битное число с плавающей точкой. Каждая строка завершается символом перевода строки (\n), а файл завершается тегом # EOF:

¹ Windows строки завершаются комбинацией символов \r\n, а в Unix – одним символом \n. Prometheus изначально создавалась для Unix и поэтому использует \n.

```
my_counter_total 14
a_small_gauge 8.3e-96
# EOF
```

Типы метрик

Типы метрик, поддерживаемые текстовым форматом Prometheus, также поддерживаются в OpenMetrics. В дополнение к счетчикам, датчикам, сводным метрикам и гистограммам в формат OpenMetrics добавлены типы: StateSet, GaugeHistograms и Info.

StateSet представляет ряд связанных логических значений, также называемых набором битов. Значение 1 означает истину, а 0 – ложь.

GaugeHistograms представляет текущие распределения. В отличие от простых гистограмм значения сегментов и сумма в GaugeHistograms могут увеличиваться и уменьшаться.

Метрики Info используются для передачи текстовой информации, которая не меняется в течение жизненного цикла процесса, такой как номер версии приложения, номер фиксации в системе управления версиями и номер версии компилятора. Значение этих метрик всегда равно 1.

В дополнение к HELP и TYPE семейства метрик в OpenMetrics имеют необязательные метаданные UNIT, которые определяют единицу измерения.

Все эти типы показаны в примере 4.16.

Пример 4.16 ♦ Формат представления разных типов метрик

```
# HELP example_gauge Пример датчика
# TYPE example_gauge gauge
example_gauge -0.7
# HELP my_counter Пример счетчика
# TYPE my_counter counter
my_counter_total 14
my_counter_created 1.640991600123e+09
# HELP my_summary Пример сводной метрики
# TYPE my_summary summary
my_summary_sum 0.6
my_summary_count 19
# HELP latency_seconds Пример гистограммы
# TYPE latency_seconds histogram
# UNIT latency_seconds seconds
latency_seconds_bucket{le="0.1"} 7
latency_seconds_bucket{le="0.2"} 18
latency_seconds_bucket{le="0.4"} 24
latency_seconds_bucket{le="0.8"} 28
latency_seconds_bucket{le="+Inf"} 29
latency_seconds_sum 0.6
latency_seconds_count 29
# TYPE my_build_info info
my_build_info{branch="HEAD",version="0.16.0rc1"} 1.0
# TYPE my_stateset stateset
```

```
# HELP my_stateset Пример stateset
my_stateset{feature="a"} 1
my_stateset{feature="b"} 0
# TYPE my_gaugehistogram gaugehistogram
# HELP my_gaugehistogram Пример gaugehistogram
my_gaugehistogram_bucket{le="1.0"} 0
my_gaugehistogram_bucket{le="+Inf"} 3
my_gaugehistogram_gcount 3
my_gaugehistogram_gsum 2
# EOF
```

В OpenMetrics, как показано в примере 4.16, метрики GaugeHistograms используют другие суффиксы (`_gcount` и `_gsum`) для подсчета и суммы, отличные от `_count` и `_sum` в гистограммах.

Метки

Определения метрик с типами Histogram и GaugeHistogram также показывают, как оформляются метки. Вы можете перечислить несколько меток через запятую, но, в отличие от формата Prometheus, в OpenMetrics нельзя добавлять завершающие запятые перед закрывающей фигурной скобкой.

Отметки времени

Для временного ряда можно указать отметку времени после значения – целочисленное значение времени в миллисекундах, прошедшее от начала эпохи Unix¹, как показано ниже:

```
# HELP foo Я попал в ловушку в клиентской библиотеке
# TYPE foo gauge
foo 1 1.5100992e9
```

 В формате OpenMetrics отметки времени выражаются в секундах, прошедших от начала эпохи, тогда как в Prometheus – в миллисекундах.

Теперь вы знаете о формате OpenMetrics практически все, что нужно на практике. Полную спецификацию можно найти в официальной документации OpenMetrics на GitHub (<https://oreil.ly/EUEza>).

Мы уже не раз упоминали метки, и в следующей главе вы узнаете, что они из себя представляют.

¹ Полночь 1 января 1970 года по всемирному координированному времени.

Глава 5

Метки

Метки – одна из ключевых особенностей Prometheus, которые делают эту систему столь мощной. В этой главе вы узнаете, что такое метки, откуда они берутся и как можно добавить их к своим собственным метрикам.

Что такое метки?

Метки – это пары ключ–значение, связанные с временными рядами, которые, в дополнение к именам метрик, однозначно идентифицируют их. Это определение звучит довольно туманно, поэтому внесем немного ясности, рассмотрев пример.

Если у вас есть метрика для HTTP-запросов, разбитая по путям, то можно попробовать указать путь в имени метрики, как это принято в Graphite¹:

```
http_requests_login_total  
http_requests_logout_total  
http_requests_adduser_total  
http_requests_comment_total  
http_requests_view_total
```

С такими метриками сложно работать в PromQL. Чтобы вычислить общее количество запросов, нужно знать все возможные пути HTTP или использовать некоторую форму потенциально дорогостоящего сопоставления, чтобы отыскать все имена метрик. Такого антипаттерна следует избегать. Метки в Prometheus помогают справиться с такими ситуациями. В данном случае можно использовать метку path:

```
http_requests_total{path="/login"}  
http_requests_total{path="/logout"}  
http_requests_total{path="/adduser"}  
http_requests_total{path="/comment"}  
http_requests_total{path="/view"}
```

¹ Graphite будет использовать точки вместо символов подчеркивания.

И затем можно работать с метрикой `http_requests_total` со всеми ее метками `path` как с единым целым. В PromQL можно получить суммарную частоту запросов, частоту только для одного из путей или долю запросов для каждого пути в общем их количестве.

Также можно определить метрику с несколькими метками. Метки не устанавливают какого-то определенного порядка, поэтому агрегировать можно по любой метке, игнорируя другие, или даже по нескольким меткам сразу.

Инструментированные и целевые метки

Метки делятся на две основные категории: *инструментированные и целевые*. В PromQL между ними нет никакой разницы, но их важно различать, чтобы получить максимальную выгоду.

Инструментированные метки, как следует из названия, неразрывно связаны с инструментированием. Они отражают все, что известно внутри вашего приложения или библиотеки, например типы получаемых HTTP-запросов, имена баз данных, с которыми взаимодействует код, и другие внутренние особенности.

Целевые метки идентифицируют конкретную цель мониторинга, т. е. цель, откуда Prometheus извлекает метрики. Целевые метки отражают информацию о вашей архитектуре, такую как имя приложения, центр обработки данных, где приложение выполняется, тип среды выполнения – разработка или промышленной эксплуатации, команда, владеющая приложением, и, конечно же, конкретный экземпляр приложения. Целевые метки прикрепляются системой Prometheus в процессе анализа метрик.

Разные серверы Prometheus, управляемые разными командами, могут иметь разные представления о том, что такое «команда», «регион» или «служба», поэтому само инструментированное приложение не должно пытаться экспортить такие метки. Соответственно, вы не найдете в клиентских библиотеках средств для добавления меток¹ к метрикам целей. Целевые метки создаются в ходе обнаружения служб и изменения меток² и обсуждаются далее в главе 8.

¹ Или префиксов к именам метрик.

² При использовании Pushgateway целевые метки могут исходить из приложения, так как каждая группа Pushgateway находится в пути мониторинга цели. В зависимости от того, кому задается вопрос, эта особенность может расцениваться как достоинство или как недостаток мониторинга с применением кеша Pushgateway.

Инструментирование

Давайте расширим пример 3.3 и используем метку. В примере 5.1 можно увидеть параметр `labelnames=['path']` в определении метрики¹. Он указывает, что метрика имеет одну метку с именем `path`. При экспортации инструментированной метрики с меткой необходимо дополнительно вызвать метод `labels` со значением метки в аргументе².

Пример 5.1 ♦ Приложение на Python, добавляющее метку к метрике-счетчику

```
import http.server
from prometheus_client import start_http_server, Counter

REQUESTS = Counter('hello_worlds_total',
    'Hello Worlds requested.',
    labelnames=['path'])

class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        REQUESTS.labels(self.path).inc()
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello World")

if __name__ == "__main__":
    start_http_server(8000)
    server = http.server.HTTPServer(('localhost', 8001), MyHandler)
    server.serve_forever()
```

Если теперь открыть URL `http://localhost:8001` и `http://localhost:8001/foo`, то на странице `/metrics` по адресу `http://localhost:8000/metrics` вы увидите временные ряды для каждого из путей:

```
# HELP hello_worlds_total Hello Worlds requested.
# TYPE hello_worlds_total counter
hello_worlds_total{path="/favicon.ico"} 6.0
hello_worlds_total{path="/" 4.0
hello_worlds_total{path="/foo"} 1.0
```

Имена меток имеют некоторые ограничения с точки зрения диапазона допустимых символов. Они должны начинаться с буквы (a–z или A–Z), за которой могут следовать буквы, цифры и символы подчеркивания. Это то же ограничение, что действует в отношении имен метрик, за исключением двоеточий.

¹ При программировании на Python будьте осторожны и не используйте аргумент `labelnames='path'`, потому что это равносильно передаче аргумента `labelnames=['p', 'a', 't', 'h']`. Это одна из наиболее распространенных ошибок в Python.

² В Java этот метод имеет то же имя `labels`, а его эквивалент в Go – имя `WithLabelValues`.

В отличие от имен метрик имена меток обычно не имеют пространства имен. Однако будьте осторожны при определении инструментированных меток, чтобы избежать конфликтов с именами, которые могут быть выбраны для целевых меток, такими как `env`, `cluster`, `service`, `team`, `zone` и `region`. Мы также не рекомендуем использовать имя `type`, так как оно является слишком общим. Еще одно соглашение, принятое в отношении имен меток, – змеиная нотация.

Имена меток `instance` и `job` изначально используются системой Prometheus, поэтому мы также не рекомендуем использовать их, так как они будут конфликтовать с целевыми метками.

Значения меток могут включать любые символы UTF-8. Также метки могут иметь пустое значение, но это может вызывать путаницу на сервере Prometheus, так как на первый взгляд метка с пустым значением выглядит как отсутствующая метка.

Зарезервированные метки и `_name_`

Имена меток могут начинаться с символа подчеркивания, но таких меток следует избегать. Имена меток, начинающиеся с двойного подчеркивания `__`, зарезервированы для использования системой.

Внутри Prometheus имя метрики – это всего лишь метка с именем `_name_`¹. Выражение `up` – это синтаксический сахар для выражения `{_name_= "up"}`. Кроме того, операторы PromQL поддерживают специальную семантику, как описано в разделе «Сопоставление векторов» главы 15.

Метрики

Как вы могли заметить, *метрика* – довольно широкий термин и может иметь разные толкования в зависимости от контекста. То же относится к понятиям «семейство метрик», «дочерние элементы» и «временные ряды»:

```
# HELP latency_seconds Latency in seconds.
# TYPE latency_seconds summary
latency_seconds_sum{path="/foo"} 1.0
latency_seconds_count{path="/foo"} 2.0
latency_seconds_sum{path="/bar"} 3.0
latency_seconds_count{path="/bar"} 4.0
```

`latency_seconds_sum{path="/bar"}` – это временной ряд, отличающийся именем и метками. Это то, с чем работает PromQL.

`latency_seconds{path="/bar"}` – это дочерний элемент, возвращаемый методом `labels()` в клиенте Python. Он содержит временные ряды `_sum` и `_count` с этими метками.

`latency_seconds` – это семейство метрик, определяющее имя метрики и связанный с ней тип. Это определение метрики при использовании клиентской библиотеки.

¹ Это не то же самое, что переменная `_name_` в Python.

Для метрик-датчиков без меток понятия семейство, дочерний элемент и временной ряд совпадают.

Несколько меток

Определяя метрику, можно указать любое количество меток, а затем перечислить их значения в том же порядке в вызове `labels` (пример 5.2).

Пример 5.2 ♦ `hello_worlds_total` имеет метки `path` и `method`

```
REQUESTS = Counter('hello_worlds_total',
    'Hello Worlds requested.',
    labelnames=['path', 'method'])

class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        REQUESTS.labels(self.path, self.command).inc()
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello World")
```

Python и Go также позволяют передавать словари с именами и значениями меток, но имена меток в словарях должны совпадать с именами, указанными в определениях метрик. Это может вызывать трудности при изменении порядка аргументов, и если такое возможно, то вы можете получить слишком много меток.

Имена меток не могут изменяться динамически, и клиентские библиотеки предотвратят любые попытки использовать их. При работе с метриками вы должны заранее знать, какие метки у вас есть, и, соответственно, осуществляя прямое инструментирование, вы должны знать имена своих меток. Если имена меток заранее неизвестны, то, скорее всего, вам следует подумать об инструменте мониторинга на основе журналов.

Дочерние элементы

Значение, возвращаемое методом `labels` в Python, называется *дочерним элементом*. Его можно сохранить для последующего использования, что избавит от необходимости искать этот элемент при каждом событии инструментирования и поможет сэкономить время в коде, производительность которого может быть критически важна. Выполняя тесты производительности Java-клиента, мы обнаружили, что в однопоточном приложении поиск дочернего элемента занимает 30 нс, а фактическое увеличение счетчика – 12 нс¹.

¹ По этой причине не поддавайтесь искушению написать обертку вокруг клиентской библиотеки Prometheus, которая принимает имя метрики в аргументе, так как это неизбежно повлечет затраты на поиск. Дешевле, проще и семантически лучше иметь переменную на уровне файла, хранящую адрес объекта метрики, чем постоянно искать его.

Когда объект ссылается только на один дочерний элемент метрики, многие предпочитают вызвать `labels` один раз и сохранить результат в объекте, как показано в примере 5.3.

Пример 5.3 ❖ Простой кеш в Python, хранящий дочерний элемент в именованном кеше

```
from prometheus_client import Counter

FETCHES = Counter('cache_fetches_total',
                  'Fetches from the cache.',
                  labelnames=['cache'])

class MyCache(object):
    def __init__(self, name):
        self._fetches = FETCHES.labels(name)
        self._cache = {}

    def fetch(self, item):
        self._fetches.inc()
        return self._cache.get(item)

    def store(self, item, value):
        self._cache[item] = value
```

Еще одно место, где вы столкнетесь с дочерними элементами, – их инициализация. Дочерние элементы появляются на странице `/metrics` только после вызова метода `labels`¹. Такое положение вещей может вызвать проблемы в PromQL, потому что временные ряды, появляющиеся и исчезающие, очень сложно обрабатывать. Соответственно, если есть возможность, инициализируйте дочерние элементы в момент запуска, как в примере 5.4; впрочем, если следовать шаблону из примера 5.3, то все это будет делаться автоматически.

Пример 5.4 ❖ Инициализация дочерних элементов метрики при запуске приложения

```
from prometheus_client import Counter

REQUESTS = Counter('http_requests_total',
                   'HTTP requests.',
                   labelnames=['path'])
REQUESTS.labels('/foo')
REQUESTS.labels('/bar')
```

При использовании декораторов Python вы также можете использовать `labels` без немедленного вызова метода возвращаемого значения, как показано в примере 5.5.

¹ Это не относится к метрикам без меток.

Пример 5.5 ♦ Использование декоратора с метками в Python

```
from prometheus_client import Summary

LATENCY = Summary('http_requests_latency_seconds',
                  'HTTP request latency.',
                  labelnames=['path'])

foo = LATENCY.labels('/foo')
@foo.time()
def foo_handler(params):
    pass
```

i Клиентские библиотеки обычно предлагают методы удаления дочерних элементов из метрики. Но их следует использовать только в модульных тестах. С точки зрения PromQL, если дочерний элемент существует, он должен продолжать существовать до завершения процесса, иначе такие функции, как `rate`, могут возвращать неожиданные результаты. Эти методы также делают недействительными предыдущие значения, полученные вызовом `labels`.

Агрегирование

Теперь, добавив метки к инструментированным метрикам, используем их в PromQL. Подробнее об операторах агрегирования мы поговорим в главе 14, а сейчас я хочу дать вам почувствовать силу меток.

В примере 5.2 метрика `hello_worlds_total` имеет метки `path` и `method`. Поскольку `hello_worlds_total` является счетчиком, прежде всего необходимо использовать функцию `rate`. В табл. 5.1 представлен один из возможных вариантов выходных данных, показывающий результаты мониторинга двух экземпляров приложения с разными путями и методами HTTP.

Таблица 5.1. Вывод `rate(hello_worlds_total [5m])`

{job="myjob",instance="localhost:1234",path="/foo",method="GET"}	1
{job="myjob",instance="localhost:1234",path="/foo",method="POST"}	2
{job="myjob",instance="localhost:1234",path="/bar",method="GET"}	4
{job="myjob",instance="localhost:5678",path="/foo",method="GET"}	8
{job="myjob",instance="localhost:5678",path="/foo",method="POST"}	16
{job="myjob",instance="localhost:5678",path="/bar",method="GET"}	32

Это может вызвать некоторые сложности, особенно если у вас гораздо больше временных рядов, чем в этом простом примере. Давайте начнем с агрегирования по метке `path`. Это можно сделать с помощью агрегирующей функции `sum`, чтобы сложить значения образцов. Оператор `without` указывает, какую метку следует исключить. В результате получается выражение `sum without(path)(rate(hello_worlds_total[5m]))`, дающее результат, показанный в табл. 5.2.

Таблица 5.2. Результат sum without(path)(rate(hello_worlds_total[5m]))

{job="myjob",instance="localhost:1234",method="GET"}	5
{job="myjob",instance="localhost:1234",method="POST"}	2
{job="myjob",instance="localhost:5678",method="GET"}	40
{job="myjob",instance="localhost:5678",method="POST"}	16

Нередко запускаются десятки или сотни экземпляров приложения, за которым осуществляется наблюдение, и, как показывает наш опыт, при просмотре отдельных экземпляров в дашбордах анализируются не более трех-пяти экземпляров. Учитывая это, можно расширить предложение `without` и включить в него метку экземпляра, что даст результат, показанный в табл. 5.3. Как и следовало ожидать, исходя из значений в табл. 5.1, $1 + 4 + 8 + 32 = 45$ запросов GET в секунду и $2 + 16 = 18$ запросов POST в секунду.

Таблица 5.3. Результат sum without(path,instance)(rate(hello_worlds_total[5m]))

{job="myjob",method="GET"}	45
{job="myjob",method="POST"}	18

Метки никак не упорядочены, поэтому можно удалить не только `path`, но и `method`, как показано в табл. 5.4.

Таблица 5.4. Результат sum without(method,instance)(rate(hello_worlds_total[5m]))

{job="myjob",path="/foo"}	27
{job="myjob",method="POST"}	36

i Существует также оператор `by`, сохраняющий только указанные метки. Но `without` предпочтительнее, потому что если есть дополнительные метки, такие как `env` или `region`, то они не будут потеряны. Это может помочь делиться своими правилами с другими.

Шаблоны использования меток

В качестве значений временных рядов Prometheus поддерживает не любые другие типы данных, а только 64-битные числа с плавающей запятой. Но метки имеют строковые значения, и есть определенные случаи использования, когда можно использовать их взамен мониторинга журналов.

Метрики-перечисления

Первый типичный случай применения строк – *перечисления*. Например, у вас может быть ресурс, который может находиться ровно в одном из состояний: `STARTING` (запускается), `RUNNING` (работает), `STOPPING` (останавливается) или `TERMINATED` (остановлен).

Вы можете экспортировать это состояние с помощью датчика, где состоянию STARTING соответствует 0, состоянию RUNNING – 1, STOPPING – 2 и TERMINATED – 3¹. Но с такими соответствиями сложно работать в PromQL. Числа от 0 до 3 сами по себе не несут никакой смысловой нагрузки, и у вас не получится написать выражение, например, чтобы узнать, какую долю времени ресурс находился в состоянии STARTING.

Чтобы решить эту проблему, можно добавить метки, описывающие состояние датчика, при этом каждое потенциальное состояние становится дочерним. При экспортации логических значений в Prometheus используйте 1 для представления истинного значения и 0 – ложного. Соответственно, один из дочерних элементов будет иметь значение 1, а все остальные – 0, что приведет к получению метрик, подобных приведенным в примере 5.6.

Пример 5.6 ♦ Пример перечисления; ресурс blaa находится в состоянии RUNNING

```
# HELP gauge The current state of resources.
# TYPE gauge resource_state
resource_state{resource_state="STARTING",resource="blaa"} 0
resource_state{resource_state="RUNNING",resource="blaa"} 1
resource_state{resource_state="STOPPING",resource="blaa"} 0
resource_state{resource_state="TERMINATED",resource="blaa"} 0
```

Поскольку нули присутствуют всегда, выражение PromQL `avg_over_time(resource_state[1h])` даст долю времени, проведенного в каждом состоянии. Также есть возможность агрегировать по `resource_state`, используя `sum without(resource) (resource_state)`, чтобы увидеть, сколько ресурсов находятся в каждом состоянии.

Для получения таких метрик можно применить `set` к датчику, но это приведет к состоянию гонки. Процедура извлечения данных может увидеть единицу в двух состояниях или не увидеть ни в одном, в зависимости от момента, когда происходит получение данных. Чтобы исключить такие ситуации, необходима изоляция датчика, чтобы его чтение не могло происходить в середине обновления.

Решением этой проблемы является использование *пользовательского сборщика*, который мы обсудим в разделе «Пользовательские сборщики» главы 12. Чтобы вы могли получить представление о том, как это сделать, в примере 5.7 приводится базовая реализация. На самом деле подобный код добавляется в существующий класс, а не создается отдельный класс².

Пример 5.7 ♦ Пользовательский сборщик для датчика, используемого в роли перечисления

```
from threading import Lock
from prometheus_client.core import GaugeMetricFamily, REGISTRY
```

¹ Именно так действуют перечисления, например, в языке C.

² Вполне вероятно, что будущие версии клиентских библиотек будут предлагать утилиты, упрощающие работу с перечислениями. OpenMetrics, например, в настоящее время планирует ввести тип набора (*множества*) состояний, частным случаем которого являются перечисления.

```
class StateMetric(object):
    def __init__(self):
        self._resource_states = {}
        self._STATES = ["STARTING", "RUNNING", "STOPPING", "TERMINATED",]
        self._mutex = Lock()

    def set_state(self, resource, state):
        with self._mutex:
            self._resource_states[resource] = state

    def collect(self):
        family = GaugeMetricFamily("resource_state",
                                   "The current state of resources.",
                                   labels=["resource_state", "resource"])
        with self._mutex:
            for resource, state in self._resource_states.items():
                for s in self._STATES:
                    family.add_metric([s, resource], 1 if s == state else 0)
        yield family

sm = StateMetric()
REGISTRY.register(sm)

# Задействовать StateMetric.
sm.set_state("bla", "RUNNING")
```

Датчики-перечисления – это самые обычные датчики, поддерживающие обычную семантику датчиков, поэтому добавлять в имя специальный метрический суффикс не требуется.

Обратите внимание, что этот прием имеет ограничения, о которых следует помнить. Если количество состояний в сочетании с количеством других меток становится слишком большим, то могут возникнуть проблемы с производительностью из-за объема выбираемых временных рядов. В таких случаях можно попробовать объединить похожие состояния, а в особенно тяжелых случаях для представления перечислений вам, возможно, придется вернуться к использованию индикаторов со значениями, такими как 0–3, и самим решать сложности, возникающие в PromQL. Этот вопрос обсуждается в разделе «Кардинальность» далее в этой главе.

Информационные метрики

Вторым распространенным случаем применения строк являются *информационные метрики*, которые по историческим причинам также могут называться *машинными ролями*¹. Нет смысла использовать их в качестве целевых меток, которые применяются ко всем метрикам в цели (обсуждается в разделе «Метки целей» главы 8).

¹ Статья Брайана (https://oreil.ly/eu_jZ) была первым документом, описывающим этот метод.

Данное соглашение состоит в том, чтобы использовать датчик со значением 1 и всеми строками, которые хотелось бы иметь в качестве целевых меток. Датчик должен иметь суффикс `_info`. Вариант такого датчика с метрикой `python_info` показан на рис. 3.2. Его определение может выглядеть примерно так, как показано в примере 5.8.

Пример 5.8 ♦ Метрика `python_info`, которую клиент Python предоставляет по умолчанию

```
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation="CPython",major="3",minor="5",patchlevel="2",
version="3.5.2"} 1.0
```

Создать такую метрику в Python можно прямым инструментированием или с помощью пользовательского сборщика. В примере 5.9 показано решение с прямым инструментированием, где также используется возможность передачи меток в виде именованных аргументов.

Пример 5.9 ♦ Создание информационной метрики прямым инструментированием

```
from prometheus_client import Info

version_info = {
    "implementation": "CPython",
    "major": "3",
    "minor": "5",
    "patchlevel": "2",
    "version": "3.5.2",
}

INFO = Info("my_python", "Python platform information")
INFO.labels(version_info)
```

Информационную метрику можно присоединить к любой другой метрике с помощью оператора умножения и модификатора `group_left`. Для объединения метрик можно использовать любой оператор, но, поскольку значение информационной метрики равно 1, умножение не изменит значения другой метрики¹.

Значение 1 определяется типом метрики, в данном случае `Info`.

Чтобы добавить метку `version` из `python_info` ко всем метрикам `up`, следует использовать выражение PromQL:

```
up
* on (instance, job) group_left(version)
  python_info
```

¹ Формально 1 – это элемент идентичности для умножения.

Параметр `group_left(version)` указывает, что это соответствие «многие к одному»¹, и метка `version` должна быть скопирована из `python_info` во все метрики `up`, имеющие одинаковые метки `job` и `instance`. Мы рассмотрим `group_left` более подробно в разделе «Многие к одному и `group_left`» главы 15.

Глядя на это выражение, можно сказать, что выходные данные будут иметь метки метрики `up` с добавленной меткой `version`. Добавить все метки из `python_info` невозможно, так как потенциально могут иметься неизвестные метки с обеих сторон выражения², что семантически неправильно. Важно всегда знать заранее, какие метки используются.

Критические изменения и метки

Добавление или удаление меток, созданных методом инструментирования, – это всегда критическое изменение. Удаление метки удаляет различие, от которого может зависеть пользователь. Добавление метки нарушает агрегирование, в котором используется оператор `without`.

Единственное исключение – информационные метрики. Для них выражения PromQL сконструированы так, что дополнительные метки не являются проблемой, поэтому вы можете без опаски добавлять метки к информационным метрикам.

Информационные метрики также имеют значение 1, поэтому с помощью `sum` легко рассчитать, сколько временных рядов имеют каждое значение метки. Количество экземпляров приложения, выполняющихся под управлением разных версий Python, будет суммироваться как `sum by (version)(python_info)`. Если бы это было другое значение, например 0, то в вашей иерархии агрегирования потребовалось бы смешивать `sum` и `count`, что чревато ошибками.

Когда использовать метки

Чтобы метрика была полезной, она должна как-то агрегироваться. Эмпирическое правило: суммирование или усреднение метрики должно давать значимый результат. Для счетчика HTTP-запросов, разделенных по путям и методам, сумма представляет общее количество запросов. А вот для очереди объединение элементов в ней и ее максимального размера в одну метрику не имеет смысла, так как ни суммирование, ни усреднение не дает никакой полезной информации.

Простым признаком, что инструментированная метка бесполезна, может служить необходимость при каждом использовании метрики указывать эту

¹ В этом случае «один к одному», потому что для метрики `python_info` существует только один временной ряд; однако это же выражение можно использовать для метрик с несколькими временными рядами для каждой цели.

² Целевые метки для `up` и любые дополнительные метки, полученные методом инструментирования, которые могут быть добавлены в `python_info` в будущем.

метку в PromQL¹. В таких случаях, вероятно, следует переместить метку в имя метрики.

Еще одна вещь, которой следует избегать, – наличие временного ряда, являющегося суммой остальной части метрики, например:

```
some_metric{label="foo"} 7
some_metric{label="bar"} 13
some_metric{label="total"} 20
```

или

```
some_metric{label="foo"} 7
some_metric{label="bar"} 13
some_metric{} 20
```

В обоих случаях нарушается агрегирование с использованием `sum` в PromQL, так как будет выполняться двойная работа. PromQL уже предоставляет возможность вычислять этот агрегат.

Исключение таблицы

Проницательные читатели, вероятно, заметили, что квантили сводных метрик нарушают правило о смысловой нагрузке для сумм и средних значений, потому что бессмысленно выполнять математические операции с квантилями.

Это то, что мы называем *исключением таблицы* (table exception), когда, даже если нет возможности выполнять математические операции с метрикой, лучше использовать метку, чем применять регулярные выражения к именам метрик. Применение регулярных выражений к именам метрик – плохой признак, и их никогда не следует использовать в графиках или уведомлениях.

Это исключение должно иметь место только в экспортёрах и никогда в прямом инструментировании. Например, у вас может быть неизвестное сочетание напряжений, скоростей вращения вентиляторов и температур, поступающих от аппаратных датчиков. В отсутствие информации, необходимой для разделения их на разные метрики, единственное, что вам остается, – это запихнуть их все в одну метрику и предоставить человеку возможность самому интерпретировать ее.



Имена меток, используемых для метрик, не должны меняться в течение всего цикла работы приложения. Если же это необходимо, то вам, вероятно, следует обратиться к мониторингу журналов.

Кардинальность

Не злоупотребляйте метками. Мониторинг – это средство достижения цели, поэтому большое количество временных рядов не всегда равноценно высокому качеству мониторинга. Для системы мониторинга, независимо от

¹ Если метка не является целевой меткой.

того, запускаете вы ее локально и самостоятельно или платите сторонней компании за ее запуск в облаке, каждый временной ряд и каждая метрика имеют свою стоимость в смысле потребления ресурсов и трудозатрат на поддержание системы мониторинга в рабочем состоянии.

В этом контексте мы хотели бы поговорить о кардинальности, под которой в Prometheus подразумевается количество имеющихся временных рядов. Если вы используете Prometheus для обработки, скажем, 10 млн временных рядов, то как лучше всего их организовать? В какой момент желательно перевести некоторые варианты использования на мониторинг журналов?

Вот как это видится нам: предположим, что кто-то, использующий ваш код, запускает тысячу экземпляров конкретного приложения¹. Добавление простой метрики-счетчика в малоизвестную подсистему добавит тысячу временных рядов в Prometheus, что составит 0.01 % его мощности. То есть добавление метрики обойдется вам почти бесплатно, и однажды она может помочь вам отладить малопонятную проблему. Во всем приложении и его библиотеках у вас может иметься сотня таких метрик, которые в сумме будут потреблять 1 % мощности системы мониторинга. Это все еще довольно дешево, даже учитывая редкость, с которой вы, вероятно, будете использовать эти метрики.

Теперь рассмотрим метрику с меткой из 10 значений, а также гистограмму, которая по умолчанию имеет 12 временных рядов². Это 120 рядов, или 1.2 % мощности системы мониторинга. Выгоды этого компромисса уже не так очевидны. Было бы неплохо иметь несколько таких метрик, но вместо этого вы также можете подумать о переходе на сводную метрику без квантилей³.

На следующем этапе хлопот становится еще больше. Если метка уже имеет кардинальность 10, то велика вероятность, что со временем она будет только увеличиваться по мере с появлением новых функций в вашем приложении. Число элементов 10 сегодня может превратиться в 15 в следующем году, а 200 может увеличиться до 300. С увеличением числа пользователей обычно приходится увеличивать число действующих экземпляров приложения. Если у вас есть более одной из таких расширяющихся меток на метрике, то ситуация усугубляется, и все завершается комбинаторным взрывом количества временных рядов. И это только одно из постоянно растущих приложений, за которыми следует Prometheus.

Итак, кардинальность может подкрасться к вам незаметно. Часто очевидно, что адреса электронной почты, клиенты и IP-адреса – плохой выбор для значений меток с точки зрения кардинальности. Менее очевидно, что HTTP-путь тоже может стать проблемой. Учитывая, что метрика, подсчитывающая HTTP-запросы, используется регулярно, удаление из нее меток, отключение от гистограммы или даже уменьшение количества сегментов в гистограмме может оказаться сложным предприятием с политической точки зрения.

¹ Можно запустить и больше, но тысяча – это достаточно консервативная верхняя граница.

² То есть 10 сегментов плюс `_sum` и `_count`.

³ Имея только временные ряды `_sum` и `_count`, сводные метрики без квантилей являются очень дешевым способом получить представление о задержках.

Эмпирическое правило, которое используем мы: кардинальность произвольной метрики в одном экземпляре приложения должна быть ниже 10. Также нет ничего плохого в том, чтобы иметь несколько метрик с кардинальностью около 100, но при этом вы должны быть готовы уменьшить кардинальность метрики перейти на мониторинг журналов с ростом количества элементов.

- i** Пара сотен метрик с высокой кардинальностью на один сервер Prometheus предполагает экспорт тысяч экземпляров, раскрывающих эту кардинальность. Если вы на 100 % уверены, что не достигнете этих цифр, например если из множества приложений вы будете запускать только одно, то можете соответствующим образом изменить эмпирическое правило.

Внедряя систему Prometheus в разных организациях, мы подметили общую закономерность. Обычно при внедрении организации сталкиваются с необходимостью обучения персонала. В какой-то момент у них что-то щелкает, и они начинают понимать силу меток. После этого обычно следуют жалобы на низкую производительность Prometheus, обусловленные кардинальностью меток. Мы советуем заранее обсудить ограничения кардинальности с вашими пользователями, а также рассмотреть возможность использования `sample_limit` в качестве аварийного предохранителя (см. раздел «Снижение нагрузки» главы 21).

Десять самых больших метрик в Prometheus обычно потребляют более половины его ресурсов, и это почти всегда связано с кардинальностью меток. Иногда возникает путаница: если проблема заключается в количестве значений меток, то можно ли решить проблему, переместив значения метки в имя метрики? Поскольку основным ограничением ресурсов на самом деле является кардинальность временного ряда (зависящая от количества значений меток), перенос значений меток в имя метрики не меняет кардинальности, а просто затрудняет использование метрик¹.

Теперь, узнав, как добавлять метрики в свои приложения, и познакомившись с некоторыми базовыми выражениями PromQL, в следующей главе вы увидите, как создавать дашборды в Grafana.

¹ А также затрудняет точное определение метрик, отвечающих за потребление ресурсов.

Глава 6

Создание дашбордов с Grafana

Получая уведомление или проверяя текущую работу своих систем, первым делом вы обратитесь к дашбордам. Браузер выражений, с которым вы познакомились в предыдущих главах, отлично подходит для получения специальных графиков и для отладки выражений PromQL, но он не предназначен для использования в качестве дашборда.

Что мы понимаем под дашбордом? Набор графиков, таблиц и других визуальных представлений, отражающих работу ваших систем. У вас может быть дашборд, отображающий глобальный трафик и объемы трафика, получаемые отдельными службами, и задержки, возникающие при обработке запросов. Для анализа работы служб у вас, вероятно, будет иметься отдельный дашборд, отображающий задержки, количество ошибок, частоту запросов, количество экземпляров, потребление процессора и памяти, а также метрики, характерные для служб. Для получения еще более подробной информации вы можете создать дашборды для отдельных подсистем и служб или для управления сборщиком мусора, которые можно использовать с любым приложением на Java.

Grafana – это популярный инструмент, с помощью которого можете создавать такие дашборды для различных систем, в том числе и для систем мониторинга, включая Graphite, InfluxDB, Jaeger, Elasticsearch и PostgreSQL. Это рекомендуемый инструмент для создания дашбордов при использовании Prometheus, и его поддержка в Prometheus постоянно улучшается.

В этой главе мы познакомимся с использованием Grafana в Prometheus и расширим возможности Prometheus и Node Exporter, которые настроили в главе 2.

Promdash и шаблоны консоли

Изначально в проекте Prometheus имелся собственный инструмент для создания дашбордов – *Promdash*. На начальном этапе Promdash был лучшим из всех вариантов для использования в Prometheus, но в 2016 году разработчики Prometheus решили отдать предпочтение Grafana и прекратить разработку своего решения для создания дашбордов. В настоящее время Prometheus – это один из самых популярных плагинов в Grafana¹.

В Prometheus есть так называемые *шаблоны консоли*, которые можно использовать для создания дашбордов. В отличие от Promdash и Grafana, которые хранят дашборды в реляционных базах данных, шаблоны консоли встроены прямо в Prometheus и настраиваются с помощью конфигурационных файлов. Они позволяют отображать веб-страницы с помощью языка шаблонов Go² и хранить дашборды в системе управления версиями. Шаблоны консоли – это очень низкоуровневая особенность, на основе которой можно построить систему дашбордов, и поэтому она рекомендуется только для узкоспециализированных вариантов использования и опытных пользователей.

Установка

Получить дистрибутив Grafana можно на веб-сайте проекта Grafana (<https://oreil.ly/ANoWC>). Там же вы найдете инструкции по установке, но если вы используете Docker, то установка должна производиться с его помощью:

```
docker run -d --name=grafana --net=host grafana/grafana:9.1.6
```

Обратите внимание, что при этом не требуется монтировать том³, поэтому вся информация о состоянии будет храниться внутри контейнера.

Здесь мы используем Grafana 9.1.6. Вы можете использовать более новую версию, но имейте в виду, что визуальный интерфейс в вашей версии может немного отличаться от того, что вы увидите здесь.

После запуска Grafana станет доступна в браузере по адресу `http://localhost:3000/`. Открыв эту страницу, вы должны увидеть форму входа, как показано на рис. 6.1.

Выполните вход с именем пользователя по умолчанию `admin` и паролем по умолчанию `admin`. Вам будет предложено сменить пароль, что мы и рекомендуем сделать.

После этого вы должны увидеть домашнюю панель управления, изображенную на рис. 6.2. Мы переключились на светлую тему в настройках, чтобы скриншоты выглядели более читабельными на страницах книги.

¹ Grafana по умолчанию сообщает анонимную статистику использования. Ее можно отключить с помощью параметра `reporting_enabled` в файле конфигурации.

² Это тот же язык шаблонов, что используется для шаблонов уведомлений, с некоторыми незначительными отличиями, касающимися списка доступных функций.

³ Способ совместного использования файловых систем контейнерами, поскольку по умолчанию хранилище контейнера Docker доступно только данному контейнеру. Чтобы смонтировать том, нужно передать команде `docker` флаг `-v`.

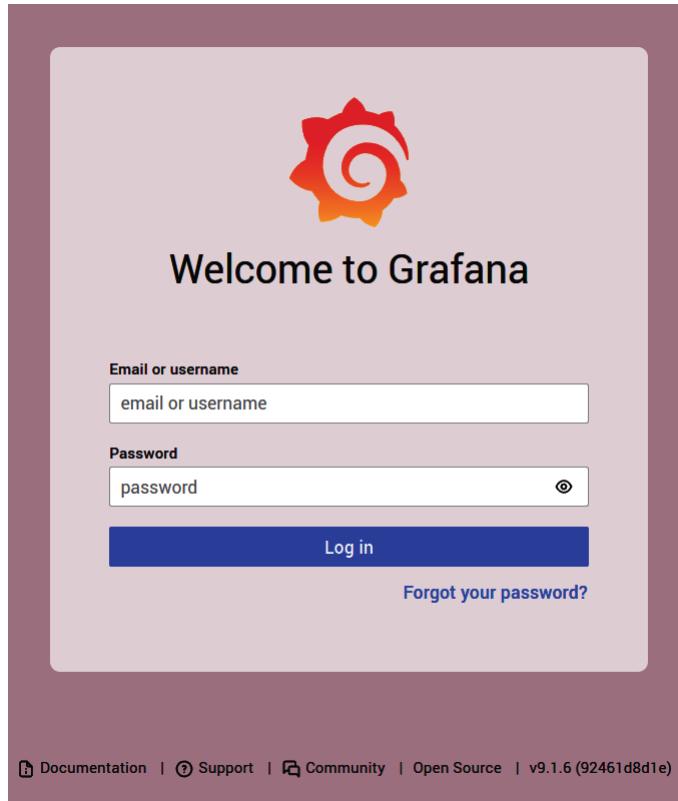


Рис. 6.1 ❖ Форма входа в Grafana

The image shows the Grafana home dashboard. On the left side, there is a sidebar with various icons: a gear, a magnifying glass, a star, a grid, a question mark, a shield, and a person icon. The main content area has a header "General / Home" and a "Welcome to Grafana" message. Below this, there is a "Need help?" section with links to "Documentation", "Tutorials", "Community", and "Public Slack". The main content is divided into three panels: 1) A "Basic" panel with text about setting up Grafana quickly. 2) A "TUTORIAL DATA SOURCE AND DASHBOARDS" panel with a "Grafana fundamentals" section and a detailed description of the tutorial. 3) A "DATA SOURCES" panel with a "Add your first data source" section and a "Learn how in the docs" link. There are also "Remove this panel" and "">>" buttons.

Рис. 6.2 ❖ Домашняя панель управления сразу после установки Grafana

Источники данных

Для получения информации, используемой при построении графиков, Grafana подключается к Prometheus через *источники данных*. По умолчанию поддерживается несколько разных типов источников данных, включая InfluxDB, PostgreSQL и, конечно же, Prometheus. У вас может быть много источников данных одного типа и обычно по одному на каждый действующий сервер Prometheus. Дашборды Grafana могут содержать графики с информацией из разных источников, и даже есть возможность смешивать данные из разных источников в одном графике.

Последние версии Grafana позволяют легко и просто добавить первый источник данных. Щелкните по ссылке **Add your first data source** (Добавить первый источник данных) и добавьте источник данных с именем **Prometheus**, типом Prometheus и URL `http://localhost:9090` (или любым другим URL, который прослушивает ваш сервер Prometheus). Форма добавления первого источника данных должна выглядеть, как показано на рис. 6.3. Оставьте все остальные настройки как есть и щелкните по кнопке **Save & Test** (Сохранить и проверить) внизу формы. В зависимости от размера окна браузера вам может потребоваться прокрутить его, чтобы увидеть кнопки. Если все в порядке, то вы получите сообщение о том, что источник данных работает.

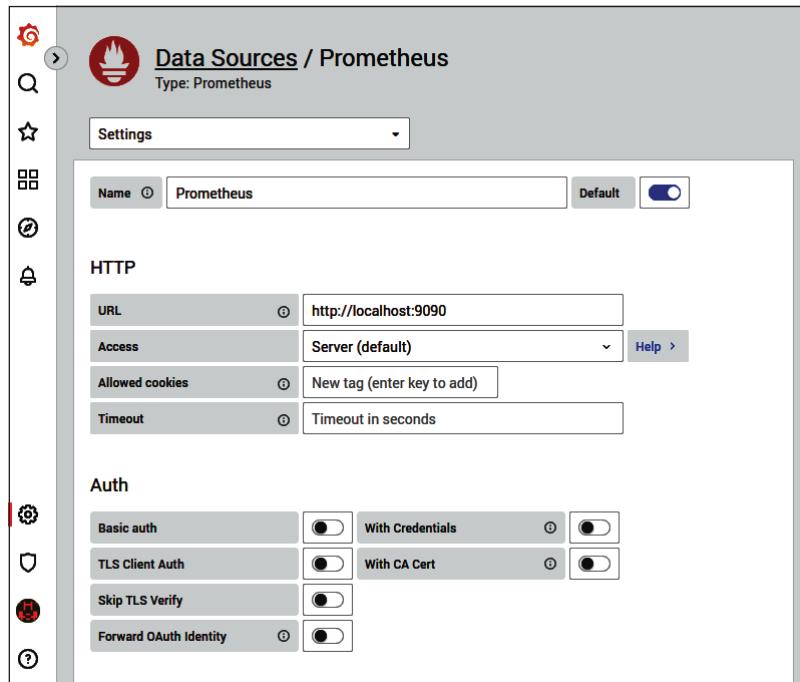


Рис. 6.3 ♦ Добавление источника данных Prometheus в Grafana

Если нет, то проверьте, запущен ли сервер Prometheus и доступен ли он из Grafana¹.

Дашборды и панели

Снова перейдите в браузере по адресу `http://localhost:3000/` и на этот раз щелкните по ссылке **Create your first dashboard** (Создать первый дашборд). В ответ откроется страница, как показано на рис. 6.4.

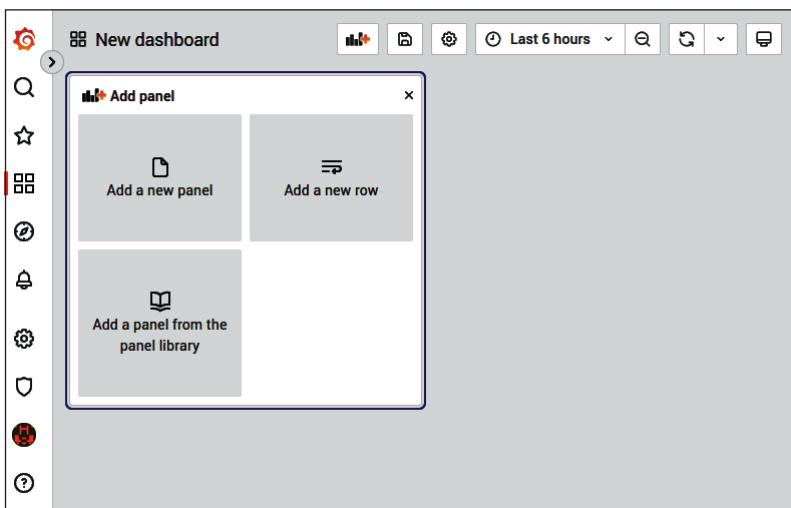


Рис. 6.4 ♦ Новый дашборд Grafana

Здесь можно щелкнуть по ссылке **Add a new panel** (Добавить новую панель) и выбрать первую панель, которую хотите добавить. Панели – это прямоугольные области, содержащие графики, таблицы или другую визуальную информацию. Остальные панели (после первой) можно добавлять щелчком по кнопке **Add panel** (Добавить панель) с оранжевым знаком «плюс». Панели организованы в виде сетки и могут переупорядочиваться перетаскиванием мышью.

! После внесения любых изменений в дашборд или в панели их нужно сохранить. Для этого щелкните по кнопке сохранения в верхней части страницы или нажмите комбинацию клавиш **Ctrl+S**.

Доступ к настройкам дашборда, таким как название, можно произвести, щелкнув по значку шестеренки вверху. С помощью открывшегося меню на-

¹ Если в параметре **Access** (Доступ) выбрано значение **Server** (Серверы), то запросы к серверу Prometheus будет посыпать сама Grafana. Напротив, при непосредственной настройке запросы будут посыпать браузер. Прямая настройка устарела и будет удалена в будущих выпусках Grafana.

строек также можно создать копию дашборда, выбрав пункт **Save As** (Сохранить как), что очень удобно для экспериментов с дашбордом.

Как избежать «стены графиков»

Нередко для каждой службы приходится создавать несколько дашбордов. Как следствие, дашборды постепенно «раздуваются», и в них включается все больше и больше графиков, что затрудняет интерпретацию происходящего. Поэтому, чтобы упростить себе жизнь в будущем, старайтесь создавать дашборды, служащие одной команде или цели.

Чем выше уровень дашборда, тем меньше рядов и панелей в нем должно быть. Дашборд с обзорной информацией должен умещаться на одном экране и быть понятным с первого взгляда. Дашборды, которые обычно используются для оценки ситуации в экстренных случаях, могут иметь на один или два ряда панелей больше, тогда как дашборд для точной настройки работы службы экспертами может занимать несколько экранов.

Почему мы рекомендуем ограничить количество графиков на каждом из дашбордов? Ответ прост: каждый график, каждая линия и каждое число на дашборде увеличивают когнитивную нагрузку и затрудняют понимание отображаемой информации. Это особенно ярко проявляется, когда вы разговариваете по телефону и обрабатываете оповещения. Находясь в состоянии стресса и, возможно, не до конца проснувшись, когда нужно действовать быстро, вы не сможете быстро принимать решения из-за необходимости помнить тонкие нюансы каждого графика на вашем дашборде.

Приведем пример: для одной из служб, над которой работал Брайан, был создан дашборд (единственный), содержащий более 600 графиков¹. Он считался превосходным инструментом мониторинга из-за огромного количества отображаемых данных. Но из-за огромного объема данных в них было трудно разобраться, к тому же этот дашборд загружался довольно долго. Брайан любит называть этот стиль построения дашбордов антипаттерном «Стена графиков».

Имейте в виду, что большое количество графиков не равно хорошему мониторингу. Главное в мониторинге – это конечный результат, такой как быстрое разрешение инцидентов и принятие оптимальных решений, а не красивые графики.

Панель временных рядов

Панель **Time series** (Временные ряды) – это основная панель, которую вы будете использовать. Как следует из названия, она отображает временные ряды. Чтобы добавить панель временных рядов, щелкните по кнопке **Add a new panel** (Добавить новую панель). После этого будет выполнен переход

¹ Еще более тяжелый случай, о котором мы слышали, – дашборд с 1000 графиков.

в режим редактирования новой панели. Чтобы позже вернуться к настройкам, щелкните по кнопке **Panel Title** (Заголовок панели), а затем **Edit** (Редактировать), как показано на рис. 6.5¹.

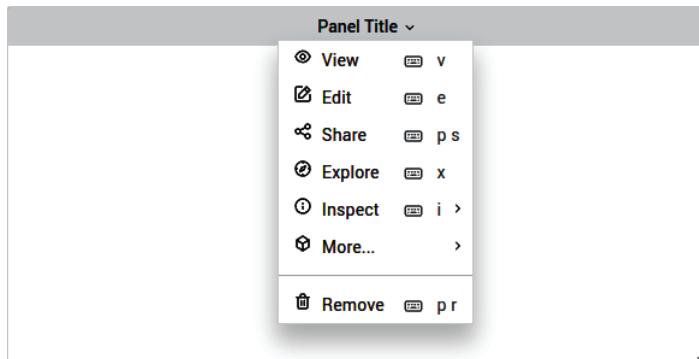


Рис. 6.5 ❖ Переход в режим редактирования панели

Редактор панели откроется на вкладке **Query** (Запрос). В текстовом поле рядом с символом **A**² введите выражение запроса `process_resident_memory_bytes`, как показано на рис. 6.6, а затем щелкните по кнопке **Run queries** (Выполнить запросы). Вы увидите график потребления памяти, подобный изображенному на рис. 2.7 (мы видели его, когда использовали то же самое выражение в браузере выражений).

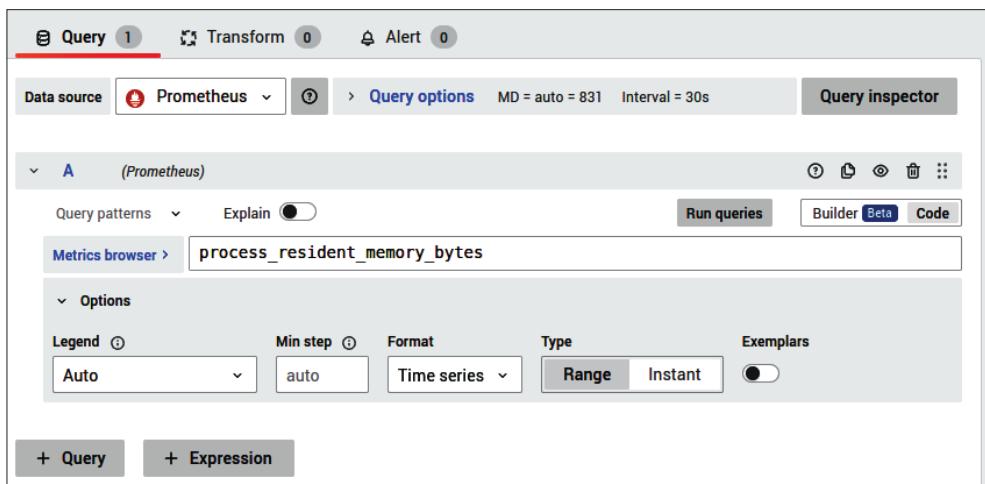


Рис. 6.6 ❖ Выражение `process_resident_memory_bytes` в редакторе графика

¹ Так же можно нажать клавишу **E** на клавиатуре, предварительно наведя указатель мыши на панель. Увидеть полный список доступных сочетаний клавиш можно нажатием клавиши **?**.

² Символ **A** указывает, что это первый запрос.

Grafana предлагает больше возможностей, чем браузер выражений. Вы можете настроить легенду для отображения чего-то другого, кроме полного имени серии. Выберите значение **Custom** (Пользовательский) в раскрывающемся списке **Legend** (Легенда) и введите `{{job}}` в текстовое поле. Справа в разделе **Standard options** (Стандартные параметры) измените единицу измерения на **data/bytes (IEC)** (данные/байты (IEC)). В разделе **Panel options** (Параметры панели) измените значение **Title** (Заголовок) на **Memory Usage**. Теперь график будет выглядеть примерно так, как показано на рис. 6.7, с более понятной легендой, соответствующими единицами измерения по осям и заголовком.

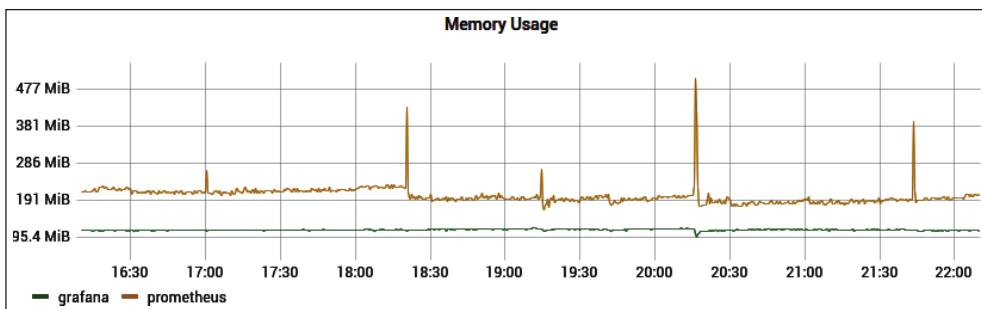


Рис. 6.7 ♦ График «Memory Usage» (Потребление памяти) с настроенными легендой, заголовком и единицами измерения по осям

Эти параметры вы будете настраивать практически во всех своих графиках, но это лишь малая часть того, что можно делать с графиками в Grafana. Также можно настроить цвета, стиль линий, всплывающие подсказки, наложение, заливку и даже включить метрики из нескольких источников данных.

Не забудьте сохранить панель и дашборд, прежде чем продолжить! Щелкните по кнопке **Apply** (Применить), затем сохраните дашборд. **New dashboard** – это специальное название дашборда в Grafana, поэтому выберите какое-нибудь более запоминающееся название.

Элементы управления временем

Возможно, вы заметили элементы управления временем, находящиеся в правом верхнем углу страницы Grafana. По умолчанию выбрано значение **Last 6 hours** (Последние 6 часов). Если щелкнуть на элементах управления временем, то откроется страница, показанная на рис. 6.8, где можно выбрать временной диапазон. В раскрывающемся списке рядом с изображением циферблата со стрелкой (рис. 6.9) можете выбрать частоту обновления. Элементы управления временем применяются ко всему дашборду, однако есть возможность настроить параметры времени для каждой панели отдельно.

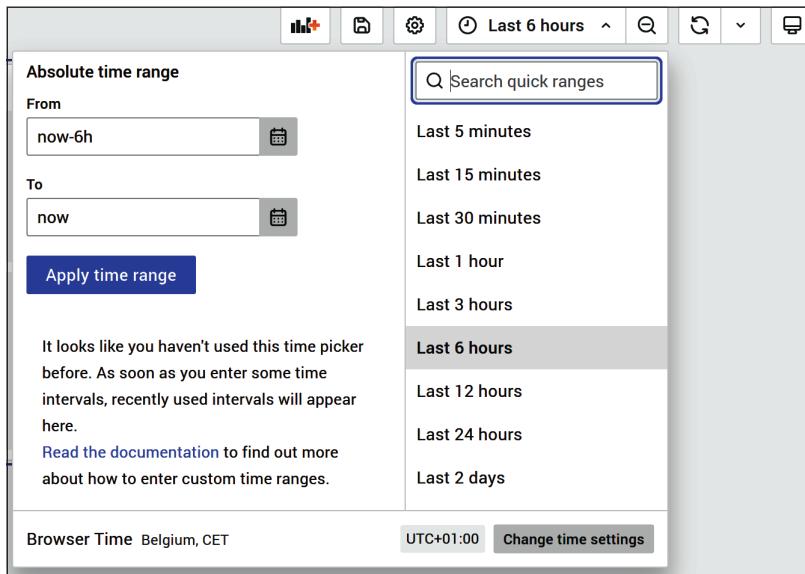


Рис. 6.8 ❖ Меню управления временем в Grafana

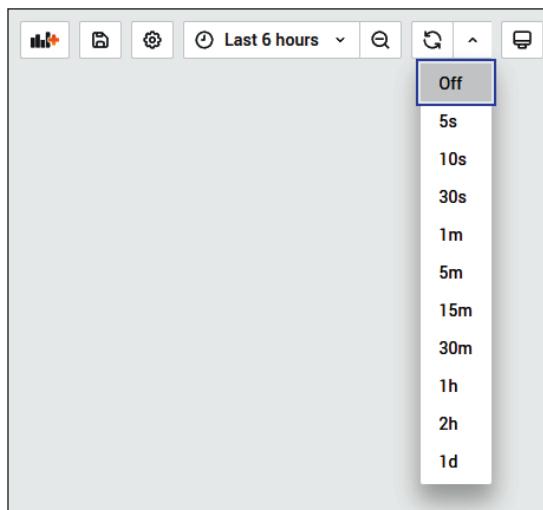


Рис. 6.9 ❖ Меню управления интервалом обновления в Grafana

Эффект искажения

Наблюдая за графиками, иногда можно заметить, что они меняют свою форму, даже притом что базовые данные не изменились. Так проявляется эффект обработки сигналов, называемый *искажением* (aliasing). Возможно, вы уже знакомы с эффектом искажения графики в видеоиграх от первого лица, когда изменяются детали изображения удаленного объекта и могут казаться мерцающими при движении в его сторону.

То же происходит и здесь. Отображение разных данных происходит в разное время, поэтому такие функции, как `rate`, будут возвращать при каждом вычислении несколько разные результаты. Ни один из этих результатов не является ошибочным, это просто разные приближения к происходящему.

Это фундаментальное ограничение мониторинга на основе метрик, наблюдаемое во всех системах, извлекающих данные, и описывается *теоремой выборки Найквиста–Шеннона*. Проблему искажения можно смягчить, увеличив частоту выборки данных и их обработки, но вообще, чтобы получить на 100 % точное представление о происходящем, вам понадобятся журналы, потому что журналы хранят точную информацию о каждом отдельном событии.

Обратите внимание, что в последних версиях Grafana реализованы решения, которые настраивают запросы к Prometheus, чтобы максимально ограничить этот эффект.

Панель статистики

На панели **Stat** (Статистика) отображаются отдельные значения временного ряда. Она также может отображать значения меток Prometheus.

Начнем знакомство с этой панелью с добавления значения временного ряда. Щелкните по кнопке **Apply** (Применить) – стрелка «назад» в правом верхнем углу, – чтобы вернуться из панели **Time series** (Временные ряды) в представление дашборда. Щелкните по кнопке **Add panel** (Добавить панель) и в раскрывающемся списке справа выберите **Stat panel** (Панель статистики)¹. На вкладке **Metrics** (Метрики) введите выражение запроса `prometheus_tsdb_head_series`, которое (грубо говоря) возвращает количество различных временных рядов, принимаемых Prometheus. По умолчанию панель **Stat** (Статистика) вычисляет последнее значение временного ряда за временной диапазон, настроенный в дашборде. Размер символов по умолчанию может показаться мелким, поэтому измените **Font Size** (Размер шрифта), выбрав **200 %**. В разделе **Panel options** (Параметры панели) введите в поле **Title** (Заголовок) текст **Prometheus Time Series**. В разделе **Thresholds** (Пороги) щелкните по изображению корзины рядом с предопределенным порогом 80, чтобы удалить его. Наконец, щелкните по кнопке **Apply** (Применить). В результате вы должны увидеть изображение, похожее на рис. 6.10.

¹ По умолчанию в этом раскрывающемся списке должно быть выбрано значение **Time series** (Временные ряды).

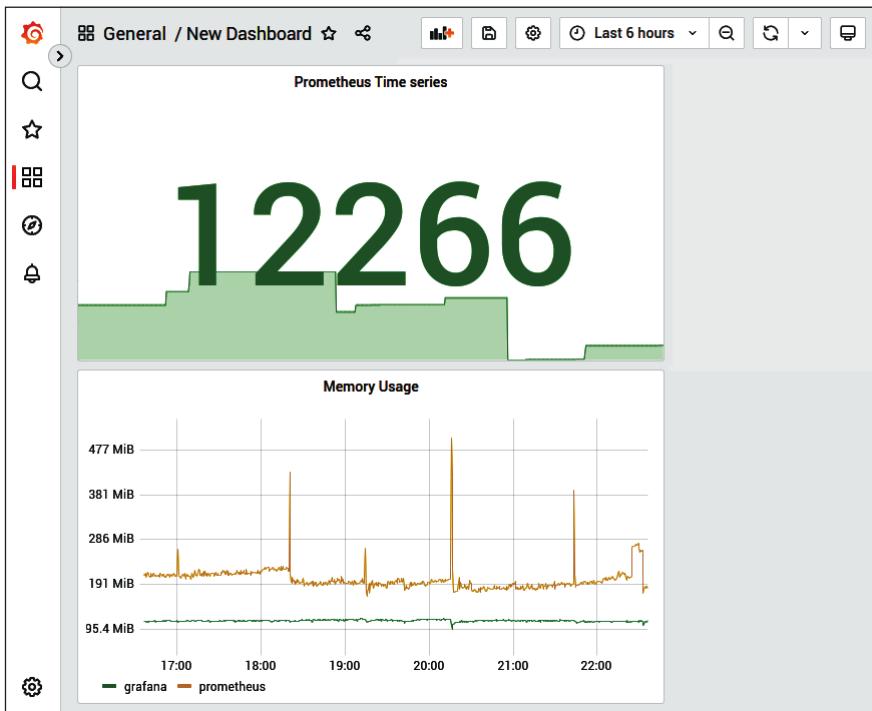


Рис. 6.10 ❖ Дашборд с графиком и панелью статистики

Возможность отображения значений меток удобно использовать для показа версий программного обеспечения на графиках. Добавьте еще одну панель статистики; но на этот раз введите выражение запроса `node_uname_info`, которое возвращает ту же информацию, что и команда `uname -a`. В параметре **Format as** (Форматировать как) установите значение **Table** (Таблица), а в разделе **Value options** (Параметры значения) установите в параметре **Fields** (Поля) значение `release`. В разделе **Panel options** (Параметры панели) укажите заголовок `Kernel version`. После возврата в представление дашборда и переупорядочения панелей с помощью мыши вы должны получить картину, изображенную на рис. 6.11.

Панель **Stat** (Статистика) имеет ряд дополнительных функций, в том числе позволяет настраивать разные цвета для разных значений временного ряда и отображение миниатюрных графиков за значениями.

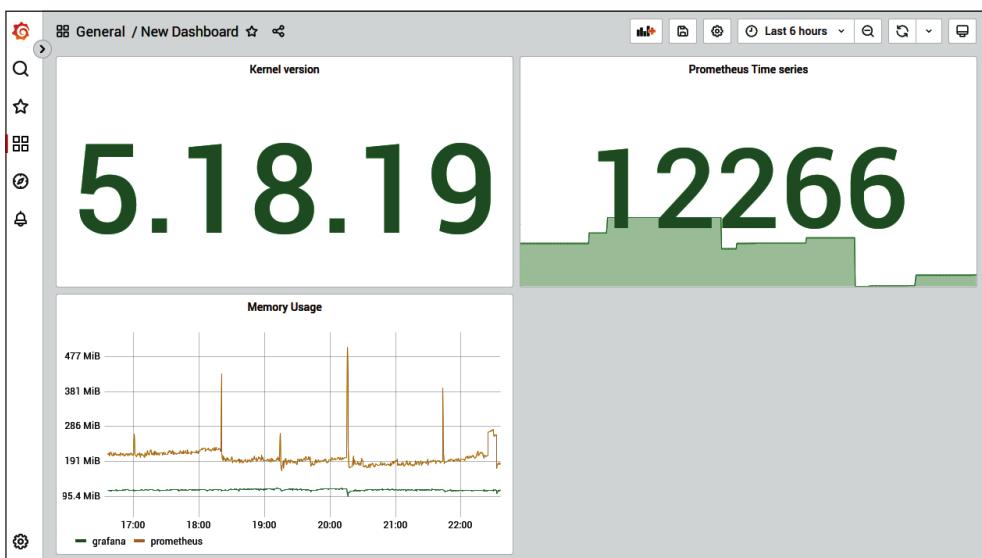


Рис. 6.11 ♦ Дашборд с графиком и с двумя панелями статистики – числовой и текстовой

Панель таблицы

Панель статистики может отображать несколько временных рядов, но каждый уникальный временной ряд занимает довольно много места. Панель **Table** (Таблица) позволяет отображать несколько временных рядов в более компактном виде и предлагает дополнительные возможности, такие как постраничный просмотр. Панели таблиц, как правило, требуют больше усилий по настройке, чем другие панели, и, как правило, оказывают большую когнитивную нагрузку.

Добавьте новую панель, на этот раз **Table panel** (Панель таблицы). Как и раньше, щелкните по кнопке **Add panel** (Добавить панель), а затем **Add a new panel** (Добавить новую панель). Выберите **Table** (Таблица) в раскрывающемся списке справа. Введите выражение запроса `rate(node_network_receive_bytes_total[1m])` на вкладке **Metrics** (Метрики) и измените тип с **Range** (Диапазон) на **Instant** (Мгновенный). В поле **Format** (Формат) выберите **Table** (Таблица).

Количество столбцов в таблице по умолчанию часто больше желаемого. В таком случае перейдите на вкладку **Transform** (Преобразование) и щелкните по кнопке **Organize fields** (Упорядочить поля). Выберите поля, которые хотите скрыть, щелкнув по значку с изображением глаза, как показано на рис. 6.12.

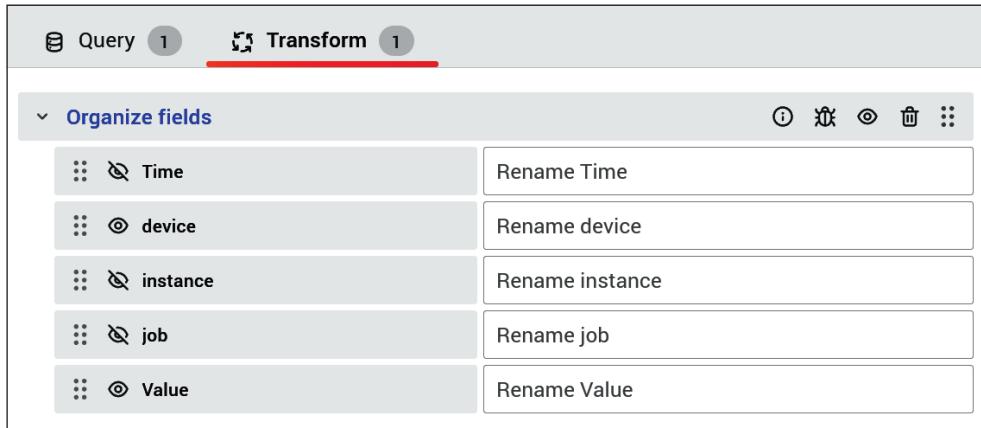


Рис. 6.12 ♦ Скрытие некоторых столбцов

В боковой панели в разделе **Standard options** (Стандартные параметры) установите единицу измерения **bytes/sec (IEC)** (байт/с (IEC)) в разделе **data rate** (Скорость передачи данных). Наконец, в разделе **Panel options** (Параметры панели) введите заголовок **Network Traffic Received**. После всего этого, вернувшись в представление дашборда и переупорядочив панели, вы должны увидеть картину, изображенную на рис. 6.13.

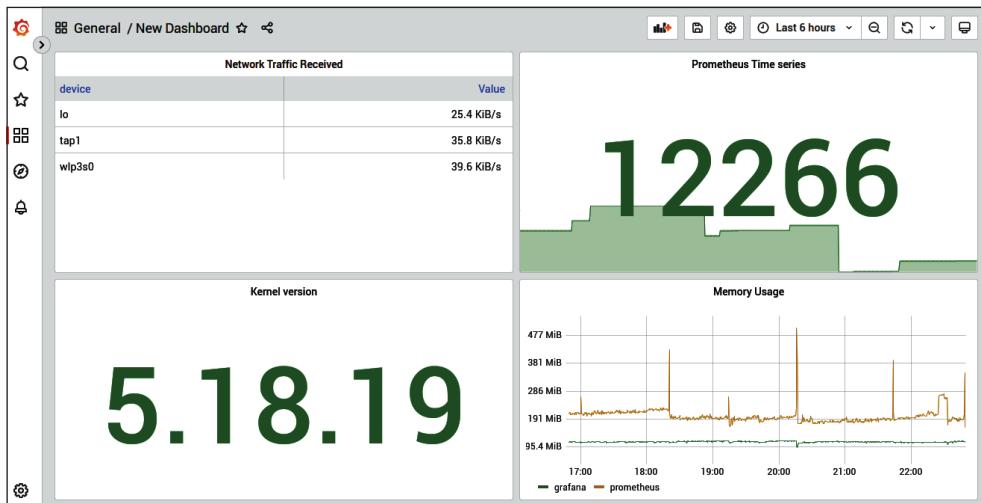


Рис. 6.13 ♦ Дашборд с несколькими панелями, включая таблицу, отображающую трафик по сетевым устройствам

Панель временной шкалы состояний

При визуализации метрик, представляющих состояние, таких как метрики `up`, удобно использовать панель временной шкалы состояния. Она показывает, как дискретное состояние меняется с течением времени.

Давайте используем такую панель для отображения метрик `up`.

Для этого добавим панель временной шкалы состояния. Как и раньше, щелкните по кнопке **Add panel** (Добавить панель), а затем по **Add a new panel** (Добавить новую панель). В раскрывающемся списке справа выберите **State Timeline** (Временная шкала состояний). Вверху на вкладке **Metrics** (Метрики) введите выражение запроса `up`. Настройте легенду: `{{job}} / {{instance}}`.

На боковой панели в разделе **Standard options** (Стандартные параметры) выберите в параметре **Color scheme** (Цветовая схема) значение **Single Color** (Один цвет). В разделе **Value mappings** (Соответствия значений) щелкните по кнопке **Add value mappings** (Добавить соответствия значений) и добавьте два соответствия: значение 1 для отображения текста UP зеленого цвета и значение 2 для отображения текста DOWN красного цвета, как показано на рис. 6.14.

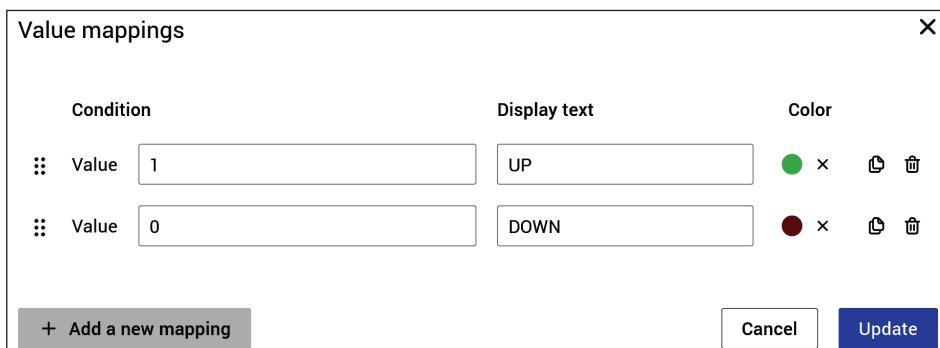


Рис. 6.14 ✦ Добавление двух соответствий значений в панели временной шкалы состояний

На рис. 6.15 показана готовая панель временной шкалы состояний.

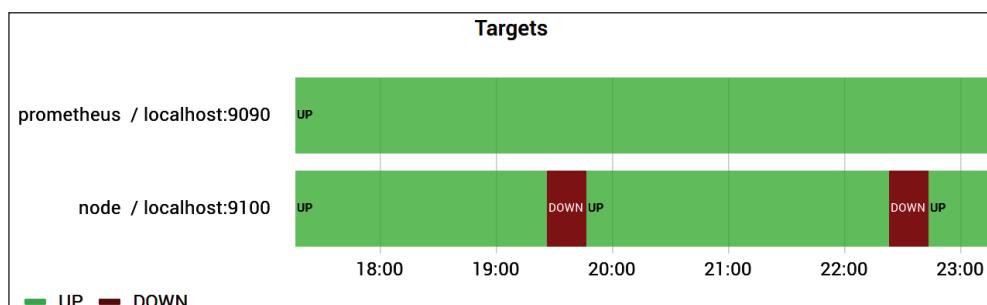


Рис. 6.15 ✦ Готовая панель временной шкалы состояний

Переменные шаблона

Все примеры дашбордов, представленные до сих пор, применялись к одному серверу Prometheus и одному Node Exporter. Такой подход хорош для демонстрации основ, но никуда не годится для мониторинга сотен или даже десятков машин. Впрочем, у нас есть хорошая новость для вас: вам не нужно создавать отдельные дашборды для каждой машины, потому что в Grafana есть возможность использовать шаблоны.

В данный момент у нас настроен мониторинг только одной машины, поэтому с целью демонстрации мы используем шаблоны для сетевых устройств, которых у вас должно быть как минимум два¹.

Для начала создайте новый дашборд, наведя указатель мыши на значок с четырьмя квадратами в боковой панели и выбрав пункт **+New dashboard** (+Новый дашборд), как показано на рис. 6.16.

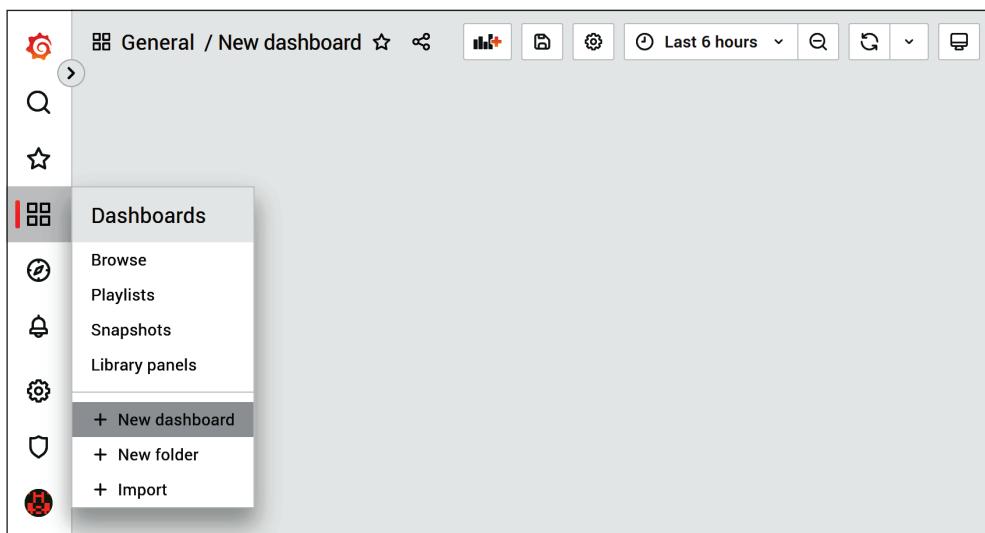


Рис. 6.16 ♦ Меню с кнопкой создания нового дашборда

Щелкните по значку с изображением шестеренки вверху, а затем выберите пункт **Variables** (Переменные)². Щелкните **+Add variable** (+Добавить переменную), чтобы добавить переменную шаблона. В поле **Name** (Имя) введите имя **Device**, в поле **Data source** (Источник данных) выберите **Prometheus** и в поле **Refresh** выберите период обновления **On time range change** (При изменении диапазона времени). Введите запрос **label_values(node_network_receive_bytes_total, device)**³. Форма с настройками должна выглядеть,

¹ Петлевое устройство (loopback) и проводной и/или WiFi-интерфейс.

² В предыдущих версиях Grafana этот пункт назывался **Templating** (Шаблоны).

³ Обратите внимание, что это не запрос PromQL. **label_values** – это функция Grafana, доступная только в шаблонах.

как показано на рис. 6.17. Щелкните по кнопке **Update** (Добавить), чтобы добавить переменную.

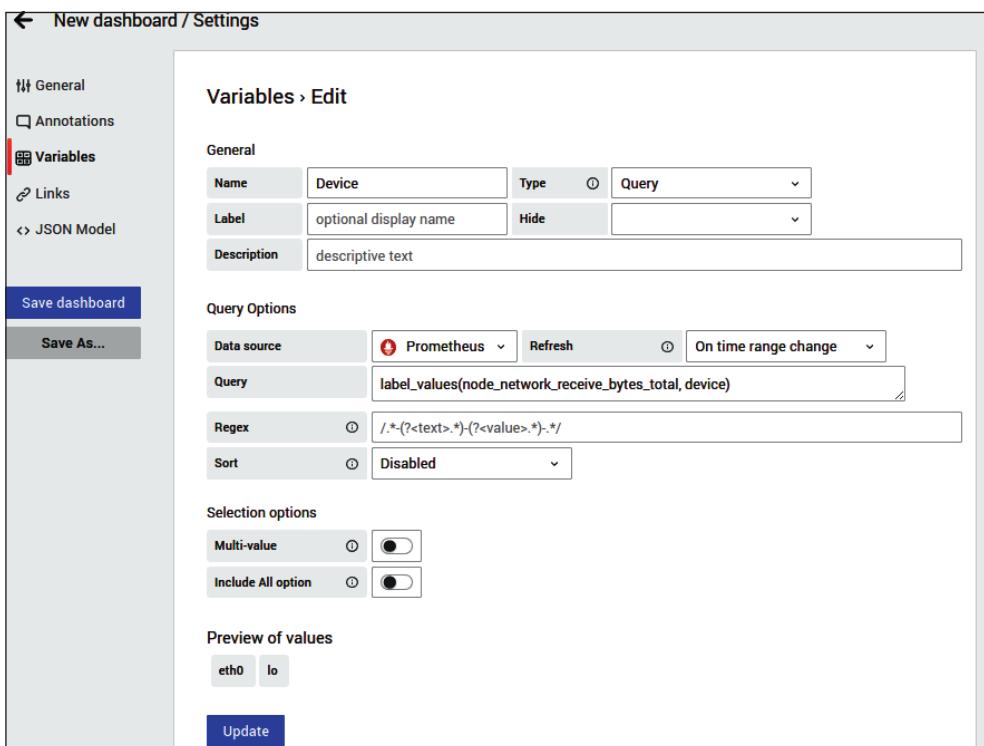


Рис. 6.17 ♦ Добавление переменной шаблона Device в дашборд Grafana

После этого вы увидите, что появилось новое раскрывающееся меню для выбора значения переменной, как показано на рис. 6.18.

Теперь нужно задействовать переменную. Щелкните по значку с изображением крестика (X), чтобы закрыть раздел **Variable** (Переменная), затем щелкните по значку с тремя точками и добавьте новую панель **Time series** (Временные ряды). Введите выражение запроса `rate(node_network_receive_bytes_total{device=\"$Device\"}[$__rate_interval])`, в которое взамен \$Device будет подставляться фактическое значение переменной шаблона. Если вы установили флагок **Multi-value** (Множественное значение), то используйте регулярное выражение `device=~"$Device"`, так как в этом случае переменная будет представлена регулярным выражением. Также регулярные выражения следует использовать в случае сложных значений, поскольку Grafana все равно попытается их экранировать. В параметре **Legend Format** (Формат легенды) установите значение **Custom** (Пользовательский) и введите формат `{device}`, в параметре **Title** (Заголовок) введите заголовок **Bytes Received**, а в параметре **Unit** (Единица измерения) в разделе **data rate** (Скорость передачи данных) выберите значение **bytes/sec** (байт/с).

i Как вы наверняка обратили внимание, мы используем `$__rate_interval` в выражении PromQL. Это функция Grafana, которая выбирает лучший интервал в зависимости от интервала выборки данных, установленного в настройках источника данных, и других параметров, таких как шаг в настройках панели. Если просматривать данные за 24 ч, то значение `$__rate_interval` будет больше, чем при просмотре только за последний час.

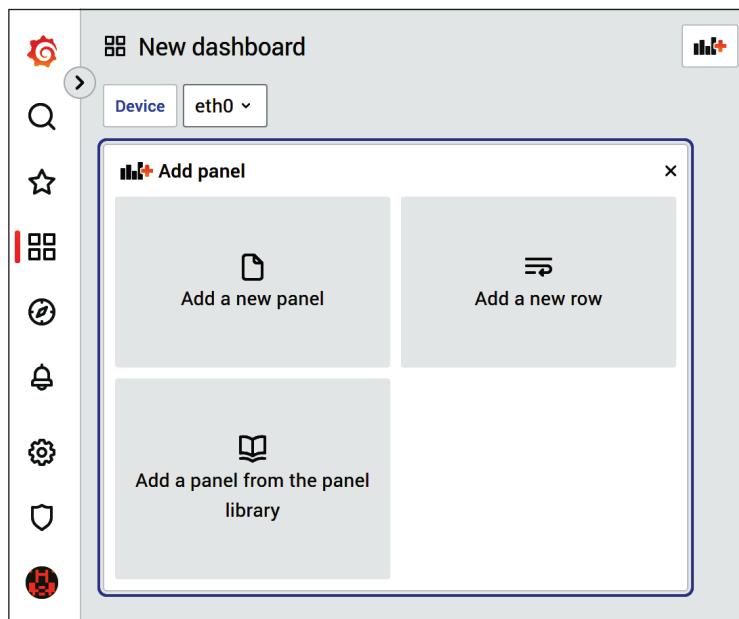


Рис. 6.18 ❖ Раскрывающееся меню для выбора значения переменной Device

Щелкните по кнопке **Apply** (Применить) и затем по заголовку панели, на этот раз выберите пункт **More** (Дополнительно), а затем **Duplicate** (Дублировать). Эта последовательность действий создаст копию существующей панели. В настройках этой новой панели введите выражение запроса `rate(node_network_transmit_bytes_total {device=~"$Device"})[$__rate_interval]`, а в поле **Title** (Заголовок) введите **Bytes Transmit**. Теперь в дашборде будут присутствовать панели, отображающие количество байтов, переданных в обоих направлениях, как показано на рис. 6.19. А кроме того, вы сможете выбрать в раскрывающемся списке сетевое устройство для просмотра.

В реальном мире вы, скорее всего, будете создавать шаблоны на основе метки `instance` и отображать сразу все метрики с одной машины, относящиеся к сети. Более того, на одном дашборде у вас может быть определено несколько переменных. Вот как может работать общий дашборд, отображающий работу сборщика мусора в Java: одна переменная для метки `job`, одна для метки `instance` и одна для выбора источника данных `Prometheus`.

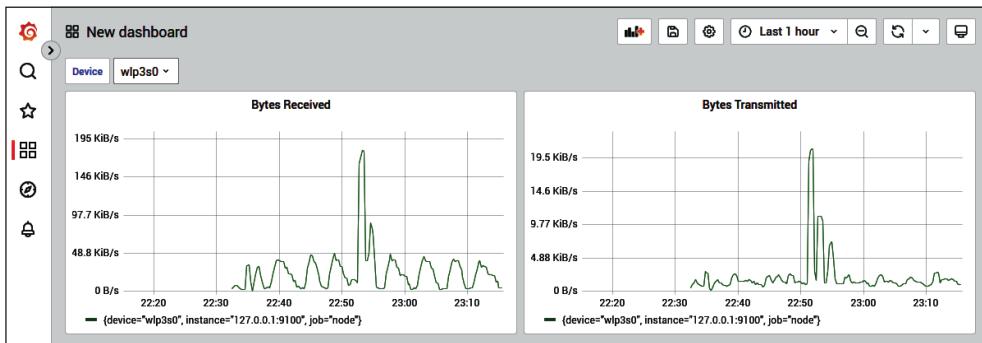


Рис. 6.19 ♦ Простой дашборд, отображающий объем сетевого трафика с помощью переменной шаблона

Возможно, вы заметили, что при изменении значения переменной меняются параметры URL. Они также меняются при использовании элементов управления временем. Эта возможность позволяет обмениваться с коллегами ссылками на дашборд с правильными значениями переменных, как показано в разделе «Шаблоны уведомлений» главы 19. В верхней части страницы есть значок **Share dashboard** (Поделиться дашбордом), с помощью которого можно создавать URL и моментальные снимки дашбордов. Моментальные снимки идеально подходят для анализа после аварий и составления отчетов о сбоях, когда требуется сохранить внешний вид дашборда.

В следующей главе мы подробнее остановимся на Node Exporter и некоторых метриках, которые он предлагает.

Часть



МОНИТОРИНГ ИНФРАСТРУКТУРЫ

Весь мир (пока) не вращается вокруг Prometheus и не предоставляет метрики Prometheus из коробки. Поэтому широкое распространение получили экспортеры – инструменты, позволяющие преобразовывать метрики из других систем в формат, понятный Prometheus.

В главе 7 подробно рассматривается один из первых экспортеров, который вы, скорее всего, будете использовать, – Node Exporter.

В главе 8 вы узнаете, как Prometheus выясняет, как и откуда брать метрики.

Глава 9 посвящена мониторингу контейнерных технологий, таких как Docker и Kubernetes.

В экосистеме Prometheus имеются буквально сотни экспортеров. В главе 10 мы расскажем, как использовать некоторые типичные экспортеры.

У многих из вас уже работает какая-то другая система мониторинга на основе метрик, и в главе 11 мы посмотрим, как можно интегрировать ее с Prometheus.

Экспортеры не появляются из ниоткуда. Если нужного экспортера не существует, то его можно создать, о чем мы и поговорим в главе 12.

Глава 7

Node Exporter

Node Exporter¹ – это один из первых экспортёров, которые вы будете использовать в своей практике, как уже было показано в главе 2. Он предоставляет метрики уровня машины, в основном из ядра операционной системы, такие как нагрузка на процессор, объём свободной/занятой памяти, свободного/занятого дискового пространства, дискового и сетевого ввода-вывода, температура материнской платы и т. д. Node Exporter используется с системами Unix. Пользователи Windows должны вместо него использовать Windows Exporter² (<https://oreil.ly/mVsAX>).

Node Exporter предназначен только для мониторинга машины, а не отдельных процессов или служб, выполняющихся на ней. Другие системы мониторинга часто имеют то, что нам нравится называть *суперагентом*, – единый процесс, следящий за всем происходящим на машине. В архитектуре Prometheus каждая из ваших служб должна будет предоставлять свои метрики, используя экспортёр при необходимости, а Prometheus будет извлекать метрики из него. Это позволяет отказаться от суперагента, создающего узкое место с точки зрения производительности, а также даёт возможность рассуждать в терминах динамических служб, а не машин.

Рекомендации по созданию метрик методом прямого инструментирования, как те, что обсуждались в разделе «Какие имена выбирать для метрик?» главы 3, относительно ясны. Но они не относятся к экспортёрам, которые по определению извлекают данные из источника, разработанного без учета рекомендаций Prometheus. В зависимости от объема и качества метрик разработчикам экспортёров приходится искать компромиссы между трудозатратами и получением идеальных метрик.

ОС Linux, например, предлагает тысячи метрик. Некоторые из них хорошо документированы и понятны, например нагрузка на процессор, другие, такие как потребление памяти, менялись от версии к версии ядра по мере изменения реализации. Можно даже найти вообще недокументированные метрики, для использования которых может понадобиться углубиться в изучение исходного кода ядра.

¹ Node Exporter не имеет ничего общего с Node.js; слово Node в его названии означает «вычислительный узел».

² Раньше Windows Exporter назывался WMI Exporter.

Node Exporter предназначен для запуска от имени пользователя без полномочий root и должен запускаться непосредственно на машине так же, как запускаются системные демоны, такие как sshd или cron.

! Запустить Node Exporter в Docker¹ возможно, но для этого потребуется использовать некоторые тома и указывать определенные параметры командной строки (`--path.procfs`, `--path.rootfs`, `--path.sysfs`) для мониторинга файловой системы хоста внутри контейнера. Если возможно, запускайте Node Exporter как службу на узле без Docker. Docker пытается изолировать контейнер от хоста, что мешает работе Node Exporter, который пытается добраться до метрик этого хоста.

В отличие от большинства других экспортёров, из-за большого разнообразия метрик, доступных в операционных системах, Node Exporter позволяет определить, какие категории метрик он должен извлекать. Это можно сделать с помощью флагов командной строки, таких как `--collector.wifi`, который активирует сборщик метрик WiFi, и `--no-collector.wifi`, отключающий его. Флаг `--collector.disable-defaults` отключает все сборщики метрик кроме тех, которые включены явно с помощью флагов командной строки. Настройки по умолчанию имеют разумные значения, поэтому вам не придется беспокоиться о них при запуске.

Разные ядра предоставляют разные метрики, так как, например, Linux и FreeBSD работают по-разному. Метрики могут перемещаться между сферами ответственности разных сборщиков с течением времени по мере развития Node Exporter. Если вы используете другую систему Unix, то обнаружите, что предлагаемые метрики и сборщики различаются.

В этой главе мы опишем некоторые ключевые метрики, которые Node Exporter 1.4.0 может извлекать из ядра Linux 5.18.0. Этот список нельзя назвать исчерпывающим. Как и в случае с большинством экспортёров и приложений, обязательно просмотрите список метрик, доступный в пути `/metrics`, чтобы получить представление о том, что доступно. Дополнительно можете попробовать запустить примеры выражений PromQL, представленные в главе 2.

Сборщик информации о процессоре

Основная метрика, предлагаемая сборщиком информации о процессоре, – `node_cpu_seconds_total`. Это счетчик, показывающий, сколько времени каждый процессор работал в каждом режиме. Доступные метки: `cput` и `mode`:

```
# HELP node_cpu_seconds_total Seconds the CPUs spent in each mode.
# TYPE node_cpu_seconds_total counter
node_cpu_seconds_total{cpu="0",mode="idle"} 13024.48
node_cpu_seconds_total{cpu="0",mode="iowait"} 9.53
```

¹ Docker – это платформа для разработчиков и системных администраторов, позволяющая создавать, упаковывать и развертывать приложения в контейнерах. Контуер – это легковесная, переносимая и самодостаточная технология упаковки, позволяющая развертывать приложения и их зависимости как единое целое.

```

node_cpu_seconds_total{cpu="0",mode="irq"} 0
node_cpu_seconds_total{cpu="0",mode="nice"} 0.11
node_cpu_seconds_total{cpu="0",mode="softirq"} 109.74
node_cpu_seconds_total{cpu="0",mode="steal"} 0
node_cpu_seconds_total{cpu="0",mode="system"} 566.67
node_cpu_seconds_total{cpu="0",mode="user"} 1220.36
node_cpu_seconds_total{cpu="1",mode="idle"} 13501.28
node_cpu_seconds_total{cpu="1",mode="iowait"} 5.96
node_cpu_seconds_total{cpu="1",mode="irq"} 0
node_cpu_seconds_total{cpu="1",mode="nice"} 0.09
node_cpu_seconds_total{cpu="1",mode="softirq"} 23.74
node_cpu_seconds_total{cpu="1",mode="steal"} 0
node_cpu_seconds_total{cpu="1",mode="system"} 423.84
node_cpu_seconds_total{cpu="1",mode="user"} 936.05

```

Для каждого процессора сумма значений режимов будет увеличиваться на одну секунду за секунду. Это позволяет рассчитать долю времени простоя для всех процессоров с помощью выражения PromQL:

```
avg without(cpu, mode)(rate(node_cpu_seconds_total{mode="idle"}[1m]))
```

Это выражение вычисляет время простоя в секунду для каждого процессора, а затем усредняет его по всем имеющимся процессорам.

Это выражение можно обобщить, чтобы получить долю времени в каждом режиме для машины:

```
avg without(cpu)(rate(node_cpu_seconds_total[1m]))
```

Потребление процессора гостевыми системами (т. е. виртуальными машинами, работающими под управлением ядра) уже включено в режимы `user` и `nice`. Увидеть, сколько процессорного времени было потреблено гостевыми системами, можно с помощью метрики `node_cpu_guest_seconds_total`.

Сборщик информации о файловой системе

Сборщик информации о файловой системе, как нетрудно догадаться, собирает сведения о смонтированных файловых системах, как если бы вы получили их с помощью команды `df`. Флаги `--collector.filesystem.mount-points-exclude` и `--collector.filesystem.fs-types-exclude` позволяют ограничить файловые системы для мониторинга (по умолчанию из мониторинга исключаются различные псевдофайловые системы). Поскольку Node Exporter не предполагает наличие привилегий `root`, вы должны убедиться, что права доступа к файлам позволяют ему использовать системный вызов `statfs` для интересующих вас точек монтирования.

Все метрики, поставляемые этим сборщиком, имеют префикс `node_file-system_` и метки `device`, `fstype` и `mountpoint`:

```
# HELP node_filesystem_size_bytes Filesystem size in bytes.  
# TYPE node_filesystem_size_bytes gauge  
node_filesystem_size_bytes{device="/dev/sda5",fstype="ext4",mountpoint="/"}
```

Подавляющее большинство метрик файловой системы не требуют пояснений. Единственный нюанс, которой вы должны знать, – разница между `node_filesystem_avail_bytes` и `node_filesystem_free_bytes`. В файловых системах Unix некоторый объем дискового пространства зарезервирован для пользователя `root`, чтобы он имел возможность что-то сделать, когда пользователи заполнят все доступное пространство. `node_filesystem_avail_bytes` – это объем дискового пространства, доступного пользователям, и при попытке подсчитать используемое дисковое пространство вы должны, соответственно, использовать:

```
node_filesystem_avail_bytes  
/  
node_filesystem_size_bytes
```

`node_filesystem_files` и `node_filesystem_files_free` сообщают общее количество индексных узлов (`inode`) и сколько из них свободно, что примерно соответствует количеству файлов в файловой системе. Эту информацию можно также получить с помощью `df -i`.

Сборщик дисковой статистики

Сборщик дисковой статистики предоставляет метрики дискового ввода-вывода из `/proc/diskstats`. По умолчанию флаг `--collector.diskstats.device-exclude` исключает объекты, не являющиеся настоящими дисками, такие как разделы и петлевые устройства:

```
# HELP node_disk_io_now The number of I/Os currently in progress.  
# TYPE node_disk_io_now gauge  
node_disk_io_now{device="sda"} 0
```

Все метрики имеют метку `device`, и почти все они являются счетчиками, как показано ниже.

`node_disk_io_now`

Количество операций ввода-вывода, выполняемых в данный момент.

`node_disk_io_time_seconds_total`

Увеличивается, когда выполняется операция ввода-вывода.

`node_disk_read_bytes_total`

Количество байтов, прочитанных операцией ввода-вывода.

`node_disk_read_time_seconds_total`

Время, затрачиваемое операциями чтения.

`node_disk_reads_completed_total`

Количество полностью выполненных операций чтения.

`node_disk_written_bytes_total`

Количество байтов, записанных операцией ввода-вывода.

`node_disk_write_time_seconds_total`

Время, затрачиваемое операциями записи.

`node_disk_writes_completed_total`

Количество полностью выполненных операций записи.

В основном они означают именно то, что вы думаете, тем не менее загляните в документацию ядра¹, чтобы получить более подробную информацию.

Метрику `node_disk_io_time_seconds_total` можно использовать для получения информации об интенсивности дискового ввода-вывода, которую можно получить также с помощью `iostat -x`:

```
rate(node_disk_io_time_seconds_total[1m])
```

Рассчитать среднее время чтения ввода-вывода можно с помощью:

```
rate(node_disk_read_time_seconds_total[1m])
/
rate(node_disk_reads_completed_total[1m])
```

Сборщик информации о сетевых устройствах

Сборщик информации о сетевых устройствах предоставляет метрики с префиксом `node_network_` и меткой `device`:

```
# HELP node_network_receive_bytes_total Network device statistic receive_bytes.
# TYPE node_network_receive_bytes_total counter
node_network_receive_bytes_total{device="lo"} 8.3213967e+07
node_network_receive_bytes_total{device="wlan0"} 7.0854462e+07
```

`node_network_receive_bytes_total` и `node_network_transmit_bytes_total` – это основные метрики, которые будут вам интересны, потому что с их помощью можно рассчитать пропускную способность сети:

```
rate(node_network_receive_bytes_total[1m])
```

Вас также могут заинтересовать метрики `node_network_receive_packets_total` и `node_network_transmit_packets_total`, подсчитывающие входящие и исходящие пакеты соответственно.

¹ Размер сектора в `/proc/diskstats` всегда равен 512 байт; вам не нужно беспокоиться, если ваши диски используют секторы большего размера. Это пример того, что можно выяснить, только заглянув в исходный код ядра Linux.

Сборщик информации о потреблении памяти

Сборщик информации о потреблении памяти предоставляет все стандартные метрики, связанные с памятью, с префиксом `node_memory_`. Все они извлекаются из `/proc/meminfo`, и это первый сборщик, семантика которого может показаться неясной. Сборщик конвертирует килобайты в байты, но все же для большей ясности следует заглянуть в документацию (<https://oreil.ly/F-0JW>) и воспользоваться своими знаниями особенностей внутренних компонентов Linux, чтобы понять, что означают эти метрики:

```
# HELP node_memory_MemTotal_bytes Memory information field MemTotal.  
# TYPE node_memory_MemTotal_bytes gauge  
node_memory_MemTotal_bytes 3.285016576e+10
```

Например, `node_memory_MemTotal_bytes` – это общий¹ объем физической памяти в машине – красиво и очевидно. Но обратите внимание, что нет метрики с используемым объемом памяти, поэтому вам придется вычислять ее по значениям других метрик.

`node_memory_MemFree_bytes` – это объем свободной, ничем не используемой памяти, но это не значит, что это вся память, которую можно выделить. Теоретически можно дополнительно освободить кеш страниц (`node_memory_Cached_bytes`) и буферы записи (`node_memory_Buffers_bytes`), что, впрочем, может отрицательно сказаться на производительности некоторых приложений². Кроме того, существуют различные другие структуры ядра, использующие память, такие как таблицы страниц.

`node_memory_MemAvailable` – это эвристика ядра, определяющая, сколько памяти действительно доступно, но она была добавлена только в версии Linux 3.14. Если вы используете достаточно новое ядро, то эту метрику можно использовать для приближения момента исчерпания памяти.

Сборщик информации об аппаратной части

Сборщик информации об аппаратной части предоставляет метрики с температурой и скоростью вращения вентиляторов с префиксом `node_hwmon_`. Это также самая информация, которую можно получить с помощью команды `sensor`:

```
# HELP node_hwmon_sensor_label Label for given chip and sensor  
# TYPE node_hwmon_sensor_label gauge
```

¹ Почти.

² Prometheus 2.0, например, – одно из приложений, которые активно пользуются кешем страниц.

```

node_hwmon_sensor_label{chip="platform_coretemp_0",
    label="core_0",sensor="temp2"} 1
node_hwmon_sensor_label{chip="platform_coretemp_0",
    label="core_1",sensor="temp3"} 1
# HELP node_hwmon_temp_celsius Hardware monitor for temperature (input)
# TYPE node_hwmon_temp_celsius gauge
node_hwmon_temp_celsius{chip="platform_coretemp_0",sensor="temp1"} 42
node_hwmon_temp_celsius{chip="platform_coretemp_0",sensor="temp2"} 42
node_hwmon_temp_celsius{chip="platform_coretemp_0",sensor="temp3"} 41

```

`node_hwmon_temp_celsius` – температура различных компонентов, которые также могут иметь метки сенсоров¹, отображаемые в `node_hwmon_sensor_label`.

Для анализа метрик на некоторых² машинах может понадобиться метка сенсора, чтобы понять, что это за сенсор. В предыдущих метриках `temp3` представляет ядро процессора с номером 1.

Вы можете добавить метку `label` из `node_hwmon_sensor_label` в `node_hwmon_temp_celsius`, используя функцию `group_left`, которая подробно обсуждается в разделе «Многие к одному и `group_left`» главы 15:

```

node_hwmon_temp_celsius
* ignoring(label) group_left(label)
node_hwmon_sensor_label

```

Сборщик статистики

Сборщик статистики предлагает смесь разных метрик из `/proc/stat`³.

`node_boot_time_seconds` – это время запуска ядра. Используя эту метрику, можно рассчитать, как долго работает ядро:

```
time() - node_boot_time_seconds
```

`node_intr_total` сообщает количество имевших место аппаратных прерываний. Существует также похожая метрика `node_interrupts_total`, предоставляемая сборщиком информации о прерываниях, который по умолчанию отключен из-за высокой кардинальности.

Остальные метрики относятся к процессам. `node_forks_total` – счетчик количества выполненных системных вызовов `fork`, `node_context_switches_total` – количество переключений контекста, а `node_procs_blocked` и `node_procs_running` – количество заблокированных или выполняющихся процессов.

¹ Слово *метки* здесь не означает метки Prometheus. Это метки сенсоров, и они берутся из таких файлов, как `/sys/devices/platform/coretemp.0/hwmon/hwmon1/temp3_label`.

² Например, наш ноутбук, где была получена предыдущая метрика.

³ Раньше он также предоставлял метрики процессора, которые теперь извлекаются сборщиком информации о процессоре.

Сборщик информации об имени узла

Сборщик информации о версии системы предоставляет единственную метрику – `node_uname_info`, которую мы уже видели в разделе «Панель статистики» главы 6:

```
# HELP node_uname_info Labeled system information as provided by the uname
#      system call.
# TYPE node_uname_info gauge
node_uname_info{domainname="(none)",machine="x86_64",nodename="kozo",
               release="4.4.0-101-generic",sysname="Linux",
               version="#124-Ubuntu SMP Fri Nov 10 18:29:59 UTC 2017"} 1
```

Метка `nodename` – это имя хоста, которое может отличаться от целевой метки `instance` (см. раздел «Целевые метки» главы 8) или любых других имен, например DNS, которые могут присваиваться узлу.

Вот как можно подсчитать, сколько машин работает с каждой версией ядра:

```
count by(release)(node_uname_info)
```

Сборщик информации об операционной системе

Сборщик информации об операционной системе предоставляет две метрики: `node_os_info` и `node_os_version`. Они содержат информацию об операционной системе:

```
# HELP node_os_info A metric with a constant '1' value labeled by
#      build_id, id, id_like, image_id, image_version, name,
#      pretty_name, variant, variant_id, version, version_codename,
#      version_id.
# TYPE node_os_info gauge
node_os_info{build_id="22.05.20220912.bf014ca",id="nixos",
             id_like="",image_id="",image_version="",name="NixOS",
             pretty_name="NixOS 22.05 (Quokka)",variant="",
             variant_id="",version="22.05 (Quokka)",
             version_codename="quokka",version_id="22.05"} 1
# HELP node_os_version Metric containing the major.minor
#      part of the OS version.
# TYPE node_os_version gauge
node_os_version{id="nixos",id_like:"",name="NixOS"} 22.05
```

Вот как можно подсчитать, сколько машин работает с каждой версией дистрибутива:

```
count by(name, version)(node_os_info)
```

Сборщик информации о средней нагрузке

Сборщик информации о средней нагрузке предоставляет средние значения нагрузки за 1, 5 и 15 мин в виде метрик `node_load1`, `node_load5` и `node_load15` соответственно.

Значения этих метрик зависят от платформы и могут не соответствовать вашим представлениям. Например, в Linux это не только количество процессов, ожидающих выполнения в очереди, но также непрерываемые процессы, которые ожидают завершения операции ввода-вывода.

- i** В системах с достаточно свежим ядром мы рекомендуем использовать сборщик информации о давлении, описанный в разделе «Сборщик информации о давлении» ниже.

Средние значения нагрузки могут помочь при выяснении причин увеличения нагрузки в текущее время, но они не лучший выбор для организации рассылки оповещений. Для более полного знакомства с ними мы рекомендуем прочитать статью Брендана Грегга (Brendan Gregg) «Linux Load Averages: Solving the Mystery» в его блоге (<https://oreil.ly/JVKfd>).

Это бессмысленное число, но люди почему-то считают его важным.

– Комментарий в файле `loadavg.c` ядра Linux

Сборщик информации о давлении

Подсистема сбора информации об остановке под давлением (Pressure Stall Information, PSI) была добавлена в ядро Linux в версии 4.20. Эти метрики измеряют нагрузку на три ресурса: процессор, память и ввод-вывод. Эту подсистему необходимо включить в ядре во время компиляции.

Ядро может быть собрано с поддержкой PSI, но сбор этой информации обычно отключен по умолчанию. В таком случае вы можете передать ядру параметр `psi=1` во время загрузки, чтобы включить ее.

Сборщик PSI предоставляет пять метрик:

```
# HELP node_pressure_cpu_waiting_seconds_total
    Total time in seconds that processes have waited for CPU time
# TYPE node_pressure_cpu_waiting_seconds_total counter
node_pressure_cpu_waiting_seconds_total 113.6605130
# HELP node_pressure_io_stalled_seconds_total
    Total time in seconds no process could make progress due to IO congestion
# TYPE node_pressure_io_stalled_seconds_total counter
node_pressure_io_stalled_seconds_total 8.630361
# HELP node_pressure_io_waiting_seconds_total
    Total time in seconds that processes have waited due to IO congestion
# TYPE node_pressure_io_waiting_seconds_total counter
node_pressure_io_waiting_seconds_total 9.609997
```

```
# HELP node_pressure_memory_stalled_seconds_total
    Total time in seconds no process could make progress
# TYPE node_pressure_memory_stalled_seconds_total counter
node_pressure_memory_stalled_seconds_total 0
# HELP node_pressure_memory_waiting_seconds_total
    Total time in seconds that processes have waited for memory
# TYPE node_pressure_memory_waiting_seconds_total counter
node_pressure_memory_waiting_seconds_total 0
```

Метрики `waiting` сообщают общее количество секунд, в течение которых некоторые задачи находились в ожидании, а метрики `stalled` – количество секунд, когда все задачи были приостановлены из-за нехватки ресурсов. Для памяти и ввода-вывода имеются оба вида метрик – `waiting` и `stalled`, а для процессора – только `waiting`. Это связано с тем, что процессор всегда выполняет какой-либо.

Поскольку эти метрики являются счетчиками, их можно использовать с функцией `rate()`, чтобы определить, не слишком ли высока нагрузка на некоторые ресурсы:

```
rate(node_pressure_memory_waiting_seconds_total[1m])
```

Сборщик текстовых файлов

Сборщик текстовых файлов несколько отличается от других сборщиков, которые мы уже описали. Он получает метрики не из ядра, а из файлов, которые вы создаете.

Node Exporter не предназначен для запуска от имени пользователя `root`. Однако для получения некоторых метрик, например предлагаемых технологией SMART¹, требуются привилегии `root` для запуска команды `smartctl`.

В дополнение к метрикам, для извлечения которых требуются привилегии `root`, можно получить некоторую информацию, просто запустив команду, такую как `iptables`. Для надежности сам Node Exporter не запускает процессы.

Чтобы использовать сборщик текстовых файлов, нужно создать задание `cron`, которое регулярно запускает такие команды, как `smartctl` или `iptables`, преобразует их вывод в текстовый формат Prometheus и атомарно записывает его в файл в предопределенном каталоге. В каждом цикле выборки метрик Node Exporter будет читать файлы в этом каталоге и включать их метрики в свои выходные данные.

 Сервер Prometheus не может читать текстовые файлы напрямую, поэтому нужно дополнительное программное обеспечение для экспортирования файла по протоколу HTTP. Для этого можно использовать любой HTTP-сервер, но лучше доверить это Node Exporter, потому что он также проверяет правильность метрик и может экспортировать метрики, поступающие из нескольких файлов.

¹ Технология самоконтроля, анализа и отчетности (Self-Monitoring, Analysis, and Reporting Technology, SMART) поставляет метрики с жестких дисков, которые могут пригодиться для прогнозирования и обнаружения сбоев.

Этот сборщик можно использовать для добавления своих собственных метрик, собираемых заданиями cron, или дополнительной статической информации из файлов, записанных системой управления конфигурацией вашей машины (эта возможность обсуждается в разделе «Информационные метрики» главы 5), такой как присвоенные машине роли Chef¹.

Как и сам Node Exporter, сборщик текстовых файлов предназначен для экспортации метрик о машине. Например, с его помощью можно экспортить метрику ядра, которую Node Exporter еще не предоставляет или для доступа к которой требуются привилегии root. Вам также может понадобиться отслеживать дополнительные метрики уровня операционной системы, например наличие пакетов, ожидающих обновления, или необходимость перезагрузки. Хотя технически это – служба, а не метрика операционной системы, тем не менее информация о времени последнего выполнения пакетных заданий, таких как резервное копирование узла Cassandra², работающего на машине, тоже является хорошим кандидатом для использования сборщика текстовых файлов, потому что всегда полезно знать, производилось ли резервное копирование перед отключением машины. То есть узел Cassandra имеет тот же жизненный цикл, что и машина³.

Не используйте сборщик текстовых файлов для превращения Prometheus в систему с пассивным приемом (push) метрик. Также не используйте его, чтобы получить метрики от других экспортёров и приложений, работающих на машине, и экспортить их все через конечную точку `/metrics`, обслуживаемую экспортёром Node Exporter, – лучше использовать Prometheus для опроса каждого экспортёра и приложения по отдельности.

Использование сборщика текстовых файлов

Сборщик текстовых файлов включен по умолчанию, но для его работы нужно указать параметр командной строки `--collector.textfile.directory` при запуске Node Exporter. Этот параметр должен ссылаться на каталог, используемый исключительно для этой цели, чтобы избежать путаницы.

Чтобы опробовать этот сборщик, создайте каталог, сохраните в нем простой файл в формате списка экспортруемых метрик (как обсуждалось в разделе «Текстовый формат экспорта» главы 4) и запустите Node Exporter, указав путь к этому каталогу в параметре `--collector.textfile.directory`, как показано в примере 7.1. Имейте в виду, что сборщик текстовых файлов просматривает только файлы с расширением `.prom`.

¹ Chef – это инструмент управления конфигурацией, позволяющий автоматизировать управление инфраструктурой с помощью сценариев, называемых «cookbook».

² Распределенная база данных.

³ Если жизненный цикл метрики пакетного задания отличается от жизненного цикла машины, то, скорее всего, это пакетное задание находится на уровне служб, и для экспорта соответствующих метрик можно использовать Pushgateway, как описано в разделе «Pushgateway» главы 4.

Пример 7.1 ♦ Простой пример использования сборщика текстовых файлов

```
hostname $ mkdir myfile  
hostname $ echo example_metric 1 > myfile/example.prom  
hostname $ ./node_exporter --collector.textfile.directory=$PWD/myfile
```

В примере 7.2 показано содержимое файла, созданного в примере 7.1.

Пример 7.2 ♦ Содержимое myfile/example.prom

```
example_metric 1
```

Если обратиться к конечной точке */metrics*, обслуживаемой Node Exporter, то вы получите свою метрику:

```
# HELP example_metric Metric read from /some/path/myfile/example.prom  
# TYPE example_metric untyped  
example_metric 1
```

Если вы не укажете сами раздел HELP для метрики, то сборщик текстовых файлов сгенерирует ее автоматически. Если вы собираетесь поместить одну и ту же метрику в несколько файлов (конечно же, с разными метками), то укажите одно и то же содержимое в разделе HELP для каждой из них, потому что иначе несовпадение справочной информации вызовет ошибку.

Обычно файлы *.prom* создаются и обновляются с помощью заданий cron. Поскольку извлечение метрик может быть запущено в любое время, важно, чтобы Node Exporter не видел не до конца заполненных файлов. С этой целью выполните сначала запись метрик во временный файл в том же каталоге, а затем переименуйте его¹.

В примере 7.3 показано задание cron, создающее текстовый файл. Он записывает метрики во временный файл² и по окончании переименовывает его. Это тривиальный пример. В нем используются только короткие команды, но в реальных случаях вам потребуется создать отдельный сценарий, чтобы задание было удобочитаемым.

Пример 7.3 ♦ Задание в */etc/crontab*, экспортирующее количество строк

в */etc/shadow* как метрику *shadow_entries*

```
TEXTFILE=/path/to/myfile/directory
```

```
# Весь следующий код должен быть набран в одну строку  
*/5 * * * * root (echo -n 'shadow_entries'; grep -c . /etc/shadow)  
    > $TEXTFILE/shadow.prom.$$\n    && mv $TEXTFILE/shadow.prom.$$ $TEXTFILE/shadow.prom
```

Ряд примеров сценариев для использования со сборщиком текстовых файлов можно найти в репозитории на GitHub (<https://oreil.ly/HMkDo>).

¹ Системный вызов *rename* действует атомарно, но может использоваться только в той же файловой системе.

² Командная оболочка заменит пару символов *\$\$* текущим идентификатором процесса (*pid*).

Отметки времени

Формат экспортования поддерживает отметки времени, но их нельзя использовать со сборщиком текстовых файлов, потому что это не имеет семантического смысла, так как ваши метрики не будут отображаться с той же отметкой времени, что и другие метрики, полученные с помощью других сборщиков.

Зато вам будет доступно время `mtime`¹ файла в метрике `node_textfile_mtime_seconds`. Ее можно использовать, чтобы определить, работают ли ваши задания cron, – если значение этой метрики находится далеко в прошлом, то это может говорить о том, что задания не выполняются:

```
# HELP node_textfile_mtime_seconds Unixtime mtime of textfiles successfully read.
# TYPE node_textfile_mtime_seconds gauge
node_textfile_mtime_seconds{file="example.prom"} 1.516205651e+09
```

Теперь, запустив Node Exporter, давайте посмотрим, как можно сообщить системе Prometheus, на каких машинах он запущен.

¹ `mtime` – время последнего изменения файла.

Глава 8

Обнаружение служб

До сих пор мы использовали статические настройки Prometheus в `static_configs`. Они прекрасно подходят для простых случаев использования¹, но необходимость вручную обновлять файл `prometheus.yml` по мере добавления и удаления компьютеров может раздражать, особенно если вы находитесь в динамической среде, где новые экземпляры могут запускаться каждую минуту. В этой главе мы покажем, как можно сообщить серверу Prometheus, за какими машинами он должен наблюдать.

Вы уже знаете, где находятся все ваши машины и службы. Обнаружение служб (Service Discovery, SD) позволяет предоставлять эту информацию в Prometheus из любой базы данных, где она хранится. Prometheus из коробки поддерживает множество популярных источников информации о службах, таких как Consul, Amazon EC2 и Kubernetes. Если ваш конкретный источник еще не поддерживается, то вы можете использовать механизмы обнаружения служб на основе файлов и HTTP. Для обнаружения служб на основе файлов можно использовать систему управления конфигурациями, такую как Ansible или Chef, – просто создайте список машин и служб, известных этим службам, в правильном формате или разработайте регулярно запускаемый сценарий, который будет формировать такой список из источника данных, который вы используете. Для обнаружения служб на основе HTTP сторонние инструменты, такие как NetBox (<https://oreil.ly/-cbF3>), предлагают плагины, которые можно установить, чтобы организовать поддержку конечных точек обнаружения служб HTTP, совместимых с Prometheus. Обратите внимание, что некоторые проекты SD² поддерживают обнаружение служб и на основе HTTP, и на основе файлов.

Наличие информации о целях мониторинга, откуда следует извлекать метрики, – это только полдела. Ключевой частью Prometheus являются метки (см. главу 5), и назначение целевых меток всем целям позволит их группировать и организовывать так, чтобы сделать их мониторинг более осмысленным. Целевые метки позволяют объединять цели с одной и той же ролью, находящиеся в одной среде или управляемые одной и той же командой.

¹ У себя дома Брайана использует Prometheus со статическими настройками, поскольку имеет всего несколько машин.

² Такие как Prometheus vCloud Director SD (<https://oreil.ly/sxi0j>).

Целевые метки настраиваются в Prometheus, а не в приложениях и экспортёрах, что позволяет различным командам определять свои иерархии меток, понятные им. Для команды, управляющей инфраструктурой, может быть достаточно знать, в какой стойке и PDU¹ находится машина, в то время как для команды, обслуживающей базы данных, важно знать, является ли машина мастером PostgreSQL в промышленном окружении. Если бы у вас был разработчик ядра, исследующий редко возникающую проблему, то его могла бы интересовать информация о версии используемого ядра.

Обнаружение служб и модель извлечения метрик по опросу позволяют существовать всем этим взглядам на мир, поскольку каждая из команд может запустить свой сервер Prometheus с целевыми метками, понятными им.

Механизмы обнаружения служб

Обнаружение служб предназначено для интеграции с уже имеющимися базами данных компьютеров и служб. В дополнение к уже знакомому статическому обнаружению Prometheus 2.37.0 поддерживает Azure, Consul, DigitalOcean, Docker, Docker Swarm, DNS, Eureka, EC2², обнаружение служб на основе файлов, GCE³, Hetzner, обнаружение служб на основе HTTP, IONOS Cloud, Kubernetes, Kuma, LightSail, Linode (Akamai), Marathon, Nerve, Nomad, OpenStack, PuppetDB, Scaleway, Serverset, Uyuni, Triton и Vultr.

Обнаружение служб не просто предоставляет список машин. Это более общая задача, которую приходится решать во многих системах; приложения должны иметь возможность отыскивать свои зависимости, а технический персонал должен знать, какие машины можно безопасно вывести из эксплуатации и отправить в ремонт. Соответственно, у вас должен быть не только простой список машин и служб, но и соглашения об их организации и их жизненных циклах.

Хороший механизм обнаружения служб должен предоставлять *метаданные*, например включающие названия служб, их описание и принадлежность той или иной команде, структурированные теги или что-то еще, что может оказаться полезным. Именно метаданные вы будете преобразовывать в целевые метки, и, как правило, чем больше метаданных, тем лучше.

Полное обсуждение организации обнаружения служб выходит за рамки этой книги. Если вы еще не formalизовали базы данных для управления конфигурацией и службами, то хорошей отправной точкой вам может послужить Consul.

¹ Блок распределения питания (Power Distribution Unit, PDU), часть системы электропитания в центре обработки данных. Блоки PDU обычно питают электричеством группу стоек, и знание нагрузки процессора каждой машины может быть полезно, чтобы гарантировать, что каждый PDU сможет обеспечить требуемую мощность.

² Amazon Elastic Compute Cloud.

³ Google Compute Engine.

Нисходящие и восходящие механизмы обнаружения служб

Существует две обширные категории механизмов обнаружения служб. Механизмы, в которых экземпляры служб регистрируются самостоятельно, такие как Consul, называются восходящими. Механизмы, которые заранее знают, какие службы должны иметься, такие как EC2, называются нисходящими.

Обе категории механизмов получили широкое распространение. С помощью нисходящего механизма легко обнаружить, что из того, что должно работать, на самом деле не работает. При использовании восходящего механизма для той же цели потребуется отдельный процесс согласования, который обеспечит синхронизацию и поможет обнаружить случаи, когда экземпляр приложения останавливается до того, как успеет зарегистрироваться.

Статическое обнаружение служб

Вы уже видели статическую конфигурацию в главе 2, где цели задаются непосредственно в *prometheus.yml*. Такую конфигурацию можно использовать, если окружение простое и редко меняется. Это может быть ваша домашняя сеть, окружение с локальным Pushgateway или даже окружение с Prometheus, как в примере 8.1.

Пример 8.1 ♦ Статическое обнаружение служб для Prometheus

`scrape_configs:`

- `job_name: prometheus`
- `static_configs:`
- `targets:`
- `localhost:9090`

При использовании инструмента управления конфигурацией, такого как Ansible, можно сделать так, чтобы его система шаблонов Jinja2 создавала список всех известных машин, экспортеры которых должны опрашиваться, как в примере 8.2.

Пример 8.2 ♦ Создание списка целей с помощью шаблонов Ansible для Node Exporter на всех компьютерах

`scrape_configs:`

- `job_name: node`
- `static_configs:`
- `targets:`
- {% for host in groups["all"] %}
- {{ host }}:9100
- {% endfor %}

Помимо списка целей статическая конфигурация также может предоставить метки для них в поле `labels`. Если вам это действительно понадобится, то

файл SD, описанный в разделе «Обнаружение служб на основе файла» ниже, послужит вам хорошей отправной точкой.

Множественное число в названии раздела `static_configs` сообщает, что это список, и в нем можно указать несколько статических конфигураций, как показано в примере 8.3. Для статических конфигураций в этом нет большого смысла, но такая возможность может быть полезной при использовании других механизмов обнаружения служб, обращающихся к нескольким источникам данных. Вы даже можете смешивать и комбинировать механизмы обнаружения служб в конфигурации, хотя в результате вряд ли получите ясную и понятную конфигурацию.

Пример 8.3 ◊ Здесь определяются две цели для мониторинга, каждая в своей статической конфигурации

```
scrape_configs:
  - job_name: node
    static_configs:
      - targets: ①
        - host1:9100
        labels:
          datacenter: paris
      - targets: ②
        - host2:9100
        - host3:9100
        labels:
          datacenter: montreal
```

① Первая статическая конфигурация с единственной целью и со значением `paris` в метке `datacenter`.

② Вторая статическая конфигурация с двумя целями и со значением `montreal` в метке `datacenter`.

То же относится и к `scrape_configs`, списку конфигураций опроса, в котором можно указать столько целей, сколько понадобится. Единственное ограничение – имя задания `job_name` должно быть уникальным.

Обнаружение служб на основе файла

Обнаружение служб на основе файла (file service discovery) не использует сеть, а извлекает цели мониторинга из файлов, хранящихся в локальной файловой системе. Это позволяет интегрироваться с системами обнаружения служб, которые Prometheus не поддерживает по умолчанию или не может сделать то, что вам нужно, с доступными метаданными.

Представляемые файлы должны содержать данные в формате JSON или YAML. Расширение файла должно быть `.json`, если содержимое в формате JSON, и `.yml` или `.yaml`, если содержимое в формате YAML. Пример содержащего файла `filesd.json` в формате JSON можно увидеть в примере 8.4. В одном файле можно определить столько целей, сколько необходимо.

Пример 8.4 ❖ Файл filesd.json с тремя целями

```
[
  {
    "targets": [ "host1:9100", "host2:9100" ],
    "labels": {
      "team": "infra",
      "job": "node"
    }
  },
  {
    "targets": [ "host1:9090" ],
    "labels": {
      "team": "monitoring",
      "job": "prometheus"
    }
  }
]
```

! Формат JSON не идеален. Одна из его проблем, с которой вы наверняка столкнетесь, заключается в том, что запятая после последнего элемента в списке или хеше вызывает ошибку. Мы рекомендуем использовать библиотеку JSON для создания файлов в формате JSON, а не пытаться сделать это вручную.

Конфигурация в Prometheus использует `file_sd_configs` в конфигурации опроса `scrape_configs`, как показано в примере 8.5. Каждый элемент в списке `file_sd_configs` задает список путей к файлам, причем в именах файлов допускается использовать подстановочные знаки¹. Пути откладываются относительно рабочего каталога Prometheus, т. е. относительно каталога, из которого запущен сервер Prometheus.

Пример 8.5 ❖ prometheus.yml с конфигурацией обнаружения служб

```
scrape_configs:
  - job_name: file
    file_sd_configs:
      - files:
          - '*.json'
```

Обычно при использовании механизма обнаружения службы на основе файлов метаданные для изменения меток не предоставляются. Вместо этого указываются конечные целевые метки, которые хотелось бы иметь.

Если открыть страницу `http://localhost:9090/service-discovery` в браузере² и щелкнуть по ссылке **show more** (показать больше), то вы увидите (см.

¹ Но подстановочные знаки нельзя использовать в именах каталогов, т. е. путь `a/b/*.json` – допустимый, а `a/*/file.json` вызовет ошибку.

² Эта конечная точка была добавлена в Prometheus 2.1.0. В более старых версиях метаданные можно увидеть, наведя указатель мыши на **Labels** (Метки) на странице **Targets** (Цели).

рис. 8.1) обе метки, `job` и `team`, из `filesd.json`¹. При использовании такой конфигурации цикл опроса (извлечения метрик) завершится ошибкой, если в вашей сети не окажется хостов `host1` и `host2`.

file	show less
Discovered Labels	Target Labels
<code>_address_="host1:9090"</code> <code>_meta_filepath="filesd.json"</code> <code>_metrics_path_="/metrics"</code> <code>_scheme_="http"</code> <code>_scrape_interval_="15s"</code> <code>_scrape_timeout_="10s"</code> <code>job="prometheus"</code> <code>team="monitoring"</code>	<code>instance="host1:9090"</code> <code>job="prometheus"</code> <code>team="monitoring"</code>
<code>_address_="host1:9100"</code> <code>_meta_filepath="filesd.json"</code> <code>_metrics_path_="/metrics"</code> <code>_scheme_="http"</code> <code>_scrape_interval_="15s"</code> <code>_scrape_timeout_="10s"</code> <code>job="node"</code> <code>team="infra"</code>	<code>instance="host1:9100"</code> <code>job="node"</code> <code>team="infra"</code>
<code>_address_="host2:9100"</code> <code>_meta_filepath="filesd.json"</code> <code>_metrics_path_="/metrics"</code> <code>_scheme_="http"</code> <code>_scrape_interval_="15s"</code> <code>_scrape_timeout_="10s"</code> <code>job="node"</code> <code>team="infra"</code>	<code>instance="host2:9100"</code> <code>job="node"</code> <code>team="infra"</code>

Рис. 8.1 ◊ Страница состояния механизма обнаружения служб на основе файлов показывает три цели, обнаруженных в файле SD

Файл с целями может генерироваться механизмом шаблонов в системе управления конфигурацией, демоном, который регулярно его обновляет, или даже веб-службой, от которой принимается список целей из задания cron с помощью `wget`. Изменения вводятся в действие автоматически с помощью подсистемы `inotify`, поэтому целесообразно гарантировать атомарное

¹ `job_name` – это просто значение по умолчанию, которое мы рассмотрим в разделе «Дубликаты заданий» далее в этой главе. Другие метки `_` являются специальными, мы рассмотрим их в разделе «Как извлекать метрики» ниже в этой главе.

изменение файлов методом переименования, как было показано в разделе «Сборщик текстовых файлов» главы 7.

Обнаружение служб через HTTP

Обнаружение служб через HTTP (обычно называется *HTTP SD*) извлекает список целей с использованием HTTP. Этот механизм позволяет интегрировать любое приложение напрямую с Prometheus без создания локальных файлов на сервере Prometheus.

При обнаружении служб через HTTP сервер Prometheus будет обновлять список целей каждую минуту¹. Как показано в примере 8.6, в минимальной конфигурации достаточно указать URL конечной точки обнаружения служб.

Пример 8.6 ♦ prometheus.yml с http_sd_configs

```
scrape_configs:
- job_name: cmdb
  http_sd_configs:
    - url: http://cmdb.local/prometheus-service-discovery
```

Конечные точки обнаружения служб должны поставлять данные в том же формате JSON, что и в примере 8.4. HTTP-заголовок Content-Type при этом должен иметь значение application/json, а HTTP-ответ должен иметь код 200. В отличие от обнаружения служб на основе файлов, HTTP SD не поддерживает формат YAML.

- ✓ В случае сбоя Prometheus сохраняет последние обнаруженные цели. Поскольку этот список целей не хранится на диске, сбои, происходящие сразу после перезапуска сервера Prometheus, приведут к созданию пустого списка целей.
Отслеживать работоспособность обнаружения служб через HTTP можно с помощью счетчика `prometheus_sd_http_failures_total`. Если он постоянно увеличивается, то это означает, что Prometheus не обновляет свои цели.

HTTP SD поддерживает все необходимые параметры HTTP для аутентификации. В примере 8.7 показан вариант с сертификатами и учетными данными TLS.

Пример 8.7 ♦ prometheus.yml с http_sd_configs и параметрами безопасности

```
scrape_configs:
- job_name: cmdb
  http_sd_configs:
    - url: http://cmdb.local/prometheus-service-discovery
      authorization:
        credentials_file: token
      tls_config:
        ca_file: ca.crt
```

¹ По умолчанию запросы выполняются каждые 60 с. Этот интервал можно изменить с помощью параметра `refresh_interval`.

Обнаружение служб с помощью Consul

Consul – это механизм обнаружения служб от HashiCorp. Если вы в своей организации пока не выбрали систему обнаружения служб, то обратите внимание на Consul – это одна из самых простых и надежных систем. У Consul есть агент, который должен запускаться на каждой машине, и эти агенты постоянно взаимодействуют между собой. Сами приложения взаимодействуют только с локальным агентом на машине. Некоторые агенты также играют роль серверов, обеспечивая сохранность и согласованность информации.

Для экспериментов в среде разработки можно настроить агент Consul, следуя примеру 8.8. Чтобы использовать Consul в промышленном окружении, следуйте рекомендациям в официальном руководстве «Getting Started» (<https://oreil.ly/SmhRW>).

Пример 8.8 ❖ Настройка агента Consul в среде разработки

```
hostname $ wget https://releases.hashicorp.com/consul/1.0.2/
          consul_1.0.2_linux_amd64.zip
hostname $ unzip consul_1.0.2_linux_amd64.zip
hostname $ ./consul agent -dev
```

После этого пользовательский интерфейс Consul будет доступен по адресу <http://localhost:8500/>. В Consul есть понятие службы, и в настройках для среды разработки есть одна служба – сам Consul. Далее запустите сервер Prometheus с конфигурацией из примера 8.9.

Пример 8.9 ❖ prometheus.yml с поддержкой обнаружения служб с помощью Consul

```
scrape_configs:
  - job_name: consul
    consul_sd_configs:
      - server: 'localhost:8500'
```

Откройте в браузере страницу <http://localhost:9090/service-discovery>, и вы увидите форму, как показано на рис. 8.2, показывающую, что Consul обнаружил единственную цель с некоторыми метаданными, которая стала целью с метками `instance` и `job`. Если бы у вас было больше агентов и служб, они бы тоже появились в этой форме.

Consul не экспортирует метрик через конечную точку `/metrics`, поэтому у вас не получится извлечь их в Prometheus. Но и того, что есть, вполне достаточно, чтобы обнаружить все машины, на которых работает агент Consul и, следовательно, должен работать Node Exporter, экспортирующие метрики. Мы рассмотрим, как это сделать, в разделе «Изменение меток» далее в этой главе.



Чтобы организовать мониторинг сам механизм обнаружения служб Consul, нужно соответствующим образом настроить Prometheus и Consul. Подробности ищите в документации Consul (<https://oreil.ly/Sz3bP>). Consul Exporter (<https://oreil.ly/4ZDhM>) предоставляет метрики уровня кластера в виде пар ключ–значение.

The screenshot shows a table with two columns: 'Discovered Labels' and 'Target Labels'. The 'Discovered Labels' column lists various metadata keys such as '_address', '_meta_consul_address', '_meta_consul_dc', '_meta_consul_health', '_meta_consul_node', '_meta_consul_service', '_meta_consul_service_id', '_meta_consul_service_metadata_grpc_port', '_meta_consul_service_metadata_grpc_tls_port', '_meta_consul_service_metadata_non_voter', '_meta_consul_service_metadata_raft_version', '_meta_consul_service_metadata_read_replica', '_meta_consul_service_metadata_serf_protocol_current', '_meta_consul_service_metadata_serf_protocol_max', '_meta_consul_service_metadata_serf_protocol_min', '_meta_consul_service_metadata_version', '_meta_consul_service_port', '_meta_consul_tagged_address_lan', '_meta_consul_tagged_address_lan_ipv4', '_meta_consul_tagged_address_wan', '_meta_consul_tagged_address_wan_ipv4', '_meta_consul_tags', '_metrics_path', '_scheme', '_scrape_interval', '_scrape_timeout', and 'job'. The 'Target Labels' column contains two entries: 'instance="127.0.0.1:8300"' and 'job="consul"'. The 'job' label in the 'Target Labels' column corresponds to the 'job="consul"' label in the 'Discovered Labels' column.

Discovered Labels	Target Labels
<pre>_address="127.0.0.1:8300" meta_consul_address="127.0.0.1" meta_consul_dc="dc1" meta_consul_health="passing" meta_consul_node="nixos" meta_consul_service="consul" meta_consul_service_id="consul" meta_consul_service_metadata_grpc_port="8502" meta_consul_service_metadata_grpc_tls_port="8503" meta_consul_service_metadata_non_voter="false" meta_consul_service_metadata_raft_version="3" meta_consul_service_metadata_read_replica="false" meta_consul_service_metadata_serf_protocol_current="2" meta_consul_service_metadata_serf_protocol_max="5" meta_consul_service_metadata_serf_protocol_min="1" meta_consul_service_metadata_version="1.14.0" meta_consul_service_port="8300" meta_consul_tagged_address_lan="127.0.0.1" meta_consul_tagged_address_lan_ipv4="127.0.0.1" meta_consul_tagged_address_wan="127.0.0.1" meta_consul_tagged_address_wan_ipv4="127.0.0.1" meta_consul_tags="" metrics_path="/metrics" scheme="http" scrape_interval="15s" scrape_timeout="10s" job="consul"</pre>	<pre>instance="127.0.0.1:8300" job="consul"</pre>

Рис. 8.2 ❖ Страница состояния обнаружения служб показывает одну обнаруженную цель, ее метаданные и целевые метки, полученные из Consul

Обнаружение служб с помощью EC2

Amazon Elastic Compute Cloud, более известный как EC2, – популярный поставщик виртуальных машин. Это один из нескольких облачных провайдеров, которые поддерживаются Prometheus для обнаружения служб.

Чтобы использовать этот механизм, следует настроить в Prometheus учетные данные для доступа к EC2 API. Один из возможных способов – настроить в файле конфигурации пользователя IAM с помощью политики AmazonEC2ReadOnlyAccess¹, а также указать ключ доступа и секретный ключ, как показано в примере 8.10.

¹ Требуется только разрешение EC2:DescribeInstances, но политики, как правило, проще настроить изначально.

Пример 8.10 ❖ prometheus.yml с поддержкой обнаружения служб с помощью EC2

```
scrape_configs:
- job_name: ec2
  ec2_sd_configs:
    - region: <region>
      access_key: <access key>
      secret_key: <secret key>
```

Если вы еще не используете EC2, то запустите хотя бы один экземпляр в регионе, который настроили для просмотра в Prometheus. Если после этого в браузере перейти по адресу <http://localhost:9090/service-discovery>, то вы увидите обнаруженные цели и метаданные, извлеченные из EC2. Например, `__meta_ec2_tag_Name="My Display Name"` – это тег Name для этого экземпляра с именем, которое вы увидите в консоли EC2 (рис. 8.3).

ec2 <small>show less</small>	
Discovered Labels	Target Labels
<code>__address__="172.31.42.223:80"</code> <code>__meta_ec2_ami="ami-00ae10ea2db12689d"</code> <code>__meta_ec2_architecture="x86_64"</code> <code>__meta_ec2_availability_zone="eu-west-3c"</code> <code>__meta_ec2_availability_zone_id="euw3-az3"</code> <code>__meta_ec2_instance_id="i-071236ca033297175"</code> <code>__meta_ec2_instance_state="running"</code> <code>__meta_ec2_instance_type="t2.micro"</code> <code>__meta_ec2_owner_id="082671108254"</code> <code>__meta_ec2_primary_subnet_id="subnet-9d6503d0"</code> <code>__meta_ec2_private_dns_name="ip-172-31-42-223.eu-west-3.compute.internal"</code> <code>__meta_ec2_private_ip="172.31.42.223"</code> <code>__meta_ec2_public_dns_name="ec2-15-188-10-95.eu-west-3.compute.amazonaws.com"</code> <code>__meta_ec2_public_ip="15.188.10.95"</code> <code>__meta_ec2_subnet_id="subnet-9d6503d0"</code> <code>__meta_ec2_tag_Name="My Display Name"</code> <code>__meta_ec2_vpc_id="vpc-317a6558"</code> <code>__metrics_path__="/metrics"</code> <code>__scheme__="http"</code> <code>__scrape_interval__="15s"</code> <code>__scrape_timeout__="10s"</code> <code>job="ec2"</code>	<code>instance="172.31.42.223:80"</code> <code>job="ec2"</code>

Рис. 8.3 ❖ Страница состояния обнаружения служб показывает одну обнаруженную цель, ее метаданные и целевые метки, полученные из EC2

Обратите также внимание, что в метке `instance` указан частный IP-адрес. Это разумное значение по умолчанию, потому что предполагается, что Prometheus будет работать на другой машине, отличной от той, за которой он следит. Не все экземпляры EC2 имеют общедоступные IP-адреса, и за присваивание общедоступных IP-адресов экземплярам EC2 взимается плата.

Механизмы обнаружения служб, предлагаемые другими облачными провайдерами, во многом похожи, но требуемая конфигурация и возвращаемые метаданные различаются.

Изменение меток

Как видно из предыдущих примеров механизмов обнаружения служб, цели и метаданные могут требовать дополнительную обработку. Вы можете использовать обнаружение служб на основе файлов и передавать Prometheus именно те цели и метки, которые нужны, но в большинстве случаев они будут иметь значения, отличные от тех, что нужны. Чтобы решить эту проблему, можно настроить в Prometheus сопоставление метаданных с целями, *изменяя метки*.

-  Многие символы, такие как точки и звездочки, недопустимы в именах меток Prometheus, поэтому в метаданных, поставляемых механизмами обнаружения служб, они будут преобразованы в символы подчеркивания.

В идеальном мире механизмы обнаружения служб и обработку меток можно настроить так, чтобы новые машины и приложения подхватывались и отслеживались автоматически. В реальном мире развивающееся окружение может стать настолько сложным, что вам придется регулярно обновлять файл конфигурации Prometheus, но к тому времени вы, скорее всего, будете иметь настолько сложную инфраструктуру, что это станет существенным препятствием.

Выбор целей для мониторинга

Первое, что нужно настроить, – это цели для мониторинга. Если вы работаете в команде, управляющей одной службой, то для вас едва ли было бы желательным, чтобы ваш сервер Prometheus опрашивал все цели в одном и том же регионе EC2.

Продолжим пример 8.5 и представим, что вам нужно просто наблюдать за машинами, которые поддерживает команда управления инфраструктурой. С этой целью вы можете использовать *действие изменения меток keep*, как показано в примере 8.11. К значениям меток, перечисленным (через точку с запятой) в `source_labels`, применяется регулярное выражение `regeх`, и если оно обнаруживает совпадение, то цель сохраняется. Поскольку в этом примере определено только одно действие, то сохранены будут все цели с `team="infra"`.

Для целей с меткой `team="monitoring"` регулярное выражение не будет находить совпадения, поэтому эти цели будут удалены.

-  Регулярные выражения, используемые механизмом изменения меток, *привязывают* к началу и концу строки, т. е. выражение `infra` не будет совпадать с `fooinfra` или `infrafoo`.

Пример 8.11 ❖ Использование действия `keep` для мониторинга только целей с меткой `team="infra"`

```
scrape_configs:
- job_name: file
  file_sd_configs:
    - files:
      - '*.json'
  relabel_configs:
    - source_labels: [team]
      regex: infra
      action: keep
```

В одном разделе `relabel_configs` можно определить несколько действий по изменению метки – все они будут выполнены по порядку, если только очередное действие `keep` или `drop` не приведет к удалению цели. Например, в примере 8.12 будут удалены все цели, так как метка не может иметь значения `infra` или `monitoring`.

Пример 8.12 ❖ Два действия по изменению метки, требующие противоречивых значений для метки `team`

```
scrape_configs:
- job_name: file
  file_sd_configs:
    - files:
      - '*.json'
  relabel_configs:
    - source_labels: [team]
      regex: infra
      action: keep
    - source_labels: [team]
      regex: monitoring
      action: keep
```

Чтобы разрешить метки, которые могут иметь одно из нескольких значений, следует использовать оператор чередования `|` (символ вертикальной черты), что можно считать необычным способом сказать «одно или другое». В примере 8.13 показано, как правильно оставить только цели для команд управления инфраструктурой и мониторингом.

Пример 8.13 ❖ Использование оператора `|`, чтобы разрешить любое из нескольких значений метки

```
scrape_configs:
- job_name: file
  file_sd_configs:
    - files:
      - '*.json'
  relabel_configs:
    - source_labels: [team]
      regex: infra|monitoring
      action: keep
```

В дополнение к действию `keep`, которое отбрасывает цели, не соответствующие регулярному выражению, можно использовать действие `drop`, которое, напротив, отбрасывает цели, соответствующие регулярному выражению. В `source_labels` можно указать несколько меток, перечислив их через точку с запятой¹. Чтобы исключить из опроса задания группы поддержки мониторинга и Prometheus, их можно удалить, как показано в примере 8.14.

Пример 8.14 ❖ Использование нескольких меток в `source_labels`

```
scrape_configs:
- job_name: file
  file_sd_configs:
    - files:
      - '*.json'
  relabel_configs:
    - source_labels: [job, team]
      regex: prometheus;monitoring
      action: drop
```

Как использовать возможность изменения меток, зависит только от вас. Но при этом желательно определить некоторые соглашения. Например, экземпляры EC2 должны иметь тег `team` с названием команды, которой они принадлежат, или все службы в промышленном окружении должны иметь тег `production` в Consul. Без соглашений каждая новая служба потребует специальной обработки для мониторинга, что, вероятно, не лучшая трата вашего времени.

Если ваш механизм обнаружения служб включает проверку работоспособности, то не используйте ее результаты для удаления неработоспособных экземпляров. Даже когда экземпляр отмечается как неработоспособный, он все еще может создавать полезные метрики, особенно касающиеся запуска и завершения работы.

i В Prometheus должна иметься цель для каждого отдельного экземпляра вашего приложения. Вариант опроса через балансировщики нагрузки не годится, потому что нет никакой гарантии, что каждый раз опрашиваться будет один и тот же экземпляр, вследствие чего, например, некоторые счетчики могут выглядеть так, будто они считают в обратную сторону.

Регулярные выражения

Prometheus использует механизм регулярных выражений *RE2*, заимствованный из Go. Он обеспечивает линейное время оценки выражений, но не поддерживает обратных ссылок, опережающих проверок и некоторых других продвинутых возможностей.

Для тех, кто не знаком с регулярными выражениями, отметим, что они определяют правила (называемые *шаблонами*), которые затем сопоставляются с текстом. Ниже приводится краткий справочник по регулярным выражениям.

¹ Символ-разделитель можно переопределить с помощью параметра `separator`.

	Соответствует
a	Символу а
.	Любому одному символу
\.	Одному символу точки
.*	Любому количеству любых символов
.+	По меньшей мере одному символу
a+	Одному или нескольким символам а
[0-9]	Любой одной цифре от 0 до 9
\d	Любой одной цифре от 0 до 9
\d*	Любому количеству цифр
[^0-9]	Одному символу, не являющемуся цифрой
ab	Символу а, за которым следует символ б
a(b/c*)	Символу а, за которым следует один символ б или любое количество символов с

Круглые скобки создают сохраняющую группу. То есть при применении шаблона `(.)\(\d+)` к тексту `a123` будут сохранены две группы: с символом `a` и с цифрами `123`. Сохраняющие группы используются для извлечения фрагментов строки с целью их использования в дальнейшем.

Метки целей

Метки целей (или целевые метки) – это метки, которые добавляются к меткам каждого временного ряда, возвращаемого процедурой опроса. Они идентифицируют цели¹ и, соответственно, не должны меняться с течением временем, как это возможно в случае с номерами версий или владельцами компьютеров.

Каждый раз, когда целевые метки меняют значение, идентификация временных рядов тоже меняется. Это изменение вызовет разрывы на графиках и, возможно, проблемы с правилами и уведомлениями.

Для чего служат целевые метки? Вы уже видели метки `job` и `instance` в примерах выше. Целевые метки есть у всех целей. Также принято добавлять целевые метки для более широкого применения, например определяющие принадлежность к окружению разработки или промышленной эксплуатации, региону, центру обработки данных и команде, управляющей целью. Метки, отражающие структуру приложения, тоже могут иметь смысл, например если используется сегментирование.

Целевые метки часто используются для выбора, группировки и объединения целей в PromQL. Например, с помощью целевых меток можно обрабатывать оповещения для окружения разработки иначе, чем для промышленного

¹ В принципе, допустимо использовать для двух разных целей одинаковые метки, но при этом другие конфигурационные параметры должны быть разными. Однако в общем случае этого следует избегать, поскольку такие метрики, как `up`, будут конфликтовать.

окружения, чтобы сообщить, какой сегмент приложения работает с наибольшей нагрузкой или какая команда использует больше всего процессорного времени.

Но целевые метки имеют свою цену. Добавление еще одной метки с точки зрения ресурсов обходится довольно дешево, но ситуация меняется, когда вы пишете PromQL. Каждая дополнительная метка – это еще один нюанс, о котором нужно помнить при написании каждого выражения PromQL. Например, если добавить метку `host`, уникальную для каждой цели, то это нарушит ожидание уникальности метки `instance`, что в свою очередь может нарушить работу выражений агрегирования, использующих `without(instance)`. Более полное обсуждение этого вопроса вы найдете в главе 14.

Как правило, целевые метки должны иметь иерархическую организацию, и каждая должна добавлять дополнительную отличительную черту. Например, можно определить иерархию, в которой регионы содержат центры обработки данных, центры содержат окружения, окружения – службы, службы – задания, задания – экземпляры. Это не жесткое правило, но вы можете проявить предусмотрительность и создать метку центра обработки данных, даже если в настоящее время у вас имеется только один центр обработки данных¹.

Для передачи информации, известной приложению, но не подходящей на роль целевой метки, такой как номер версии, можно использовать информационные метрики, как описано в разделе «Информационные метрики» главы 5.

Если у вас появится желание добавить для каждой цели в Prometheus некоторые общие метки, такие как регион, то для этого можно использовать внешние метки `external_labels`, как обсуждается в разделе «Внешние метки» главы 18.

replace

Итак, как лучше добавлять целевые метки? Ответ – с помощью действия `replace`, которое позволяет копировать метки и применять регулярные выражения.

Продолжим пример 8.5 и предположим, что команда поддержки мониторинга с названием `monitoring` получила другое название – `monitor`, но у вас пока нет возможности изменить файл с конфигурацией механизма обнаружения служб. В таком случае можно использовать функцию изменения меток. В примере 8.15 настроен поиск метки `team`, значение которой соответ-

¹ С другой стороны, не стоит заглядывать вперед слишком далеко. Это нормально, если с развитием архитектуры будет меняться и иерархия целевых меток. Точно предсказать, как она изменится в будущем, практически невозможно. Представьте, например, замену традиционного центра обработки данных услугами облачного провайдера, такого как EC2, имеющего зоны доступности.

ствует регулярному выражению `monitoring` (т. е. точно совпадает со строкой `monitoring`), и изменение ее значения на `monitor`.

Пример 8.15 ♦ Использование действия `replace_relabel` для замены `team="monitoring"` на `team="monitor"`

```
scrape_configs:
  - job_name: file
    file_sd_configs:
      - files:
          - '*.json'
    relabel_configs:
      - source_labels: [team]
        regex: monitoring
        replacement: monitor
        target_label: team
        action: replace
```

Это довольно простой пример, но на практике иногда приходится указывать значения меток для замены одно за другим, что может быть утомительно. Допустим, оказалось, что проблема была в суффиксе `ing` в слове `monitoring`, и вы хотели бы убрать суффиксы `ing` во всех значениях метки `team`. В примере 8.16 показано, как это сделать с помощью регулярного выражения `(.*)ing`, которое совпадает со всеми строками, оканчивающимися на `ing`, и оставляет начало значения метки, захваченное первой сохраняющей группой. Замещающее значение включает первую сохраняющую группу, содержимое которой и будет помещено в метку `team`.

Пример 8.16 ♦ Использование действия `replace_relabel` для удаления суффикса «`ing`» из метки `team`

```
scrape_configs:
  - job_name: file
    file_sd_configs:
      - files:
          - '*.json'
    relabel_configs:
      - source_labels: [team]
        regex: '(.*)ing'
        replacement: '${1}'
        target_label: team
        action: replace
```

Если одна из целевых меток имеет какое-то отличающееся значение, например `team="infra"`, то действие `replace` не повлияет на нее, как можно видеть на рис. 8.4.

Метка с пустым значением равнозначна отсутствию метки, поэтому при желании можно удалить метку `team`, как показано в примере 8.17.

file	show less
Discovered Labels	Target Labels
<code>__address__="host1:9100"</code> <code>__meta_filepath="filesd.json"</code> <code>__metrics_path__="/metrics"</code> <code>__scheme__="http"</code> <code>__scrape_interval__="15s"</code> <code>__scrape_timeout__="10s"</code> <code>job="node"</code> <code>team="infra"</code>	<code>instance="host1:9100"</code> <code>job="node"</code> <code>team="infra"</code>
<code>__address__="host2:9100"</code> <code>__meta_filepath="filesd.json"</code> <code>__metrics_path__="/metrics"</code> <code>__scheme__="http"</code> <code>__scrape_interval__="15s"</code> <code>__scrape_timeout__="10s"</code> <code>job="node"</code> <code>team="infra"</code>	<code>instance="host2:9100"</code> <code>job="node"</code> <code>team="infra"</code>

Рис. 8.4 ❖ Суффикс «ing» удаляется из значения monitoring целевой метки team, а значение «infra» остается неизменным

Пример 8.17 ❖ Использование действия replace для удаления метки team

```
scrape_configs:
- job_name: file
  file_sd_configs:
    - files:
      - '*.json'
  relabel_configs:
    - source_labels: []
      regex: '(.*)'
      replacement: '${1}'
      target_label: team
      action: replace
```



Все метки, значения которых начинаются с двух символов подчеркивания __, автоматически отбрасываются в конце процедуры изменения целевых меток, поэтому нет необходимости делать это вручную.

Поскольку применение регулярного выражения к строке, ее захват и использование в качестве строки замены является обычным явлением, все эти настройки действуют по умолчанию. То есть их можно опустить¹. В результате пример 8.18 будет оказывать тот же эффект, что и пример 8.17.

Пример 8.18 ◊ Использование настроек по умолчанию для удаления метки team

```
scrape_configs:
- job_name: file
  file_sd_configs:
    - files:
      - '*.json'
  relabel_configs:
    - source_labels: []
      target_label: team
```

Теперь, получив представление о работе действия replace, рассмотрим более реалистичный пример. В примере 8.9 создается цель с портом 80, и было бы полезно изменить его на порт 9100, который использует Node Exporter. В примере 8.19 мы берем адрес из Consul, добавляем в конец :9100 и помещаем результат в метку __address__.

Пример 8.19 ◊ Использование IP-адреса, полученного из Consul, и подстановка порта 9100 для доступа к Node Exporter

```
scrape_configs:
- job_name: node
  consul_sd_configs:
    - server: 'localhost:8500'
  relabel_configs:
    - source_labels: [__meta_consul_address]
      regex: '(.*)'
      replacement: '${1}:9100'
      target_label: __address__
```

Если процедура изменения меток создаст две идентичные цели в одной из конфигураций опроса, дубликаты будут автоматически удалены. Поэтому если у вас на каждой машине запущено много служб Consul, то при использовании настроек из примера 8.19 вы получите только одну цель для каждой машины.

job, instance и __address__

В предыдущих примерах можно заметить, что целевая метка instance присутствует, но в метаданных нет соответствующей метки instance. Так откуда

¹ Также можете опустить source_labels: []. Мы оставили этот параметр, чтобы было понятно, что метка удаляется.

она взялась? Ответ заключается в том, что, если цель не имеет метки `instance`, она создается автоматически и получает значение метки `_address`.

`instance` и `job` – это две метки, которые создаются для целей всегда. Метка `job` по умолчанию получает значение конфигурационного параметра `job_name`. Метка `job` идентифицирует набор экземпляров, которые служат одной цели и, как правило, все выполняют один и тот же двоичный файл с одной и той же конфигурацией¹. Метка `instance` идентифицирует один экземпляр в задании.

`_address` – это хост и порт, к которым Prometheus будет подключаться при выполнении процедуры опроса. Она определяет значение по умолчанию для метки `instance`, но вообще является отдельной меткой, поэтому ей можно задать другое значение. Например, в метке `instance` можно использовать имя узла `Consul`, а в `_address` оставить IP-адрес, как показано в примере 8.20. Это лучшее решение, чем добавление дополнительной метки `host`, `node` или `alias` с более красивым именем, потому что позволяет избежать добавления второй метки, уникальной для каждой цели, что может вызвать сложности в PromQL.

Пример 8.20 ❖ Использование IP-адреса Consul с портом 9100 в качестве адреса с именем узла в метке `instance`

```
scrape_configs:
- job_name: consul
  consul_sd_configs:
    - server: 'localhost:8500'
  relabel_configs:
    - source_labels: [__meta_consul_address]
      regex: '(.*)'
      replacement: '${1}:9100'
      target_label: _address_
    - source_labels: [__meta_consul_node]
      regex: '(.*)'
      replacement: '${1}:9100'
      target_label: instance
```

 Prometheus выполняет разрешение имен DNS для `_address`, поэтому один из способов получить более удобочитаемые значения метки `instance` – указать в `_address` значение в формате `host:port` вместо `ip:port`.

labelmap

Действие `labelmap` отличается от действий `drop`, `keep` и `replace`, которые вы уже видели, – оно применяется к именам меток, а не к значениям меток.

Оно может пригодиться, если используемый вами механизм обнаружения служб уже имеет форму меток ключ–значение, и вы хотели бы использовать некоторые из них в качестве целевых меток, например чтобы разрешить настройку произвольных целевых меток без необходимости изменять конфигурацию Prometheus каждый раз, когда появляется новая метка.

¹ Задание потенциально можно разделить на сегменты с помощью другой метки.

Теги EC2, например, имеют вид пар ключ–значение. Вы можете придерживаться соглашения, помещая имя службы в тег `service`, и согласовывать его семантику с семантикой метки `job` в Prometheus. Вы также можете принять соглашение по преобразованию любых тегов с префиксом `monitor_` в целевые метки. Например, преобразовывать тег EC2 `monitor_foo=bar` в целевую метку Prometheus `foo="bar"`. В примере 8.21 показано, как это реализовать с помощью действия `relabel` в отношении метки `job` и действия `labelmap` в отношении префикса `monitor_`.

Пример 8.21 ✦ Преобразование тега EC2 `service` в метку `job` и всех тегов с префиксом `monitor_` в дополнительные целевые метки

```
scrape_configs:
- job_name: ec2
  ec2_sd_configs:
    - region: <region>
      access_key: <access key>
      secret_key: <secret key>
  relabel_configs:
    - source_labels: [__meta_ec2_tag_service]
      target_label: job
    - regex: __meta_ec2_public_tag_monitor_(.*)
      replacement: '${1}'
      action: labelmap
```

Но опасайтесь слепого копирования всех меток в подобном сценарии, так как маловероятно, что Prometheus является единственным потребителем таких метаданных в вашей общей архитектуре. Например, во все ваши экземпляры EC2 может быть добавлен новый тег с целью учета потребляемых ресурсов для выставления счетов. Если `labelmap` автоматически превратит этот тег в целевую метку, то это изменит все ваши целевые метки и, вероятно, нарушит работу механизмов формирования графиков и отправки уведомлений. В связи с этим разумно использовать общезвестные имена (например, тег `service`) или имена в четко ограниченном пространстве имен (например, `monitor_`).

Изменение регистра значений меток

Иногда бывает полезно изменить регистр значений меток. Это позволит согласовать значения меток при использовании нескольких механизмов обнаружения служб. Изменить регистр значения метки можно с помощью действий `lowercase` и `uppercase`, как показано в примере 8.22.

Пример 8.22 ✦ `prometheus.yml` с действием `lowercase` в настройках процедуры изменения меток

```
- job_name: ionos
  ionos_sd_configs:
    - basic_auth:
        username: john.doe@example.com
```

```
password: <secret>
datacenter_id: 57375146-e890-4b84-8d59-c045d3eb6f4c
relabel_configs:
- source_labels: [__meta_ionos_server_type]
  target_label: server_type
  action: lowercase
```

Списки

Не все механизмы обнаружения служб имеют метки или теги в виде пар ключ–значение. Некоторые имеют просто список тегов, причем каноническим примером являются теги Consul. Хотя Consul – это наиболее вероятное место, где вы столкнетесь с такими списками, есть и другие места, где механизм обнаружения служб должен каким-то образом преобразовать список в метаданные вида ключ–значение, такие как идентификатор подсети EC2¹.

Это делается путем объединения элементов в список (через запятую) и использования этого списка в качестве значения метки. Запятые также добавляются в начало и в конец значения, чтобы упростить написание регулярных выражений.

В качестве примера предположим, что служба Consul имеет теги `dublin` и `prod`. Метка `__meta_consul_tags` может иметь значение `,dublin,prod,` или `,prod,dublin,,`, поскольку теги не упорядочены. Чтобы организовать опрос только целей в промышленном окружении, следует использовать действие `keep`, как показано в примере 8.23.

Пример 8.23 ♦ Опрашивать только службы Consul с тегом prod

```
scrape_configs:
- job_name: node
  consul_sd_configs:
    - server: 'localhost:8500'
  relabel_configs:
    - source_labels: [__meta_consul_tags]
      regex: '.*,prod,.*'
      action: keep
```

Иногда у вас могут иметься теги, являющиеся парами ключ–значение. Такие значения можно преобразовать в метки, но для этого нужно знать возможные значения. В примере 8.24 показано, как тег, обозначающий среду, в которой находится цель, можно преобразовать в метку `env`.

Пример 8.24 ♦ Использование тегов `prod`, `staging` и `dev` для создания метки `env`

```
scrape_configs:
- job_name: node
  consul_sd_configs:
    - server: 'localhost:8500'
```

¹ Экземпляр EC2 может иметь несколько сетевых интерфейсов, связанных с разными подсетями.

```
relabel_configs:
- source_labels: [__meta_consul_tags]
  regex: '.*,(prod|staging|dev),.*'
  target_label: env
```



При использовании сложных правил изменения меток может понадобиться временная метка для ввода значения. Для этой цели зарезервирован префикс `__tmp`.

Как извлекать метрики

Теперь у нас есть цели с целевыми метками целей и меткой `__address__` для подключения к ним. Однако есть еще несколько дополнительных параметров, которые можно настроить, например путь, отличный от `/metrics`, или параметры аутентификации клиента.

В примере 8.25 показаны некоторые наиболее распространенные варианты, которые вы можете использовать. Они имеют свойство меняться со временем, поэтому обязательно сверьтесь с документацией, чтобы уточнить имеющиеся настройки.

Пример 8.25 ❖ Конфигурация опроса с несколькими доступными вариантами

```
scrape_configs:
- job_name: example
  consul_sd_configs:
    - server: 'localhost:8500'
  scrape_timeout: 5s
  metrics_path: /admin/metrics
  params:
    foo: [bar]
  scheme: https
  tls_config:
    insecure_skip_verify: true
  basic_auth:
    username: brian
    password: hunter2
```

`metrics_path` – это только путь для URL, и если попытаться указать значение, например `/metrics?foo=bar`, оно будет преобразовано в значение `/metrics%3Ffoo=bar`. Вместо этого любые параметры URL следует перечислять в параметре `params`, хотя обычно это требуется только для федерации и классов экспортёров, включая SNMP Exporter и Blackbox Exporter. Добавление произвольных заголовков не поддерживается, так как это затрудняет отладку. Если вам нужна гибкость сверх того, что предлагается, то для настройки запросов на получение метрик вы всегда можете использовать прокси-сервер, определив параметр `proxy_url`.

`scheme` может иметь значение `http` или `https`. При использовании схемы `https` есть возможность настроить дополнительные параметры, включая `key_file` и `cert_file`, чтобы использовать TLS-аутентификацию клиента. `in-`

`secure_skip_verify` позволяет отключить проверку сертификата TLS цели, что не рекомендуется с точки зрения безопасности.

Помимо TLS-аутентификации клиента, можно настроить базовую аутентификацию HTTP Basic Authentication и аутентификацию с токеном HTTP Bearer Token Authentication с помощью `basic_auth`, `OAuth2` и `authorization`. Токен также можно прочитать из файла, а не из конфигурации с помощью `credentials_file` в `authorization`. Поскольку предполагается, что токены и пароли в базовой аутентификации должны сохраняться в тайне, они будут замаскированы на страницах состояния Prometheus, чтобы исключить случайную их утечку.

В дополнение к `scrape_timeout` в конфигурации опроса также можно переопределить `scrape_interval`, но в целом для простоты желательно стремиться использовать в Prometheus один интервал опроса.

Параметры `scheme`, `metrics_path` и `params` доступны для изменения в процедуре изменения меток в виде меток `__scheme__`, `__metrics_path__` и `__param_<name>`. Если имеется несколько параметров URL с одинаковыми именами, то для изменения будет доступен только первый. Другие параметры изменить нельзя по разным причинам, от здравомыслия до безопасности.

Метаданные механизма обнаружения служб считаются неважными с точки зрения безопасности¹ и обычно доступны всем, имеющим доступ к пользовательскому интерфейсу Prometheus. Поскольку секретные сведения могут указываться только в конфигурации опроса, рекомендуется стандартизировать учетные данные, используемые для доступа к вашим службам.

Дубликаты заданий

Значение метки `job_name` должно быть уникальным, но это всего лишь значение по умолчанию, и нет никаких правил, не запрещающих иметь разные конфигурации опроса, создающие цели с одной и той же меткой `job`.

Например, если есть задания, для которых требуется другой секрет, определяемый тегом Consul, то их можно отделить, используя действия `keep` и `drop`, а затем использовать процедуру изменения меток, чтобы установить метку `job`:

```
- job_name: my_job
  consul_sd_configs:
    - server: 'localhost:8500'
  relabel:
    - source_labels: [__meta_consul_tag]
      regex: '.*,specialsecret,.*'
      action: drop
  basic_auth:
    username: brian
    password: normalSecret
- job_name: my_job_special_secret
  consul_sd_configs:
    - server: 'localhost:8500'
```

¹ Системы обнаружения служб обычно не предназначены для хранения секретов.

```

relabel
- source_labels: [__meta_consul_tag]
  regex: '.*,specialsecret,.*'
  action: keep
- replacement: my_job
  target_label: job
basic_auth:
  username: brian
  password: specialSecret

```

metric_relabel_configs

Изменение меток можно использовать не только для преобразования метаданных механизма обнаружения служб в целевые метки, как было задумано первоначально, но также и для решения других задач. Одна из них – *изменение метрик*: такое изменение применяется к временным рядам, извлеченным из цели.

Действия `keep`, `drop`, `replace`, `lowercase`, `uppercase` и `labelmap`, представленные выше, можно также использовать в `metric_relabel_configs`, потому что механизм изменения меток не имеет никаких ограничений на область его применения¹.

 На всякий случай напомним, что конфигурация `relabel_configs` применяется для выяснения опрашиваемых целей, а `metrics_relabel_configs` – после опроса.

Есть два случая, когда может понадобиться изменить метрики: при удалении дорогостоящих метрик и при исправлении плохих метрик. Такие проблемы лучше решать на стороне источника, тем не менее всегда полезно знать, что в запасе есть также тактические средства исправления.

Изменение метрик дает доступ к временным рядам после их извлечения, но до того, как они будут сохранены в хранилище. Действия `keep` и `drop` можно применить к метке `__name__` (обсуждается во врезке «Зарезервированные метки и `_name_`» в главе 5), чтобы выбрать временной ряд, который действительно нужно сохранить. Если, например, обнаружится, что метрика `http_request_size_bytes`² имеет чрезмерную кардинальность и может вызвать проблемы с производительностью, то ее можно удалить, как показано в примере 8.26. Она все еще будет передаваться по сети и анализироваться, но такое решение даст вам некоторую передышку.

Пример 8.26 ♦ Удаление дорогостоящей метрики с помощью `metric_relabel_configs`

`scrape_configs:`

`- job_name: prometheus`

¹ Это не означает, что все действия по изменению меток имеют смысл во всех контекстах.

² В Prometheus 2.3.0 эту метрику заменили гистограммой с именем `prometheus_http_response_size_bytes`.

```

static_configs:
- targets:
  - localhost:9090
metric_relabel_configs:
- source_labels: [__name__]
  regex: http_request_size_bytes
  action: drop

```

Метки `le` также остаются доступными. Как упоминалось во врезке «Кумулятивные гистограммы» в главе 3, есть возможность отбрасывать определенные сегменты (но не `+Inf`) гистограмм, при этом вы по-прежнему сможете вычислять квантили, как показано в примере 8.27 с гистограммой `prometheus_tsdb_compaction_duration_seconds`.

Пример 8.27 ❖ Удаление сегментов гистограммы для уменьшения кардинальности

```

scrape_configs:
- job_name: prometheus
  static_configs:
    - targets:
      - localhost:9090
  metric_relabel_configs:
    - source_labels: [__name__, le]
      regex: 'prometheus_tsdb_compaction_duration_seconds_bucket;(4|32|256)'
      action: drop

```

 Конфигурация `metric_relabel_configs` применяется только к метрикам, извлекаемым из цели. Она не применяется к таким метрикам, как `up`, которые касаются самого опроса и имеют только целевые метки.

Также `metric_relabel_configs` можно использовать для переименования метрик и меток и даже для извлечения меток из имен метрик.

labeldrop и *labelkeep*

Механизм изменения меток поддерживает два дополнительных действия, которые вряд ли когда-либо потребуются для изменения целевых меток, но могут принести пользу при изменении метрик. Иногда экспортеры могут проявлять чрезмерный энтузиазм в отношении применяемых ими меток или путать инструментированные метки с целевыми и возвращать инструментированные метки под видом целевых. Действие `labelace` может применяться только к именам известных заранее меток, что бывает не всегда.

В таких случаях на помощь приходят действия `labeldrop` и `labelkeep`. Подобно `labelmap`, они применяются к именам, а не к значениям меток. Вместо копирования `labeldrop` и `labelkeep` удаляют метки. В примере 8.28 действие `labeldrop` используется для удаления всех меток с заданным префиксом.

Пример 8.28 ❖ Удаление всех полученных меток, начинающихся с `node_`

```

scrape_configs:
- job_name: misbehaving

```

```

static_configs:
- targets:
  - localhost:1234
metric_relabel_configs:
- regex: 'node_.*'
  action: labeldrop

```

Когда возникает необходимость в этих действиях, отдавайте предпочтение `labeldrop`, где это целесообразно. При использовании `labelkeep` вам придется указать каждую метку, которую нужно сохранить, включая `__name__`, `le` и `quantile`.

Конфликты меток и `honor_labels`

Действие `labeldrop` можно использовать, когда экспортер ошибочно полагает, что знает, какие метки вам нужны, однако существует несколько экспортёров, которые действительно знают, что вам нужно. Например, метрики в Pushgateway не должны иметь метки `instance`, как упоминалось в разделе «Pushgateway» главы 4. Для таких случаев нужен некоторый способ, предотвращающий применение целевой метки `instance`, полученной с помощью Pushgateway.

Но сначала давайте посмотрим, что происходит в случаях, когда имеется целевая метка с тем же именем, что и инструментированная метка. При наличии инструментированных меток с теми же именами, что и целевые метки, последние получают больший приоритет. Например, если возникнет конфликт с меткой `job`, то одноименная инструментированная метка будет переименована в `exrogated_job`.

Если требуется, чтобы приоритет отдавался инструментированной метке и она замещала целевую, то можно установить параметр `honor_labels: true` в конфигурации опроса. Это единственное место в Prometheus, где пустая метка – это не то же самое, что отсутствующая метка. Если извлеченная метрика имеет явно заданную метку `instance=""` и в конфигурации опроса присутствует параметр `honor_labels: true`, то результирующий временной ряд не будет иметь метки `instance`. Этот метод используется в Pushgateway.

Помимо Pushgateway, параметр `honor_labels` также может появляться в настройках извлечения метрик из других систем мониторинга, если вы не последовали рекомендации в главе 11, советующей запускать не более одного экспортёра на каждый экземпляр приложения.



Если нужен более детальный контроль над конфликтующими целевыми и инструментированными метками, то можно использовать `metric_relabel_configs` для настройки меток перед добавлением метрик в хранилище. Обработка конфликтов меток с учётом параметра `honor_labels` выполняется перед `metric_relabel_configs`.

Теперь, познакомившись с обнаружением служб, рассмотрим мониторинг контейнеров и возможности обнаружения служб вместе с Kubernetes.

Глава 9

Контейнеры и Kubernetes

Контейнерные технологии приобретают все большую популярность, благодаря таким технологиям, как Docker и Kubernetes. Вполне возможно, что вы тоже уже используете их. В этой главе мы рассмотрим экспортеры, которые можно использовать с контейнерами, и объясним, как использовать Prometheus с Kubernetes.

Все компоненты Prometheus успешно работают в контейнерах. Единственное исключение – Node Exporter, как отмечалось в главе 7.

cAdvisor

cAdvisor во многом похож на Node Exporter. Последний предоставляет метрики о машине, а первый – экспортирует метрики о *cgroups*. Cgroups – это механизм изоляции ядра Linux, который обычно используется для реализации контейнеров в Linux, а также такими средами выполнения, как systemd.

Вы можете запустить cAdvisor с помощью Docker:

```
docker run \
--volume=/:/rootfs:ro \
--volume=/var/run:/var/run:rw \
--volume=/sys:/sys:ro \
--volume=/var/lib/docker:/var/lib/docker:ro \
--volume=/dev/disk:/dev/disk:ro \
--publish=8080:8080 \
--detach=true \
--name=cadvisor \
gcr.io/cadvisor/cadvisor:v0.45.0
```

Если теперь в браузере открыть адрес <http://localhost:8080/metrics>, то вы увидите длинный список метрик, как показано на рис. 9.1.

```
# HELP cadvisor_version_info A metric with a constant '1' value labeled by kernel version, OS
version, docker_version, cAdvisor version & cAdvisor revision.
# TYPE cadvisor_version_info gauge
cadvisor_version_info{cadvisorRevision="86b11c65",cadvisorVersion="v0.45.0",dockerVersion="",kernel
Version="5.18.19",osVersion="Alpine Linux v3.16"} 1
# HELP container_blkio_device_usage_total Blkio Device bytes usage
# TYPE container_blkio_device_usage_total counter
container_blkio_device_usage_total{device="/dev/disk/by-uuid/6451bb4f-81d7-4739-9638-
ca9b79ef6f38",id="/",image="",major="254",minor="3",name="",operation="Read"} 1.37632256e+09
1664884883416
container_blkio_device_usage_total{device="/dev/disk/by-uuid/6451bb4f-81d7-4739-9638-
ca9b79ef6f38",id="/",image="",major="254",minor="3",name="",operation="Write"} 1.62582528e+09
1664884883416
container_blkio_device_usage_total{device="/dev/dm-
0",id="/",image="",major="254",minor="0",name="",operation="Read"} 2.548012032e+09 1664884883416
container_blkio_device_usage_total{device="/dev/dm-
0",id="/",image="",major="254",minor="0",name="",operation="Write"} 6.952341504e+09 1664884883416
container_blkio_device_usage_total{device="/dev/dm-
0",id="/init.scope",image="",major="254",minor="0",name="",operation="Read"} 1.196032e+06
1664884884114
```

Рис. 9.1 ♦ Начало страницы со списком метрик, экспортруемых cAdvisor

Метрики контейнеров имеют префикс `container_`, и, как нетрудно заметить, все они имеют метку `id`. Метки `id`, начинающиеся с `/docker/` или `/system.slice/docker/`, взяты из Docker и его контейнеров, а метки `id` со значениями `/user.slice/` и `/system.slice/`, берутся из systemd. Если у вас есть другое программное обеспечение, использующее контрольные группы (cgroups), то его контрольные группы тоже будут присутствовать в списке.

Эти метрики можно получить с помощью `prometheus.yml`, например:

```
scrape_configs:
- job_name: cAdvisor
  static_configs:
    - targets:
      - localhost:8080
```

Метрики потребления процессора

Для контейнеров экспортируются три метрики, характеризующие потребление процессора: `container_cpu_usage_seconds_total`, `container_cpu_system_seconds_total` и `container_cpu_user_seconds_total`.

`container_cpu_usage_seconds_total` делится по процессорам, а не по режимам. `container_cpu_system_seconds_total` и `container_cpu_user_seconds_total` – это режимы `user` и `system` соответственно по аналогии с Node Exporter, как описано в разделе «Сборщик информации о процессоре» главы 7. Это все счетчики, к которым можно применять функцию `rate`.



При наличии множества контейнеров и процессоров на одном компьютере совокупная кардинальность метрик из cAdvisor может превратиться в проблему производительности. Чтобы отбросить наименее ценные метрики на этапе извлечения, можно использовать действие `drop` изменения меток, как описано в разделе «`metric_label_configs`» главы 8.

Метрики потребления памяти

Метрики памяти, экспортируемые cAdvisor, тоже могут вносить путаницу, как и аналогичные метрики Node Exporter, и для их понимания требуется копаться в коде и документации.

`container_memory_cache` – это объем кеша страниц, используемый контейнером, в байтах. `container_memory_rss` – размер резидентного набора (Resident Set Size, RSS) в байтах. Это не размер резидентной, или физической, памяти используемой процессом, потому что в этой метрике не учитываются размеры отображаемых файлов¹. `container_memory_usage_bytes` – это RSS и кеш страниц, и он ограничен значением `container_spec_memory_limit_bytes`, если этот предел не равен нулю. `container_memory_working_set_bytes` вычисляется путем вычитания неактивной файловой памяти (`total_inactive_file` в ядре) из `container_memory_usage_bytes`.

На практике `container_memory_working_set_bytes` наиболее близок к RSS. Также можно следить за `container_memory_usage_bytes`, так как эта метрика включает кеш страниц.

В общем случае мы рекомендуем полагаться на такие метрики, как `process_resident_memory_bytes`, из самого процесса, а не на метрики из контрольных групп cgroups. Если ваши приложения не предоставляют метрик для Prometheus, то cAdvisor может оказаться хорошей временной мерой, чьи метрики с успехом можно использовать для нужд отладки и профилирования.

Метки

Контрольные группы (cgroups) организованы в иерархию с корнем `/`. Метрики для каждой из ваших контрольных групп включают ее подчиненные контрольные группы. Это противоречит обычному правилу, согласно которому сумма или среднее значение метрики имеют самостоятельное значение, и, таким образом, является примером исключения таблицы, как описано во врезке «Исключение таблицы» в главе 5.

В дополнение к метке `id` cAdvisor добавляет дополнительные метки, описывающие контейнеры, если они у него есть. Для контейнеров Docker всегда будут создаваться метки `image` и `name` для конкретного образа и контейнера Docker.

Любые метки метаданных, которые Docker определяет для контейнера, также будут включены с префиксом `container_label_`. Произвольные метки, подобные этим, могут нарушить работу системы мониторинга. Чтобы этого не произошло, их можно удалить с помощью `labeldrop`, как показано в примере 9.1 и как рассказывалось в разделе «`labeldrop` и `labelkeep`» главы 8².

¹ В число отображаемых файлов входят `mmap` и любые библиотеки, используемые процессом. Их объем отображается ядром как `file_mapped`, но не используется экспортером cAdvisor, поэтому стандартный RSS недоступен в cAdvisor.

² Именно особенности поведения cAdvisor стали основной причиной добавления действий `labeldrop` и `labelkeep`.

Пример 9.1 ❖ Использование labeldrop для удаления меток container_label_, экспортируемых cAdvisor

```
scrape_configs:
- job_name: cadvisor
  static_configs:
    - targets:
      - localhost:9090
  metric_relabel_configs:
    - regex: 'container_label_.*'
      action: labeldrop
```

Kubernetes

Kubernetes – популярная платформа оркестрации контейнеров. Как и Prometheus, проект Kubernetes является частью Cloud Native Computing Foundation (CNCF). Здесь мы рассмотрим запуск Prometheus в Kubernetes и использование механизма обнаружения служб в Kubernetes.

Поскольку Kubernetes – это большой и быстро развивающийся проект, мы не будем подробно описывать его. Желающим познакомится с этой платформой поближе мы рекомендуем книгу Брэндана Бернса (Brendan Burns), Джо Беды (Joe Beda), Келси Хайтауэра (Kelsey Hightower) и Лаклана Эвенсона (Lachlan Evenson) «Kubernetes: Up and Running, 3rd Edition» (O'Reilly).

Запуск в Kubernetes

Чтобы продемонстрировать использование Prometheus с Kubernetes, мы будем использовать Minikube (<https://oreil.ly/SI3or>) – инструмент для запуска кластера Kubernetes с одним узлом внутри виртуальной машины.

Выполните действия, описанные в примере 9.2. Мы использовали машину Linux amd64 с уже установленным VirtualBox. Если вы работаете в другой среде, следуйте инструкциям в документации Minikube (<https://oreil.ly/9oMjE>). Здесь мы используем Minikube 1.27.0 и Kubernetes 1.25.0.

Пример 9.2 ❖ Загрузка и запуск Minikube

```
hostname $ curl -LO \
  https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
hostname $ mv minikube-linux-amd64 minikube
hostname $ chmod +x minikube
hostname $ ./minikube start --kubernetes-version=v1.25.0
minikube v1.27.0 on Nixos 22.05 (Quokka)
Starting control plane node minikube in cluster minikube
Pulling base image ...
Downloading Kubernetes v1.25.0 preload ...
Creating docker container (CPUs=2, Memory=7800MB) ...
Preparing Kubernetes v1.25.0 on Docker 20.10.17 ...
```

```
> Generating certificates and keys ...
> Booting up control plane ...
> Configuring RBAC rules ...
Verifying Kubernetes components...
> Using image gcr.io/k8s-minikube/storage-provisioner:v5
Enabled addons: storage-provisioner, default-storageclass
```

В строке `minikube dashboard --url` вы найдете URL дашборда Kubernetes Dashboard, где можно исследовать свой кластер Kubernetes.

Вам также потребуется установить `kubectl` – инструмент командной строки, используемый для взаимодействия с кластерами Kubernetes. В примере 9.3 показано, как установить его и проверить возможность подключения к вашему кластеру Kubernetes.

Пример 9.3 ♦ Загрузка и тестирование kubectl

```
hostname $ wget \
  https://storage.googleapis.com/kubernetes-release/release/v1.25.0/bin/linux
  /amd64/kubectl
hostname $ chmod +x kubectl
hostname $ ./kubectl get services
NAME      TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP  10.96.0.1    <none>        443/TCP   44s
```

В примере 9.4 показано, как запустить пример Prometheus в Minikube. Файл `prometheus-deployment.yml` определяет разрешения, позволяющие серверу Prometheus получить доступ к ресурсам, таким как модули Pod и узлы в кластере. Для хранения файла конфигурации Prometheus будет создан объект `configMap`, для запуска Prometheus – объект `Deployment` и для упрощения доступа к интерфейсу Prometheus – объект `Service`. Последняя команда – `./minikube service` – сообщит вам URL для доступа к пользовательскому интерфейсу Prometheus.

Пример 9.4 ♦ Настройка разрешений и запуск Prometheus в Kubernetes

```
hostname $ ./kubectl apply -f prometheus-deployment.yml
hostname $ ./minikube service prometheus --url
http://192.168.99.100:30114
```

Страница состояния цели должна выглядеть, как показано на рис. 9.2. Файл `prometheus-deployment.yml` можно найти на GitHub (<https://oreil.ly/Qdwvp>).

Эта простая конфигурация Kubernetes предназначена для демонстрации основных идей мониторинга в Kubernetes. Не используйте ее в промышленном окружении, потому что она теряет все данные при каждом перезапуске Prometheus.

The screenshot shows the Prometheus Targets page with the following details:

- cadvisor (1/1 up)**: https://192.168.49.2:10250/metrics/cadvisor, State: UP, Labels: instance="minikube", job="cadvisor". Last Scrape: 1.666s ago, Scrape Duration: 32.226ms.
- k8apiserver (1/1 up)**: https://192.168.49.2:8443/metrics, State: UP, Labels: instance="192.168.49.2:8443", job="k8apiserver". Last Scrape: 5.985s ago, Scrape Duration: 148.972ms.
- k8pods (1/1 up)**: http://172.17.0.2:9153/metrics, State: UP, Labels: instance="172.17.0.2:9153", job="coredns". Last Scrape: 620.000ms ago, Scrape Duration: 4.766ms.
- k8services (1/1 up)**: http://172.17.0.5:9090/metrics, State: UP, Labels: instance="172.17.0.5:9090", job="prometheus". Last Scrape: 7.376s ago, Scrape Duration: 7.517ms.
- kubelet (1/1 up)**: https://192.168.49.2:10250/metrics/kubelet, State: UP, Labels: instance="192.168.49.2:10250", job="kubelet". Last Scrape: 1.666s ago, Scrape Duration: 32.226ms.

Рис. 9.2 ✦ Цели в примере запуска Prometheus в Kubernetes

Обнаружение служб

В настоящее время в Kubernetes существует шесть разных механизмов обнаружения служб, которые можно использовать с Prometheus: *node*, *endpoints*¹, *endpointslice*, *service*, *pod* и *ingress*. Для обнаружения целей Prometheus использует Kubernetes API.

Node

Механизм обнаружения служб *node* применяется для обнаружения узлов, составляющих кластер Kubernetes, и вы будете использовать его для мониторинга инфраструктуры Kubernetes. На каждом узле выполняется агент *Kubelet*, и вы должны настроить его опрос для мониторинга работоспособности кластера Kubernetes (пример 9.5).

¹ Механизм *endpoints* считается устаревшим, и вместо него рекомендуется использовать *endpointslice*.

Пример 9.5 ❖ Фрагмент prometheus.yml с настройками опроса Kubelet

```

scrape_configs:
- job_name: 'kubelet'
  kubernetes_sd_configs:
  - role: node
    scheme: https
    tls_config:
      ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      insecure_skip_verify: true
    authorization:
      credentials_file: /var/run/secrets/kubernetes.io/serviceaccount/token

```

В примере 9.5 показана конфигурация Prometheus для опроса Kubelet. Посмотрим эту конфигурацию подробнее:

```
job_name: 'kubelet'
```

Задает значение по умолчанию для метки `job`, и, поскольку настройки `label_configs` отсутствуют, метка `job` получит значение `kubelet`¹:

```

kubernetes_sd_configs:
- role: node

```

Единый механизм обнаружения служб в Kubernetes предоставляется с ролью `node`. Она обнаруживает одну цель для каждого из агентов Kubelet. Поскольку Prometheus работает внутри кластера, обнаружение служб в Kubernetes использует аутентификацию через Kubernetes API:

```

scheme: https
tls_config:
  ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
  insecure_skip_verify: true
authorization:
  credentials_file: /var/run/secrets/kubernetes.io/serviceaccount/token

```

Kubelet обслуживает конечную точку `/metrics` через HTTPS, поэтому мы должны задать параметр `scheme`. Кластеры Kubernetes обычно имеют свои центры сертификации (certificate authority) для подписи сертификатов TLS, и в параметре `ca_file` определяется, где процедура опроса сможет их найти. К сожалению, Minikube не совсем правильно понимает эти настройки, поэтому для обхода проверок безопасности требуется добавить параметр `insecure_skip_verify`.

Блок `authorization` и параметр `credentials_file` заставляют Prometheus использовать токен учетной записи службы при опросе целей.

Возвращаемая цель указывает на Kubelet, а аутентификация/авторизация отключены в этой конфигурации Minikube, поэтому дополнительных настроек не требуется.

¹ Мы не используем значение `node` для метки `job`, как это обычно принято в Node Exporter.



Раздел `tls_config` в конфигурации опроса содержит настройки TLS. Конфигурация `kubernetes_sd_configs` тоже имеет раздел `tls_config` с настройками TLS, определяющими параметры взаимодействия механизма обнаружения служб с Kubernetes API.

В наборе метаданных доступны аннотации узлов и метки (рис. 9.3). Эти метаданные можно использовать в `label_configs`, чтобы добавить метки, отличающие интересные подмножества узлов, например с разной аппаратной начинкой.

kubelet	show less	
Discovered Labels		Target Labels
<code>_address_="192.168.49.2:10250"</code>		<code>instance="minikube"</code>
<code>_meta_kubernetes_node_address_Hostname="minikube"</code>		<code>job="kubelet"</code>
<code>_meta_kubernetes_node_address_InternalIP="192.168.49.2"</code>		
<code>_meta_kubernetes_node_annotation_kubeadm_alpha_kubernetes_io_cri_socket="unix:///var/run/cri-dockerd.sock"</code>		
<code>_meta_kubernetes_node_annotation_node_alpha_kubernetes_io_ttl="0"</code>		
<code>_meta_kubernetes_node_annotation_volumes_kubernetes_io_controller_managed_attach_detach="true"</code>		
<code>_meta_kubernetes_node_annotationpresent_kubeadm_alpha_kubernetes_io_cri_socket="true"</code>		
<code>_meta_kubernetes_node_annotationpresent_node_alpha_kubernetes_io_ttl="true"</code>		
<code>_meta_kubernetes_node_annotationpresent_volumes_kubernetes_io_controller_managed_attach_detach="true"</code>		
<code>_meta_kubernetes_node_label_beta_kubernetes_io_arch="amd64"</code>		
<code>_meta_kubernetes_node_label_beta_kubernetes_io_os="linux"</code>		
<code>_meta_kubernetes_node_label_kubernetes_io_arch="amd64"</code>		
<code>_meta_kubernetes_node_label_kubernetes_io_hostname="minikube"</code>		
<code>_meta_kubernetes_node_label_kubernetes_io_os="linux"</code>		
<code>_meta_kubernetes_node_label_minikube_k8s_io_commit="4243041b7a72319b9be7842a7d34b6767bbdac2b"</code>		
<code>_meta_kubernetes_node_label_minikube_k8s_io_name="minikube"</code>		
<code>_meta_kubernetes_node_label_minikube_k8s_io_primary="true"</code>		

Рис. 9.3 ♦ Метаданные Kubelet
на странице состояния механизма обнаружения служб в Prometheus

Собственная конечная точка `/metrics` в Kubelet содержит только метрики самого агента Kubelet – она не предоставляет информацию уровня контейнера. Kubelet имеет встроенный cAdvisor, доступный через конечную точку `/metrics/cadvisor`. Для опроса встроенного cAdvisor требуется только добавить `metrics_path` в конфигурацию опроса для Kubelet, как показано в примере 9.6. Встроенный cAdvisor включает метки `namespace` и `pod_name`, характерные для Kubernetes.

Пример 9.6 ♦ Фрагмент `prometheus.yml` с конфигурацией опроса cAdvisor, встроенного в Kubelet

```
- job_name: 'cadvisor'
  kubernetes_sd_configs:
    - role: node
  scheme: https
  tls_config:
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
```

```
insecure_skip_verify: true
authorization:
  credentials_file: /var/run/secrets/kubernetes.io/serviceaccount/token
metrics_path: /metrics/cadvisor
```

Механизм обнаружения служб node можно использовать для мониторинга чего угодно на каждой машине в кластере Kubernetes. Если на узле Minikube запустить Node Exporter, то его опрос можно организовать, например, изменив порт.

Service

Механизм обнаружения служб node применяется для мониторинга инфраструктуры Kubernetes, но он не особенно полезен для мониторинга приложений, работающих в Kubernetes.

В настоящее время поддерживается несколько способов организации приложений в Kubernetes, и пока не существует единого четкого стандарта. Но чаще всего приложения организованы как *службы*, и именно так приложения в Kubernetes находят друг друга.

В Kubernetes существует роль `service`, но часто это не совсем то, что нужно. Роль `service` возвращает одну цель для каждого порта¹ службы. Службы – это в основном балансировщики нагрузки, а опрашивать цели через балансировщики нагрузки неразумно, потому что каждый раз может опрашиваться другой экземпляр приложения. Однако роль `service` может пригодиться для мониторинга служб с неизвестной структурой исключительно для проверки работоспособности.

Endpointslice

Prometheus должен быть настроен так, чтобы иметь цель для каждого экземпляра приложения. Именно это и обеспечивает роль `endpointslice`. Службы основаны на модулях Pod. Модули Pod – это группы взаимосвязанных контейнеров, совместно использующих сеть и хранилище. Для каждого порта службы в Kubernetes обнаружение служб возвращает цель для каждого модуля Pod в этой службе. Кроме того, в качестве целей будут возвращаться любые другие порты модулей Pod.

Это объяснение выглядит немного туманным, поэтому давайте обратимся к примеру. Одной из служб, работающих в Minikube, является служба `kubernetes`, представляющая серверы Kubernetes API. А примере 9.7 показана конфигурация опроса, которая будет обнаруживать и опрашивать серверы API.

Пример 9.7 ♦ Фрагмент `prometheus.yml` с конфигурацией опроса серверов Kubernetes API

```
scrape_configs:
- job_name: 'k8apiserver'
  kubernetes_sd_configs:
```

¹ Служба может иметь несколько портов.

```

- role: endpointslice
scheme: https
tls_config:
  ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
  insecure_skip_verify: true
authorization:
  credentials_file: /var/run/secrets/kubernetes.io/serviceaccount/token
relabel_configs:
- source_labels:
  - __meta_kubernetes_namespace
  - __meta_kubernetes_service_name
  - __meta_kubernetes_endpoint_port_name
action: keep
regex: default;kubernetes;https

```

Рассмотрим эту конфигурацию подробнее.

```
job_name: 'k8apiserver'
```

Метка job получит значение k8apiserver, так как в конфигурации отсутствуют настройки для ее изменения:

```
kubernetes_sd_configs:
- role: endpointslice
```

 Обратите внимание, что здесь используется роль endpointslice, которая заменяет конечные точки. В последних версиях Kubernetes роль endpointys может содержать не более 1000 записей. Чтобы иметь возможность перечислить все цели в будущем, когда ваш проект разрастется, мы рекомендуем использовать роль endpointslices, не имеющую этого ограничения.

В Kubernetes существует единственный механизм обнаружения служб с ролью endpointslice, и он возвращает цель для каждого порта каждого модуля Pod в каждой из ваших служб:

```

scheme: https
tls_config:
  ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
  insecure_skip_verify: true
authorization:
  credentials_file: /var/run/secrets/kubernetes.io/serviceaccount/token

```

Как и в случае с Kubelet, серверы API обслуживаются через HTTPS. Кроме того, требуется аутентификация, которую обеспечивает credentials_file:

```

relabel_configs:
- source_labels:
  - __meta_kubernetes_namespace
  - __meta_kubernetes_service_name
  - __meta_kubernetes_endpointslice_port_name
action: keep
regex: default;kubernetes;https

```

Эта конфигурация изменения меток будет возвращать только цели, находящиеся в пространстве имен `default` и являющиеся частью службы под названием `kubernetes` с портом `https`.

Получившуюся цель можно увидеть на рис. 9.4. Сервер API – особый случай, поэтому метаданных не так много. Все остальные потенциальные цели были отброшены.

k8apiserver	
Discovered Labels	Target Labels
<code>_address="192.168.49.2:8443"</code> <code>_meta_kubernetes_endpointslice_address_type="IPv4"</code> <code>_meta_kubernetes_endpointslice_endpoint_conditions_ready="true"</code> <code>_meta_kubernetes_endpointslice_name="kubernetes"</code> <code>_meta_kubernetes_endpointslice_port="8443"</code> <code>_meta_kubernetes_endpointslice_port_name="https"</code> <code>_meta_kubernetes_endpointslice_port_protocol="TCP"</code> <code>_meta_kubernetes_namespace="default"</code> <code>_meta_kubernetes_service_label_component="apiserver"</code> <code>_meta_kubernetes_service_label_provider="kubernetes"</code> <code>_meta_kubernetes_service_labelpresent_component="true"</code> <code>_meta_kubernetes_service_labelpresent_provider="true"</code> <code>_meta_kubernetes_service_name="kubernetes"</code> <code>_metrics_path="/metrics"</code> <code>_scheme_="https"</code> <code>_scrape_interval_="10s"</code> <code>_scrape_timeout_="10s"</code> <code>job="k8apiserver"</code>	<code>instance="192.168.49.2:8443"</code> <code>job="k8apiserver"</code>

Рис. 9.4 ♦ Метаданные сервера API
на странице состояния механизма обнаружения служб в Prometheus

Опрос серверов API настраивается довольно часто, но все же большее внимание уделяется приложениям. В примере 9.8 показано, как организовать автоматический опрос модулей Pod всех ваших служб.

Пример 9.8 ♦ Фрагмент `prometheus.yml` с конфигурацией опроса модулей Pod во всех службах в Kubernetes, кроме серверов API

```
scrape_configs:
- job_name: 'k8services'
  kubernetes_sd_configs:
    - role: endpointslice
  relabel_configs:
    - source_labels:
      - __meta_kubernetes_namespace
      - __meta_kubernetes_service_name
```

```

    regex: default;kubernetes
- source_labels:
  - __meta_kubernetes_namespace
    regex: default
    action: keep
- source_labels: [__meta_kubernetes_service_name]
  target_label: job

```

И снова рассмотрим эту конфигурацию подробнее.

```

job_name: 'k8services'
kubernetes_sd_configs:
- role: endpointslice

```

Как и в предыдущем примере, эта часть конфигурации определяет имя задания и роль endpointslice в Kubernetes, но это имя задания не становится меткой job из-за последующего изменения метки.

Здесь нет настроек HTTPS, поскольку мы знаем, что все цели доступны по простому протоколу HTTP. Также отсутствует параметр credentials_file, так как аутентификация не требуется, а отправка токена-носителя (bearer token) всем службам может позволить им действовать от вашего имени¹:

```

relabel_configs:
- source_labels:
  - __meta_kubernetes_namespace
  - __meta_kubernetes_service_name
    regex: default;kubernetes
    action: drop
- source_labels:
  - __meta_kubernetes_namespace
    regex: default
    action: keep

```

Мы исключили сервер API, так как его опрос настроен в другой конфигурации. Мы также ограничились пространством имен default, в котором запускаем приложения²:

```

- source_labels: [__meta_kubernetes_service_name]
  target_label: job

```

Это действие изменения меток получает имя службы Kubernetes и преобразует его в метку job. Имя задания job_name, указанное в конфигурации опроса, является значением по умолчанию и не применяется.

Так можно заставить Prometheus автоматически обнаруживать новые службы и опрашивать их, присваивая значимую метку job. В данном случае это всего лишь сам Prometheus, как показано на рис. 9.5.

¹ Это также относится к базовой аутентификации, но не к механизму запрос–ответ, такому как аутентификация клиента с использованием сертификата TLS.

² И чтобы не было путаницы с примером 9.10, так как kube-dns находится в пространстве имен kube-system.

k8services	
Discovered Labels	Target Labels
<code>_address__=“172.17.0.4:9090”</code>	<code>instance=“172.17.0.4:9090”</code>
<code>_meta_kubernetes_endpointslice_address_target_kind=“Pod”</code>	<code>job=“prometheus”</code>
<code>_meta_kubernetes_endpointslice_address_target_name=“prometheus-56d54d7555-jjc4t”</code>	
<code>_meta_kubernetes_endpointslice_address_type=“IPv4”</code>	
<code>_meta_kubernetes_endpointslice_endpoint_conditions_ready=“true”</code>	
<code>_meta_kubernetes_endpointslice_name=“prometheus-q7nvh”</code>	
<code>_meta_kubernetes_endpointslice_port=“9090”</code>	
<code>_meta_kubernetes_endpointslice_port_protocol=“TCP”</code>	
<code>_meta_kubernetes_namespace=“default”</code>	
<code>_meta_kubernetes_pod_container_name=“prometheus”</code>	
<code>_meta_kubernetes_pod_container_port_name=“default”</code>	
<code>_meta_kubernetes_pod_container_port_number=“9090”</code>	
<code>_meta_kubernetes_pod_container_port_protocol=“TCP”</code>	
<code>_meta_kubernetes_pod_controller_kind=“ReplicaSet”</code>	
<code>_meta_kubernetes_pod_controller_name=“prometheus-56d54d7555”</code>	
<code>_meta_kubernetes_pod_host_ip=“192.168.49.2”</code>	
<code>_meta_kubernetes_pod_ip=“172.17.0.4”</code>	
<code>_meta_kubernetes_pod_label_app=“prometheus”</code>	
<code>_meta_kubernetes_pod_label_template_hash=“56d54d7555”</code>	
<code>_meta_kubernetes_pod_labelpresent_app=“true”</code>	

Рис. 9.5 ♦ Prometheus автоматически обнаружил самого себя с помощью механизма обнаружения служб endpointslice

Можете пойти еще дальше и использовать изменение меток для создания дополнительных меток из метаданных службы или модуля Pod или даже преобразовать аннотации Kubernetes в `__scheme__` или `__metrics_path__`, как показано в примере 9.9. Эта конфигурация отыскивает и использует аннотации `prometheus.io/scheme`, `prometheus.io/path` и `prometheus.io/port`, если они есть¹.

Пример 9.9 ♦ Использование аннотаций служб Kubernetes для получения схемы, пути и порта целей

```
relabel_configs:
- source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scheme]
  regex: (.+)
  target_label: __scheme__
- source_labels: [__meta_kubernetes_service_annotation_prometheus_io_path]
  regex: (.+)
  target_label: __metrics_path__
- source_labels:
```

¹ Косая черта недопустима в именах меток, поэтому она заменяется символом подчеркивания.

```

- __address__
- __meta_kubernetes_service_annotation_prometheus_io_port
  regex: ([^:]+)(:\d+)?;(\d+)
  replacement: ${1}:${3}
  target_label: __address__

```

Эта конфигурация ограничивается мониторингом только одного порта для каждой службы. Однако вы можете добавить обработку аннотации `prometheus.io/port2`, `prometheus.io/port3` и т. д. для любого количества портов.

Pod

Механизм обнаружения служб `endpointslice` отлично подходит для мониторинга основных процессов служб, но не позволяет обнаружить модули Pod, не являющиеся частью служб.

Роль `pod` обнаруживает модули Pod и возвращает цель для каждого порта каждого модуля Pod. Этот механизм работает с модулями, поэтому метаданные службы, такие как метки и аннотации, ему недоступны, так как модули Pod не знают, членами каких служб они являются. Зато он дает доступ ко всем метаданным модулей Pod. Использование этих метаданных сводится к вопросу о том, какие соглашения вы предполагаете использовать. Экосистема Kubernetes быстро развивается, и единого стандарта пока нет.

Можно принять соглашение о том, что каждый модуль Pod должен быть частью какой-то службы, а затем использовать роль `endpointslice` для обнаружения служб. Можно принять соглашение о том, что каждый модуль Pod имеет метку, указывающую на (единственную) службу Kubernetes, частью которой он является, и для обнаружения служб использовать роль `pod`. Поскольку у всех портов есть имена, можно использовать соглашение, чтобы все порты с префиксом `rom-` опрашивать с помощью HTTP, а порты `rom-https` – с помощью HTTPS.

Один из компонентов Minikube – `kube-dns`, предоставляющий услугу DNS. Его модуль Pod открывает несколько портов, в том числе порт с именем `metrics`, через который обслуживает метрики Prometheus. В примере 9.10 показано, как обнаружить этот порт и использовать имя контейнера в качестве метки `job` (рис. 9.6).

Пример 9.10 ✧ `prometheus.yml` с настройками для обнаружения всех портов модуля Pod с именем `metrics` и использования имени контейнера в качестве метки `job`

```

scrape_configs:
- job_name: 'k8pods'
  kubernetes_sd_configs:
    - role: pod
  relabel_configs:
    - source_labels: [__meta_kubernetes_pod_container_port_name]
      regex: metrics
      action: keep
    - source_labels: [__meta_kubernetes_pod_container_name]
      target_label: job

```

k8pods <small>show less</small>	
Discovered Labels	Target Labels
<pre>_address__="172.17.0.2:9153" meta_kubernetes_namespace="kube-system" meta_kubernetes_pod_container_init="false" meta_kubernetes_pod_container_name="coredns" meta_kubernetes_pod_container_port_name="metrics" meta_kubernetes_pod_container_port_number="9153" meta_kubernetes_pod_container_port_protocol="TCP" meta_kubernetes_pod_controller_kind="ReplicaSet" meta_kubernetes_pod_controller_name="coredns-565d847f94" meta_kubernetes_pod_host_ip="192.168.49.2" meta_kubernetes_pod_ip="172.17.0.2" meta_kubernetes_pod_label_k8s_app="kube-dns" meta_kubernetes_pod_label_pod_template_hash="565d847f94" meta_kubernetes_pod_labelpresent_k8s_app="true" meta_kubernetes_pod_labelpresent_pod_template_hash="true" meta_kubernetes_pod_name="coredns-565d847f94-sv5wv" meta_kubernetes_pod_node_name="minikube" meta_kubernetes_pod_phase="Running" meta_kubernetes_pod_ready="true" meta_kubernetes_pod_uid="a05af6fa-f738-4f75-bf4b-c38c9940907e" metrics_path__="/metrics" scheme__="http" scrape_interval__="10s"</pre>	<pre>instance="172.17.0.2:9153" job="coredns"</pre>

Рис. 9.6 ♦ Модуль Pod, обнаруженный с помощью механизма обнаружения служб pod

Ingress

Механизм *ingress* – это способ экспортования службы Kubernetes за пределы кластера. Роль *ingress* подобна роли *service* и точно так же в основном является балансировщиком нагрузки. Если служба состоит из нескольких модулей Pod, то это может вызвать проблемы при опросе со стороны Prometheus. Соответственно, эту роль следует использовать только для мониторинга работоспособности служб.

kube-state-metrics

Используя механизмы обнаружения служб в Kubernetes, можно настроить Prometheus на опрос ваших приложений и инфраструктуры Kubernetes, но

при этом вы не будете получать метрики ресурсов Kubernetes, таких как Service, Pod, Deployment и др. Это связано с тем, что такие приложения, как Kubelet и серверы Kubernetes API, должны предоставлять информацию о своей работе, а не передавать свои внутренние структуры данных¹.

По этой причине вы должны получать метрики с другой конечной точки² или, если таковой не существует, использовать экспортёр, извлекающий необходимую информацию. Для Kubernetes таким экспортёром является `kube-state-metrics`.

Чтобы запустить `kube-state-metrics`, выполните шаги, описанные в примере 9.11, а затем откройте в своем браузере страницу `/metrics` по полученному URL. Соответствующий файл `kube-state-metrics.yml` можно найти на GitHub (<https://oreil.ly/xY3SK>).

Пример 9.11 ◆ Запуск `kube-state-metrics`

```
hostname $./kubectl apply -f kube-state-metrics.yml
hostname $./minikube service kube-state-metrics --url
http://192.168.99.100:31774
```

В числе некоторых полезных метрик можно назвать `kube_deployment_spec_replicas` (предполагаемое количество реплик в развертывании), `kube_node_status_condition` (сообщает проблемы с узлами) и `kube_pod_container_status_restarts_total` (количество повторных запусков модуля Pod).

 Метрики из `kube-state-metrics` будут извлекаться автоматически согласно конфигурации опроса в примере 9.8.

`kube-state-metrics` содержит несколько примеров метрик-перечислений и информационных метрик, обсуждавшихся в разделах «Перечисления» и «Информационные метрики» главы 5, в том числе `kube_node_status_condition` и `kube_pod_info` соответственно.

Альтернативные развертывания

Теперь вы знаете, как развернуть Prometheus в Kubernetes. Это довольно простая задача, тем не менее есть несколько проектов, предоставляющих дополнительные ресурсы CRD³ и утилиты, способные облегчить вам жизнь. Их обсуждение выходит за рамки этой книги, поэтому мы просто порекомендуем обратить особое внимание на два из этих проектов:

¹ Другими словами, экспортёр базы данных не должен выгружать содержимое базы данных в виде метрик.

² Например, Kubelet предоставляет метрики cAdvisor через другую конечную точку.

³ Custom Resources Definition – определение пользовательских ресурсов в Kubernetes.

- Prometheus Operator (<https://oreil.ly/8S74Q>) – проект, поддерживаемый сообществом и включающий ресурсы CRD, конфигурации Prometheus и определения его целей;
- Prometheus Community Kubernetes Helm Charts (<https://oreil.ly/Mg824>) – более 30 диаграмм для развертывания компонентов экосистемы Prometheus с помощью Helm.

Теперь, получив представление об использовании Prometheus в контейнерных средах, познакомимся с некоторыми популярными экспортерами, с которыми вы наверняка столкнетесь.

Глава 10

Популярные экспортеры

В главе 7 вы уже познакомились с Node Exporter, но, кроме него, доступны буквально сотни других экспортёров.

Мы не собираемся рассматривать все имеющиеся экспортёры, а просто покажем вам наиболее популярные из них, с которыми вы почти наверняка столкнетесь в своей практике. Это подготовит вас к использованию экспортёров в вашей собственной среде.

В самом простом случае экспортёры работают «из коробки», не требуя дополнительных настроек, как вы уже видели на примере Node Exporter. Обычно вам достаточно будет выполнить минимальную настройку, чтобы сообщить экспортёру, какой экземпляр приложения следует опрашивать. Однако есть экспортёры, требующие расширенной настройки, поскольку данные, с которыми они работают, слишком общие.

Как правило, вы будете запускать по одному экспортёру для каждого экземпляра приложения. Это связано с тем, что Prometheus предполагает прямое инструментирование каждого приложения, чтобы сервер Prometheus мог обнаружить его и напрямую извлечь метрики. Когда прямое инструментирование невозможно, используются экспортёры, которые стремятся максимально точно придерживаться этой архитектуры. Когда экспортёр работает рядом с экземпляром приложения, который он опрашивает, им легче управлять, а возможные сбои ограничиваются изолированным окружением. Некоторые экспортёры нарушают это правило и предлагают возможность опроса нескольких экземпляров, но их тоже можно развертывать, следуя тем же правилам, и использовать методы, показанные в разделе «metric_relabel_configs» главы 8, для удаления любых посторонних меток.

Consul

Мы с вами уже установили и запустили Consul в разделе «Consul» главы 8. Если он все еще работает, то вы можете загрузить и запустить Consul Exporter, как показано в примере 10.1. Поскольку Consul обычно обслуживает порт 8500, вам не придется выполнять дополнительную настройку, так как Consul Exporter использует этот порт по умолчанию.

Пример 10.1 ❖ Загрузка и запуск Consul Exporter

```
hostname $ wget https://github.com/prometheus/consul_exporter/releases/
    download/v0.3.0/consul_exporter-0.8.0.linux-amd64.tar.gz
hostname $ tar -xzf consul_exporter-0.8.0.linux-amd64.tar.gz
hostname $ cd consul_exporter-0.8.0.linux-amd64/
hostname $ ./consul_exporter
msg="Starting consul_exporter" version="(version=0.8.0, branch=HEAD,
    revision=176aef0f2d437e9fd1cb3a9e29dc4730de717e05)"
build_context="(go=go1.17.6, user=root@566e953b1722, date=20220210-16:54:21)"
msg="Listening on address" address=:9107
```

Если в браузере открыть страницу `http://localhost:9107/metrics`, то можно увидеть доступные метрики.

Первая метрика, на которую следует обратить внимание, – это `consul_up`. Некоторые экспортеры возвращают Prometheus ошибку HTTP при сбое выборки данных, из-за чего Prometheus устанавливает метрику `up` в значение 0. Но многие другие экспортеры успешно обрабатывают такие ошибки и предлагают метрики, подобные `consul_up`, чтобы сообщить о наличии проблемы. Соответственно, предусматривая рассылку уведомлений в случае потери работоспособности Consul, нужно проверять обе метрики – `up` и `consul_up`. Если остановить Consul, а затем проверить `/metrics`, то можно увидеть, что значение метрики изменится на 0, а после запуска Consul снова получит значение 1.

`consul_catalog_service_node_healthy` сообщает о работоспособности различных служб на узле Consul, подобно тому как `kube-state-metrics` (обсуждается в разделе «`kube-state-metrics`» главы 9) сообщает о работоспособности узлов и контейнеров в кластере Kubernetes.

`consul_serf_lan_members` – количество агентов Consul в кластере. Можно было бы подумать, что эта метрика должна извлекаться только из лидера кластера Consul, но имейте в виду, что у каждого агента может быть свое представление о количестве членов кластера, особенно в сегментированных сетях. В общем случае такие метрики следует извлекать из всех членов кластера и синтезировать желаемое значение, используя агрегирование в PromQL.

Есть также метрики, характеризующие самого экспортёра Consul Exporter. `consul_exporter_build_info` поставляет информацию о номере версии и сборки. Также существует множество метрик `process_` и `go_` со сведениями про процесс и среде выполнения Go. Они могут пригодиться для отладки проблем с самим Consul Exporter.

В примере 10.2 показана конфигурация с настройками Prometheus для опроса Consul Exporter. Несмотря на то что опрос происходит через экспортёр, мы использовали метку `job` со значением `consul`, так как на самом деле метрики извлекаются из Consul.



Экспортёры можно рассматривать как своеобразные прокси-серверы. Они принимают запрос от Prometheus, извлекают метрики из процесса, преобразуют их в формат, понятный Prometheus, и возвращают результат в Prometheus.

Пример 10.2 ❖ `prometheus.yml` для опроса локального экспортера Consul Exporter

```
global:
  scrape_interval: 10s
scrape_configs:
- job_name: consul
  static_configs:
  - targets:
    - localhost:9107
```

MySQLd

MySQLd – типичный представитель экспортёров. Для демонстрации запустите экземпляр MySQL и настройте учетную запись пользователя Prometheus.

Как это сделать, показано в примере 10.3.

Пример 10.3 ❖ Загрузка и запуск MySQL Exporter

```
hostname $ docker run -it --net=host --rm mysql mysql -h 127.0.0.1 -P 3306
          -uroot -pmy-secret-pw
mysql: [Warning] Using a password on the command line interface can be insecure.
mysql> CREATE USER 'prometheus'@'127.0.0.1' IDENTIFIED BY 'my-secret-prom-pw'
      WITH MAX_USER_CONNECTIONS 3;
Query OK, 0 rows affected (0.03 sec)

mysql> GRANT PROCESS, REPLICATION CLIENT, SELECT ON *.* TO 'prometheus'@'127.0.0.1';
Query OK, 0 rows affected (0.01 sec)
```

Создайте файл `my.cnf` с учетными данными, как показано в примере 10.4.

Пример 10.4 ❖ `~/.my.cnf` с учетными данными Prometheus

```
[client]
user = prometheus
password = my-secret-prom-pw
host = 127.0.0.1
```

Затем загрузите и запустите экспортёра MySQLd Exporter, как показано в примере 10.5.

Пример 10.5 ❖ Загрузка и запуск MySQLd Exporter

```
hostname $ wget https://github.com/prometheus/mysql_exporter/releases/download/
          v0.9.0/mysql_exporter-0.9.0.linux-amd64.tar.gz
hostname $ tar -xzf mysql_exporter-0.9.0.linux-amd64.tar.gz
hostname $ cd mysql_exporter-0.9.0.linux-amd64/
hostname $ ./mysql_exporter
```

Если теперь открыть страницу `http://localhost:9104/metrics`, то можно увидеть создаваемые метрики. По аналогии с метрикой `consul_up` в Consul Exporter существует метрика `mysql_up`, сообщающая, удалось ли связаться с MySQLd.

 Экспортер MySQLd Exporter также работает с производными MySQL, такими как MariaDB.

В списке вы увидите множество метрик, связанных с MySQL, имена которых начинаются с префикса `mysql_global_variables_` (сообщают значения глобальных переменных конфигурации) и `mysql_global_status_` (сообщают текущие значения, возвращаемые запросом SHOW STATUS).

Порты экспортера по умолчанию

Возможно, вы заметили, что Prometheus, Node Exporter, Alertmanager и другие экспортеры, представленные в этой главе, по умолчанию обслуживают порты с одними и теми же номерами.

Раньше, когда количество экспортеров было невелико, большинство из них обслуживало один и тот же порт по умолчанию. Например, экспортеры Node и HAProxy по умолчанию использовали порт 8080. Это раздражало при опробовании или развертывании Prometheus, поэтому была создана вики-страница (https://oreil.ly/Cx_7b), помогающая создателям экспортеров определить, какие номера портов зарезервированы за официальными экспортерами.

Постепенно она превратилась в исчерпывающий список экспортеров, и теперь она служит цели, выходящей за рамки первоначальной.

В примере 10.6 показано, как настроить экспортер MySQLd, чтобы Prometheus опрашивал его так же, как любой другой экспортер.

Пример 10.6 ❖ `prometheus.yml` с конфигурацией опроса локального экспортера MySQLd

```
global:
  scrape_interval: 10s
scrape_configs:
  - job_name: mysqld
    static_configs:
      - targets:
          - localhost:9104
```

Grok Exporter

Не все приложения выдают метрики в форме, которую с помощью экспортера можно преобразовать во что-то понятное Prometheus. Но многие такие при-

ложении создают журналы, записи в которых можно преобразовывать в метрики с помощью Grok Exporter (<https://oreil.ly/6NaQL>)¹. Grok – это средство анализа неструктурированных журналов, обычно используемое в комплексе с Logstash². Grok Exporter использует тот же самый язык шаблонов, что позволяет повторно использовать уже имеющиеся шаблоны.

Скажем, у вас есть простой журнал в файле с именем *example.log*, который выглядит так:

```
GET /foo 1.23
GET /bag 3.2
POST /foo 4.6
```

Его можно преобразовать в метрики с помощью Grok Exporter. Для этого сначала загрузите выпуск Grok Exporter 0.2.8 для Linux amd64 (<https://oreil.ly/StPAZ>) и разархивируйте. Затем создайте файл с именем *grok.yml* и с содержимым, показанным в примере 10.7.

Пример 10.7 ♦ grok.yml с настройками для анализа простого журнала и получения метрик

```
global:
  config_version: 2
input:
  type: file
  path: example.log
  readall: true # В промышленном окружении используйте значение false
grok:
  additional_patterns:
    - 'METHOD [A-Z]+'
    - 'PATH [^ ]+'
    - 'NUMBER [0-9.]+' 
metrics:
  - type: counter
    name: log_http_requests_total
    help: HTTP requests
    match: '%{METHOD} %{PATH:path} %{NUMBER:latency}'
    labels:
      path: '{{.path}}'
  - type: histogram
    name: log_http_request_latency_seconds_total
    help: HTTP request latency
    match: '%{METHOD} %{PATH:path} %{NUMBER:latency}'
    value: '{{.latency}}'
server:
  port: 9144
```

¹ Обратите также внимание на инструмент <https://oreil.ly/1phnz>.

² Буква L в названии стека ELK.

И запустите Grok Exporter:

```
./grok_exporter -config grok.yml
```

Рассмотрим эти настройки подробнее. В начале следует некоторый шаблонный код:

```
global:  
  config_version: 2
```

Далее определяется анализируемый файл. Здесь используется параметр `readall: true`, поэтому вы увидите те же результаты, что и в этом примере. В промышленном окружении настройте в этом параметре значение по умолчанию `false`, чтобы в каждом цикле читалась только конечная часть файла (его «хвост»):

```
input:  
  type: file  
  path: example.log  
  readall: true # В промышленном окружении используйте значение false
```

Grok работает с шаблонами на основе регулярных выражений. Здесь мы определили все шаблоны вручную, чтобы вам было проще понять происходящее, но в своей практике вы можете использовать уже имеющиеся у вас шаблоны:

```
grok:  
  additional_patterns:  
    - 'METHOD [A-Z]+'  
    - 'PATH [^ ]+'  
    - 'NUMBER [0-9.]+'
```

У нас есть две метрики. Первая – это счетчик с именем `log_http_requests_total` и с меткой `path`:

```
metrics:  
  - type: counter  
    name: log_http_requests_total  
    help: HTTP requests  
    match: '%{METHOD} %{PATH:path} %{NUMBER:latency}'  
    labels:  
      path: '{{.path}}'
```

Вторая метрика – гистограмма с именем `log_http_request_latency_seconds_total`, отслеживающая величины задержек. Она не имеет меток:

```
- type: histogram  
  name: log_http_request_latency_seconds_total  
  help: HTTP request latency  
  match: '%{METHOD} %{PATH:path} %{NUMBER:latency}'  
  value: '{{.latency}}'
```

Открыв в браузере страницу <http://localhost:9144>, среди всего прочего вы увидите следующие метрики:

```
# HELP log_http_request_latency_seconds_total HTTP request latency
# TYPE log_http_request_latency_seconds_total histogram
log_http_request_latency_seconds_total_bucket{le="0.005"} 0
log_http_request_latency_seconds_total_bucket{le="0.01"} 0
log_http_request_latency_seconds_total_bucket{le="0.025"} 0
log_http_request_latency_seconds_total_bucket{le="0.05"} 0
log_http_request_latency_seconds_total_bucket{le="0.1"} 1
log_http_request_latency_seconds_total_bucket{le="0.25"} 2
log_http_request_latency_seconds_total_bucket{le="0.5"} 3
log_http_request_latency_seconds_total_bucket{le="1"} 3
log_http_request_latency_seconds_total_bucket{le="2.5"} 3
log_http_request_latency_seconds_total_bucket{le="5"} 3
log_http_request_latency_seconds_total_bucket{le="10"} 3
log_http_request_latency_seconds_total_bucket{le="+Inf"} 3
log_http_request_latency_seconds_total_sum 0.57
log_http_request_latency_seconds_total_count 3
# HELP log_http_requests_total HTTP requests
# TYPE log_http_requests_total counter
log_http_requests_total{path="/bar"} 1
log_http_requests_total{path="/foo"} 2
```

Как видите, Grok Exporter требует больше усилий по настройке, чем обычные экспортеры; почти столько же, как при прямом инструментировании, потому что приходится определить каждую экспортируемую метрику. Обычно для каждого экземпляра приложения запускается свой экземпляр экспортёра Grok Exporter, который опрашивается сервером Prometheus как обычно (см. пример 10.8).

Пример 10.8 ♦ prometheus.yml конфигурация опроса локального Grok Exporter

```
global:
  scrape_interval: 10s
scrape_configs:
  - job_name: grok
    static_configs:
      - targets:
          - localhost:9144
```

Blackbox

В общем случае рекомендуется развертывать экспортёры рядом с каждым экземпляром приложения, но иногда это невозможно по техническим причинам¹. Это типичный случай мониторинга «черного ящика» – системы, о внут-

¹ В отличие от случаев, когда это невозможно по политическим причинам.

реннем устройстве которой практически ничего не известно. Нам нравится думать о мониторинге «черного ящика» как о дымовых тестиах в модульном тестировании; их цель в первую очередь состоит в том, чтобы быстро предупредить, когда что-то пошло не так.

При наблюдении за работоспособностью веб-службы с точки зрения пользователя мониторинг обычно производится с помощью тех же балансировщиков нагрузки и виртуальных IP-адресов, к которым обращается пользователь. Вы не сможете запустить экспортёр на конкретном виртуальном IP-адресе, просто потому что он виртуальный. Нужна другая архитектура.

В Prometheus есть класс экспортёров, обычно называемых экспортёрами в стиле Blackbox или SNMP, которые не могут работать рядом с экземпляром приложения. Blackbox Exporter обычно запускается где-то в другом месте в сети и не привязан к конкретному экземпляру приложения. Что касается экспортёра SNMP¹ Exporter, запустить свой собственный код на сетевом устройстве возможно в крайне редких случаях, но если имеется такая возможность, то, скорее всего, вы использовали бы Node Exporter.

Так чем же отличаются экспортёры в стиле Blackbox или в стиле SNMP? Вместо настройки взаимодействия только с одной целью они принимают цель как параметр URL. Все остальные настройки определяются на стороне экспортёра как обычно. Они точно так же используют Prometheus для обнаружения служб, а также преобразуют метрики в форму, понятную Prometheus.

Blackbox Exporter позволяет выполнять зондирование ICMP, TCP, HTTP и DNS. Мы покажем, как это делается, но сначала запустим Blackbox Exporter, как показано в примере 10.9.

Пример 10.9 ♦ Загрузка и запуск Blackbox Exporter

```
hostname $ wget https://github.com/prometheus/blackbox_exporter/releases/download/
          v0.22.0/blackbox_exporter-0.22.0.linux-amd64.tar.gz
hostname $ tar -xzf blackbox_exporter-0.22.0.linux-amd64.tar.gz
hostname $ cd blackbox_exporter-0.22.0.linux-amd64/
hostname $ sudo ./blackbox_exporter
msg="Starting blackbox_exporter" version="(version=0.22.0,
branch=HEAD, revision=0bbd65d1264722f7afb87a72ec4128b9214e5840)"
msg="Loaded config file"
msg="Listening on address" address=:9115
```

Открыв страницу <http://localhost:9115> в браузере, вы должны увидеть страницу состояния, подобную той, что показана на рис. 10.1.

ICMP

Протокол управляющих сообщений интернета (Internet Control Message Protocol, ICMP) является частью протокола интернета (Internet Protocol, IP).

¹ Simple Network Management Protocol (простой протокол управления сетью) – стандарт для (кроме всего прочего) экспортации метрик сетевых устройствах. Его иногда можно встретить на разных сетевых устройствах.

В контексте экспортера Blackbox основной интерес представляют **эхо-запросы** и **эхо-ответы**, более известные как *ping*¹.

- ❶ ICMP использует низкоуровневые сокеты, поэтому ему требуется больше привилегий, чем обычному экспортеру. По этой причине в примере 10.9 используется sudo. В Linux вместо этого можно предоставить Blackbox Exporter возможность CAP_NET_RAW.

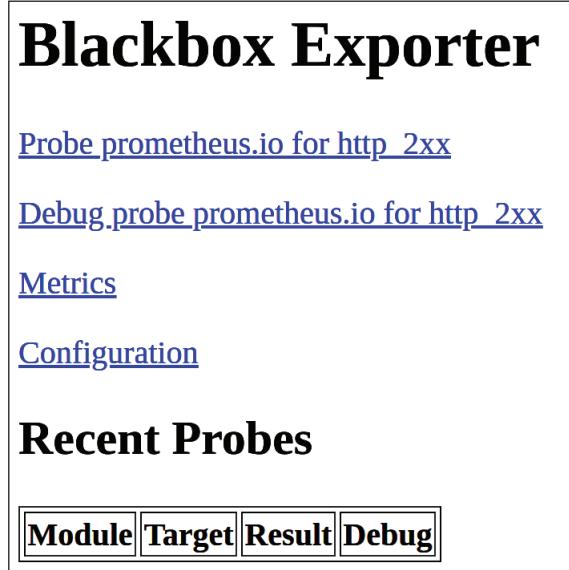


Рис. 10.1 ♦ Страница состояния экспортера Blackbox Exporter

Для начала попросим Blackbox Exporter прозондировать локальный хост, открыв страницу `http://localhost:9115/probe?module=icmp&target=localhost` в браузере. В ответ вы должны получить нечто подобное:

```
# HELP probe_dns_lookup_time_seconds Returns the time taken for probe dns
      lookup in seconds
# TYPE probe_dns_lookup_time_seconds gauge
probe_dns_lookup_time_seconds 0.000580415
# HELP probe_duration_seconds Returns how long the probe took to complete
      in seconds
# TYPE probe_duration_seconds gauge
probe_duration_seconds 0.001044791
# HELP probe_icmp_duration_seconds Duration of icmp request by phase
# TYPE probe_icmp_duration_seconds gauge
probe_icmp_duration_seconds{phase="resolve"} 0.000580415
probe_icmp_duration_seconds{phase="rtt"} 0.000123794
probe_icmp_duration_seconds{phase="setup"} 0.000130416
# HELP probe_icmp_reply_hop_limit Replied packet hop limit (TTL for ipv4)
```

¹ Некоторые эхо-запросы также могут работать через UDP или TCP, но они используются относительно редко.

```
# TYPE probe_icmp_reply_hop_limit gauge
probe_icmp_reply_hop_limit 64
# HELP probe_ip_addr_hash Specifies the hash of IP address. It's useful
# to detect if the IP address changes.
# TYPE probe_ip_addr_hash gauge
probe_ip_addr_hash 1.751717746e+09
# HELP probe_ip_protocol Specifies whether probe ip protocol is IP4 or IP6
# TYPE probe_ip_protocol gauge
probe_ip_protocol 6
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 1
```

Ключевая метрика здесь – это `probe_success`, которая имеет значение 1, если зондирование прошло успешно, и 0 в противном случае. Она похожа на метрику `consul_up` в том смысле, что вы должны проверять две метрики: `up` и `probe_success` при реализации рассылки предупреждений. Пример вы найдете в разделе «`for`» главы 18.

 В пути `/metrics` экспортера Blackbox Exporter доступны метрики, характеризующие работу самого экспортера Blackbox, например сколько процессорного времени он потребил. Для зондирования экспортера Blackbox можно использовать путь `/probe`.

Есть и другие полезные метрики: `probe_ip_protocol` сообщает используемый IP-протокол, в данном случае IPv4; `probe_ip_addr_hash` – это хеш IP-адреса и может использоваться для обнаружения его изменения; и `probe_duration_seconds` – время, затраченное на зондирование, включая разрешение DNS.

 Разрешение имен, используемое в Prometheus и Blackbox Exporter, – это разрешение DNS, а не системный вызов `gethostbyname`. Другие потенциальные источники разрешения имен, такие как `/etc/hosts` и `nsswitch.conf`, Blackbox Exporter не рассматривает. Это может привести к тому, что команда `ping` сработает, но Blackbox Exporter потерпит неудачу из-за невозможности разрешить свою цель через DNS.

Если заглянуть внутрь `blackbox.yml`, то можно увидеть модуль `icmp`:

```
icmp:
  prober: icmp
```

Это тот самый модуль `icmp`, к которому мы обратились с помощью параметра `?module=icmp` в URL. Он использует зонд `icmp` без дополнительных параметров. Протокол ICMP довольно прост, поэтому только в особых случаях вам может понадобиться указать `dont_fragment` или `payload_size`.

Вы также можете попробовать прозондировать другие цели. Например, чтобы исследовать `google.com`, откройте в браузере страницу `http://localhost:9115/probe?module=icmp&target=www.google.com`. Для зонда `icmp` параметр `target` в URL определяет IP-адрес или имя хоста.

Иногда зондирование завершается неудачей с выводом сообщений, как показано ниже:

```

# HELP probe_dns_lookup_time_seconds Returns the time taken for probe dns
#   lookup in seconds
# TYPE probe_dns_lookup_time_seconds gauge
probe_dns_lookup_time_seconds 0.018805905
# HELP probe_duration_seconds Returns how long the probe took to complete
#   in seconds
# TYPE probe_duration_seconds gauge
probe_duration_seconds 0.019061888
# HELP probe_icmp_duration_seconds Duration of icmp request by phase
# TYPE probe_icmp_duration_seconds gauge
probe_icmp_duration_seconds{phase="resolve"} 0.018805905
probe_icmp_duration_seconds{phase="rtt"} 0
probe_icmp_duration_seconds{phase="setup"} 9.8677e-05
# HELP probe_ip_addr_hash Specifies the hash of IP address. It's useful to
# detect if the IP address changes.
# TYPE probe_ip_addr_hash gauge
probe_ip_addr_hash 4.125764906e+09
# HELP probe_ip_protocol Specifies whether probe ip protocol is IP4 or IP6
# TYPE probe_ip_protocol gauge
probe_ip_protocol 6
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 0

```

Здесь метрика `probe_success` имеет значение 0, что указывает на сбой. Обратите внимание, что `probe_ip_protocol` имеет значение 6, т. е. использован протокол IPv6. В нашем примере мы использовали машину, не поддерживающую IPv6. Но почему Blackbox Exporter использовал IPv6?

Выполнив разрешение цели, Blackbox Exporter отдаст предпочтение адресу IPv6, если он есть; в противном случае будет использоваться адрес IPv4. Сайт `google.com` имеет адреса обоих типов, поэтому Blackbox выбрал IPv6, который не работает на нашей машине.

Увидеть детали происходящего в этом случае можно, если добавить параметр `&debug=true` в конец URL: `http://localhost:9115/probe?module=icmp&target=www.google.com&debug=true`. В результате вы получите примерно такой вывод:

```

Logs for the probe:
... module=icmp target=www.google.com level=info
    msg="Beginning probe" probe=icmp timeout_seconds=119.5
... module=icmp target=www.google.com level=info
    msg="Resolving target address" preferred_ip_protocol=ip6
... module=icmp target=www.google.com level=info
    msg="Resolved target address" ip=2a00:1450:400c:c07::69
... module=icmp target=www.google.com level=info
    msg="Creating socket"
... module=icmp target=www.google.com level=info
    msg="Creating ICMP packet" seq=10 id=3483
... module=icmp target=www.google.com level=info
    msg="Writing out packet"
... module=icmp target=www.google.com level=warn
    msg="Error writing to socket" err="write udp"

```

```
[::]:3->[2a00:1450:400c:c07::69]:0: sendto: network is unreachable"
... module=icmp target=www.google.com level=error
msg="Probe failed" duration_seconds=0.001902969

Metrics that would have been returned:
# HELP probe_dns_lookup_time_seconds Returns the time taken for probe dns
    lookup in seconds
# TYPE probe_dns_lookup_time_seconds gauge
probe_dns_lookup_time_seconds 0.001635165
# HELP probe_duration_seconds Returns how long the probe took to complete
    in seconds
# TYPE probe_duration_seconds gauge
probe_duration_seconds 0.001902969
# HELP probe_icmp_duration_seconds Duration of icmp request by phase
# TYPE probe_icmp_duration_seconds gauge
probe_icmp_duration_seconds{phase="resolve"} 0.001635165
probe_icmp_duration_seconds{phase="rtt"} 0
probe_icmp_duration_seconds{phase="setup"} 9.6612e-05
# HELP probe_ip_addr_hash Specifies the hash of IP address. It's useful to
    detect if the IP address changes.
# TYPE probe_ip_addr_hash gauge
probe_ip_addr_hash 4.142542525e+09
# HELP probe_ip_protocol Specifies whether probe ip protocol is IP4 or IP6
# TYPE probe_ip_protocol gauge
probe_ip_protocol 6
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 0

Module configuration:
prober: icmp
http:
  ip_protocolFallback: true
  followRedirects: true
  enableHttp2: true
tcp:
  ip_protocolFallback: true
icmp:
  ip_protocolFallback: true
  ttl: 64
dns:
  ip_protocolFallback: true
  recursionDesired: true
```

В режиме отладки выводится много подробной информации. Внимательно прочитав ее, можете точно понять, что делает зонд. Ошибка, которая выделена здесь жирным, связана с системным вызовом `sendto`, который не может назначить адрес IPv6. Чтобы устранить эту проблему и придать адресам IPv4 более высокий приоритет, добавьте в `blackbox.yml` новый модуль с параметром `selected_ip_protocol: ipv4`:

```
icmp_ipv4:
  prober: icmp
```

```
icmp:
  preferred_ip_protocol: ip4
```

Если после перезапуска Blackbox Exporter¹ использовать этот модуль в URL, например `http://localhost:9115/probe?module=icmp_ipv4&target=www.google.com`, то зондирование будет осуществляться через IPv4.

TCP

Протокол управления передачей (Transmission Control Protocol, TCP) – это часть «TCP» в аббревиатуре «TCP/IP». Его используют многие стандартные протоколы, включая веб-сайты (HTTP), электронная почта (SMTP), удаленный вход в систему (Telnet и SSH) и чат (IRC). TCP-зонд в Blackbox Exporter позволяет проверять службы TCP и выполнять операции в диалоговом режиме, если службы поддерживают текстовые протоколы управления.

Для начала можно проверить, прослушивает ли ваш локальный SSH-сервер порт 22, открыв в браузере страницу `http://localhost:9115/probe?module=tcp_connect&target=localhost:22`:

```
# HELP probe_dns_lookup_time_seconds Returns the time taken for probe dns lookup
#      in seconds
# TYPE probe_dns_lookup_time_seconds gauge
probe_dns_lookup_time_seconds 0.000202381
# HELP probe_duration_seconds Returns how long the probe took to complete in
#      seconds
# TYPE probe_duration_seconds gauge
probe_duration_seconds 0.000881654
# HELP probe_failed_due_to_regex Indicates if probe failed due to regex
# TYPE probe_failed_due_to_regex gauge
probe_failed_due_to_regex 0
# HELP probe_ip_protocol Specifies whether probe ip protocol is IP4 or IP6
# TYPE probe_ip_protocol gauge
probe_ip_protocol 4
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 1
```

Очень похоже на метрики, созданные зондом ICMP, и, как видите, зондирование увенчалось успехом, потому что `probe_success` получила значение 1. Вот определение модуля `tcp_connect` в `blackbox.yml`:

```
tcp_connect:
  prober: tcp
```

Он попытается подключиться к цели и сразу после подключения закрывает соединение. Модуль `ssh_banner` идет дальше, проверяя конкретный ответ удаленного сервера:

¹ Как и в случае с Prometheus, экспортеру Blackbox можно отправить сигнал SIGHUP, чтобы он перезагрузил конфигурацию.

```
ssh_banner:  
  prober: tcp  
  tcp:  
    query_response:  
      - expect: "SSH-2.0-"
```

Поскольку в самом начале сеанса SSH выводится обычный текст, можно проверить эту часть протокола с помощью зонда `tcp`. Это лучше, чем `tcp_connect`, так проверяется не только факт открытия порта TCP, но и конкретный ответ сервера SSH.

Если сервер вернул что-то другое, то его ответ не будет совпадать с ожидаемым, и `probe_success` получит значение 0. Кроме того, метрика `probe_failed_due_to_regex` получит значение 1. Поскольку Prometheus – это система, основанная на метриках, полный вывод отладки нельзя сохранить, так как это будет событие журналирования¹. Тем не менее Blackbox Exporter может предоставить небольшое количество метрик, которые помогут вам определить, что пошло не так.

- ✓ Если обнаружится, что для каждой службы требуется использовать отдельный модуль, то подумайте о возможности стандартизации проверки работоспособности разных служб. Если служба предоставляет страницу `/metrics`, то нет никакой необходимости использовать простые проверки подключения в Blackbox Exporter, потому что Prometheus уже имеет такие проверки.

Зонд `tcp` также может подключаться через TLS. Добавьте `tcp_connect_tls` в файл `blackbox.yml` со следующей конфигурацией:

```
tcp_connect_tls:  
  prober: tcp  
  tcp:  
    tls: true
```

Открыв страницу `http://localhost:9115/probe?module=tcp_connect_tls&target=www.oreilly.com:443` после перезапуска Blackbox Exporter, можно проверить, доступен ли веб-сайт O'Reilly по HTTPS². Для зонда `tcp` в параметре `target` передается имя хоста или IP-адрес и номер порта через двоеточие.

В выводе среди метрик можно заметить:

```
# HELP probe_ssl_last_chain_expiry_timestamp_seconds Returns last SSL chain  
expiry in timestamp  
# TYPE probe_ssl_last_chain_expiry_timestamp_seconds gauge  
probe_ssl_last_chain_expiry_timestamp_seconds 1.686095999e+09
```

Как побочный эффект зондирования создается метрика `probe_ssl_last_chain_expiry_timestamp_seconds`, указывающая, когда истечет срок действия вашего сертификата TLS/SSL³. Ее можно использовать, чтобы выявить сер-

¹ Однако отладочная информация для зондов, выполнявшихся последними, доступна на странице состояния Blackbox Exporter.

² 443 – это стандартный порт для HTTPS.

³ Точнее, первого сертификата в вашей цепочке сертификатов.

тификаты с истекающим сроком действия до того, как они перестанут работать.

Протокол HTTP является текстовым протоколом¹, и с ним можно использовать зонд `tcp`, однако существует зонд `http`, более подходящий для этой цели.

HTTP

Протокол передачи гипертекста (HyperText Transfer Protocol, HTTP) является основой современной Всемирной паутины и, вероятно, используется большинством предоставляемых вами служб. Хотя в большинстве случаев мониторинг веб-приложений лучше выполнять с помощью Prometheus, извлекающего метрики через HTTP, иногда все же может потребоваться осуществлять мониторинг ваших HTTP-служб методом «черного ящика».

Зонд `http` принимает URL², переданный в параметре `target`. Например, открыв страницу `http://localhost:9115/probe?module=http_2xx&target=https://www.oreilly.com/`, можно проверить доступность веб-сайта O'Reilly по протоколу HTTPS, используя³ модуль `http_2xx`, и получить следующий вывод:

```
# HELP probe_dns_lookup_time_seconds Returns the time taken for probe
      dns lookup in seconds
# TYPE probe_dns_lookup_time_seconds gauge
probe_dns_lookup_time_seconds 0.001481084
# HELP probe_duration_seconds Returns how long the probe took to complete
      in seconds
# TYPE probe_duration_seconds gauge
probe_duration_seconds 0.165316519
# HELP probe_failed_due_to_regex Indicates if probe failed due to regex
# TYPE probe_failed_due_to_regex gauge
probe_failed_due_to_regex 0
# HELP probe_http_content_length Length of http content response
# TYPE probe_http_content_length gauge
probe_http_content_length -1
# HELP probe_http_duration_seconds Duration of http request by phase, summed
      over all redirects
# TYPE probe_http_duration_seconds gauge
probe_http_duration_seconds{phase="connect"} 0.02226464
probe_http_duration_seconds{phase="processing"} 0.05238605
probe_http_duration_seconds{phase="resolve"} 0.001481084
probe_http_duration_seconds{phase="tls"} 0.043717698
probe_http_duration_seconds{phase="transfer"} 0.044905889
# HELP probe_http_last_modified_timestamp_seconds Returns the Last-Modified
      HTTP response header in unixtime
# TYPE probe_http_last_modified_timestamp_seconds gauge
```

¹ По крайней мере, для HTTP до версии 2.0.

² В который можно даже включить дополнительные параметры, правильно закодировав их.

³ Кстати, модуль `http_2xx` используется по умолчанию, если не указать его в параметре URL.

```
probe_http_last_modified_timestamp_seconds 1.665390603e+09
# HELP probe_http_redirects The number of redirects
# TYPE probe_http_redirects gauge
probe_http_redirects 0
# HELP probe_http_ssl Indicates if SSL was used for the final redirect
# TYPE probe_http_ssl gauge
probe_http_ssl 1
# HELP probe_http_status_code Response HTTP status code
# TYPE probe_http_status_code gauge
probe_http_status_code 200
# HELP probe_http_uncompressed_body_length Length of uncompressed response body
# TYPE probe_http_uncompressed_body_length gauge
probe_http_uncompressed_body_length 75719
# HELP probe_http_version Returns the version of HTTP of the probe response
# TYPE probe_http_version gauge
probe_http_version 2
# HELP probe_ip_addr_hash Specifies the hash of IP address. It's useful to
      detect if the IP address changes.
# TYPE probe_ip_addr_hash gauge
probe_ip_addr_hash 1.793027101e+09
# HELP probe_ip_protocol Specifies whether probe ip protocol is IP4 or IP6
# TYPE probe_ip_protocol gauge
probe_ip_protocol 4
# HELP probe_ssl_earliest_cert_expiry Returns earliest SSL cert expiry
      in unixtime
# TYPE probe_ssl_earliest_cert_expiry gauge
probe_ssl_earliest_cert_expiry 1.697068799e+09
# HELP probe_ssl_last_chain_expiry_timestamp_seconds Returns last SSL chain
      expiry in timestamp seconds
# TYPE probe_ssl_last_chain_expiry_timestamp_seconds gauge
probe_ssl_last_chain_expiry_timestamp_seconds 1.697068799e+09
# HELP probe_ssl_last_chain_info Contains SSL leaf certificate information
# TYPE probe_ssl_last_chain_info gauge
probe_ssl_last_chain_info{fingerprint_sha256="849c8863b"} 1
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 1
# HELP probe_tls_version_info Contains the TLS version used
# TYPE probe_tls_version_info gauge
probe_tls_version_info{version="TLS 1.3"} 1
```

Видите, здесь присутствует метрика `probe_success`, а также ряд других метрик, которые могут пригодиться для отладки, таких как код состояния, версия HTTP и продолжительность различных фаз запроса.

Зонд `http` имеет множество параметров, которые влияют на выполнение запроса и определяют, можно ли считать ответ успешным. С их помощью можно настроить HTTP-аутентификацию, заголовки, тело POST-запроса, а затем в ответе проверить код состояния, версию HTTP и содержимое его тела.

Например, можно проверить, перенаправляются ли пользователи <http://www.oreilly.com> на веб-сайт через HTTPS с кодом состояния 200 и присутствует ли «Prometheus» в теле ответа. Для этого можно создать модуль, как показано ниже:

```
http_200_ssl_prometheus:
  prober: http
  http:
    valid_status_codes: [200]
    fail_if_not_ssl: true
    fail_if_not_matches_regex:
      - oreillymedia
```

Открыв страницу `http://localhost:9115/probe?module=http_200_ssl_prometheus&target=https://oreilly.com` в своем браузере, вы должны увидеть, что проверка прошла успешно, потому что метрика `probe_success` получила значение 1. Аналогичный запрос можно использовать для проверки <http://prometheus.io>, если открыть в браузере страницу `http://localhost:9115/probe?module=http_200_ssl_prometheus&target=http://prometheus.io`¹.

! Blackbox Exporter поддерживает HTTP-перенаправление², но не все функции работают идеально, когда производится перенаправление.

Этот пример немного надуманный, тем не менее каждый модуль Blackbox Exporter представляет собой особый тест, который можно запустить для проверки разных целей, указав разные значения в параметре `target` в URL, как было показано выше на примерах целей `http://www.oreilly.com` и `http://prometheus.io`. Например, можно проверить, возвращает ли правильный результат каждый экземпляр внешнего интерфейса, обслуживающий ваш веб-сайт. Если для разных служб нужны разные тесты, то можно создать модули для каждой из них. Но имейте в виду, что невозможно переопределить модули через параметры URL, так как это превратило бы Blackbox Exporter в открытый прокси-сервер³ и запутало разделение обязанностей между Prometheus и экспортёрами.

Зонд `http` обладает самыми обширными настройками из всех зондов в Blackbox Exporter (параметры вы найдете в документации <https://oreil.ly/Jj0mf>). Но, несмотря на свою гибкость, Blackbox Exporter поддерживает далеко не все возможные варианты использования, так как, в конце концов, это относительно простой HTTP-зонд. Если вам потребуется что-то более мощное, то напишите свой собственный экспортёр или воспользуйтесь существующими экспортёрами, такими как WebDriver Exporter (<https://oreil.ly/qTbHY>), который имитирует посещение URL браузером.

DNS

Зонд `dns` предназначен в первую очередь для тестирования серверов DNS. С его помощью, например, можно проверить, все ли ваши реплики DNS возвращают результаты.

¹ Предполагается, что ваша машина поддерживает IPv6; если нет, то добавьте `preferred_ip_protocol: ip4`.

² Если в параметре `follow_redirects` установлено значение `false`.

³ Что очень неразумно с точки зрения безопасности.

Чтобы проверить, отвечают ли ваши серверы DNS по TCP¹, можно создать модуль в файле *blackbox.yml*:

```
dns_tcp:  
  prober: dns  
  dns:  
    transport_protocol: "tcp"  
    query_name: "www.prometheus.io"
```

После перезапуска Blackbox Exporter откройте страницу *http://localhost:9115/probe?module=dns_tcp&target=8.8.8.8*, чтобы проверить, работает ли общедоступная служба DNS компании Google² через TCP. Обратите внимание, что параметр *target* в URL – это IP-адрес проверяемого сервера DNS, а *query_name* – это DNS-запрос, отправляемый серверу DNS. Этот URL действует подобно команде `dig -tcp @8.8.8.8 www.prometheus.io`.

Для зонда *dns* параметр *target* в URL может содержать IP-адрес или имя хоста, за которым следует двоеточие, а затем номер порта. Также можно указать только IP-адрес или имя хоста, и в этом случае будет использоваться стандартный порт 53.

Зонд *dns* можно использовать не только для тестирования серверов DNS, но и чтобы убедиться, что операция разрешения DNS возвращает определенные результаты. Нередко возникает желание пойти еще дальше и подключиться к полученной службе через HTTP, TCP или ICMP, и в таких случаях лучше использовать специализированные зонды для этих протоколов, которые по-прежнему выполняют проверку DNS.

Примером использования зонда *dns* для проверки конкретных результатов может быть проверка присутствия ваших записей MX³.

Для этого можно создать модуль в файле *blackbox.yml*, как показано ниже:

```
dns_mx_present_rp_io:  
  prober: dns  
  dns:  
    query_name: "prometheus.io"  
    query_type: "MX"  
    validate_answer_rrs:  
      fail_if_not_matches_regex:  
        - ".+"
```

После перезапуска Blackbox Exporter можно открыть страницу *http://localhost:9115/probe?module=dns_mx_present_rp_io&target=8.8.8.8* и проверить наличие записей MX на `prometheus.io`. Обратите внимание, что, поскольку для каждого модуля указывается *query_name*, вам понадобится создать модуль для каждого проверяемого домена. Здесь мы используем 8.8.8.8, поскольку

¹ Обычно серверы DNS используют UDP, но также могут использовать TCP, например, для возврата объемных ответов. К сожалению, многие администраторы сайтов не знают об этом и блокируют TCP-порт 53, который является портом DNS.

² Доступна на IP-адресах 8.8.8.8, 8.8.4.4, 2001:4860:4860::8888 и 2001:4860:4860::8844.

³ Используется для электронной почты, MX означает Mail eXchanger (почтовый обменник, или сервер электронной почты).

сервер DNS от Google является общедоступным, но вы также можете использовать локальный сервер DNS.

Зонд `dns` позволяет проверить множество аспектов в ответах DNS, таких как авторитетность и дополнительные записи. Узнать больше о них вы сможете в документации (<https://oreil.ly/3pY9E>). Для лучшего понимания DNS мы рекомендуем прочитать RFC 1034 (<https://oreil.ly/y7Rra>) и 1035 (<https://oreil.ly/dtwcY>)¹ или такую книгу, как «DNS and BIND»² Пола Альбитца (Paul Albitz) и Крикет Лью (Cricket Liu), выпущенную издательством O'Reilly.

Настройка Prometheus

Как вы видели, Blackbox Exporter принимает параметры `module` и `target` в URL конечной точки `/probe`. Используя `params` и `metrics_path`, как обсуждалось в разделе «Как извлекать метрики» главы 8, их можно указать в конфигурации опроса, но это будет означать наличие конфигурации опроса для каждой цели, что слишком громоздко, поскольку мешает возможности использовать преимущества обнаружения служб в Prometheus.

Однако эту проблему можно решить с помощью метки `__param_<name>`, используя ее для подготовки параметров URL на этапе изменения меток. Кроме того, метки `instance` и `__address__` различны, как обсуждалось в разделе «`job`, `instance` и `__address__`» главы 8, что позволяет организовать взаимодействие Prometheus с Blackbox Exporter, определяя метку `instance`, ссылающуюся на фактическую цель.

В примере 10.10 показано, как это сделать.

Пример 10.10 ✦ `prometheus.yml` для проверки работоспособности нескольких сайтов

```
scrape_configs:
  - job_name: blackbox
    metrics_path: /probe
    params:
      module: [http_2xx]
    static_configs:
      - targets:
          - http://www.prometheus.io
          - http://www.robustperception.io
          - http://demo.robustperception.io
    relabel_configs:
      - source_labels: [__address__]
        target_label: __param_target
      - source_labels: [__param_target]
        target_label: instance
```

¹ Мы узнали о DNS из этих RFC; они немного устарели, но все еще дают хорошее представление о том, как работает DNS.

² Крикет Ли, Пол Альбитц. DNS и BIND. СПб.: Символ-Плюс, 2008. ISBN: 5-93286-105-3. – Прим. перев.

```
- target_label: __address__  
  replacement: 127.0.0.1:9115
```

Разберем этот пример подробнее:

```
- job_name: 'blackbox'  
  metrics_path: /probe  
  params:  
    module: [http_2xx]
```

Здесь задается значение по умолчанию для метки `job`, путь и один из параметров URL:

```
static_configs:  
- targets:  
  - https://www.prometheus.io  
  - https://www.oreilly.com  
  - https://demo.do.prometheus.io
```

Определено три веб-сайта для мониторинга:

```
relabel_configs:  
- source_labels: [__address__]  
  target_label: __param_target  
- source_labels: [__param_target]  
  target_label: instance  
- target_label: __address__  
  replacement: 127.0.0.1:9115
```

В `relabel_configs` творится основное волшебство. Значение метки `__address__` помещается в параметр `target URL` и в метку `instance`. После этого метка `instance` и параметр `target URL` получают желаемое значение, но `__address__` по-прежнему содержит URL сайта, а не экспортера Blackbox Exporter. Последнее действие записывает в метку `__address__` IP-адрес и номер порта локального экземпляра Blackbox Exporter.

Если запустить Prometheus с этой конфигурацией и открыть страницу состояния **Targets** (Цели), то можно увидеть таблицу, как показано на рис. 10.2. Конечная точка имеет нужные параметры URL, а метка `instance` содержит URL сайта.

! Тот факт, что экспортер Blackbox Exporter находится в состоянии UP, не означает, что проверка прошла успешно. Это лишь признак, что экспортер доступен и был опрошен¹. О работоспособности самого сайта может свидетельствовать только значение 1 в метрике `probe_success`.

Однако для определения целей можно использовать не только `static_configs`, но и любые другие механизмы обнаружения служб (как обсуждалось в главе 8). Например, опираясь на пример 8.19 с конфигурацией опроса экспортёров Node Exporter на всех узлах, зарегистрированных в Consul, мы на-

¹ На самом деле проверка <http://www.prometheus.io> на рис. 10.2 терпит неудачу, потому что на нашей машине не настроена поддержка IPv6.

писали пример 10.11, проверяющий доступность серверов SSH на всех узлах, зарегистрированных в Consul.

The screenshot shows the Prometheus 'Targets' page. At the top, there are tabs for 'All', 'Unhealthy', and 'Expand All'. A search bar and a filter input 'Filter by endpoint or labels' are also present. Below this, a section titled 'blackbox (3/3 up)' shows three healthy targets:

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
<code>http://127.0.0.1:9115/probe</code> <code>module="http_2xx"</code> <code>target="http://www.prometheus.io"</code>	UP	<code>instance="http://www.prometheus.io"</code> <code>job="blackbox"</code>	931.000 ms ago	297.940ms	
<code>http://127.0.0.1:9115/probe</code> <code>module="http_2xx"</code> <code>target="http://www.robustperception.io"</code>	UP	<code>instance="http://www.robustperception.io"</code> <code>job="blackbox"</code>	18.461s ago	685.287ms	
<code>http://127.0.0.1:9115/probe</code> <code>module="http_2xx"</code> <code>target="http://demo.robustperception.io"</code>	UP	<code>instance="http://demo.robustperception.io"</code> <code>job="blackbox"</code>	33.408s ago	261.493ms	

Рис. 10.2 ♦ Страница состояния экспортера Blackbox Exporter

Пример 10.11 ♦ Проверка серверов SSH на всех узлах, зарегистрированных в Consul

```
scrape_configs:
- job_name: node
  metrics_path: /probe
  params:
    module: [ssh_banner]
  consul_sd_configs:
  - server: 'localhost:8500'
  relabel_configs:
  - source_labels: [__meta_consul_address]
    regex: '(.*)'
    replacement: '${1}:22'
    target_label: __param_target
  - source_labels: [__param_target]
    target_label: instance
  - target_label: __address__
    replacement: 127.0.0.1:9115
```

Этот подход позволяет использовать механизмы обнаружения служб не только для опроса `/metrics`, но и для мониторинга приложений по принципу «черного ящика».

Тайм-ауты Blackbox

Возможно, вам интересно узнать, как настроить тайм-ауты для зондов. Сразу же отметим, что Blackbox автоматически определяет тайм-ауты на основе параметра `scrape_timeout` в Prometheus.

Prometheus посылает HTTP-заголовок `X-Prometheus-Scrape-Timeout-Seconds` при каждом опросе, а Blackbox Exporter использует его для определения тайм-аута за вычетом буфера¹. В результате Blackbox Exporter обычно возвращает некоторые метрики, которые будут полезны при отладке взаимодействий с медленной целью.

Также есть возможность дополнительно уменьшить тайм-аут с помощью параметра `timeout` в файле `blackbox.yml`.

Теперь, получив представление о типах доступных экспортёров, мы готовы перейти к изучению вопроса извлечения метрик из существующих систем мониторинга.

¹ Задается флагом командной строки `--timeout-offset`.

Глава 11

Работа с другими системами мониторинга

В идеальном мире все ваши приложения будут напрямую экспортить метрики Prometheus, но в реальном мире такое едва ли случится. К моменту, когда вы обратите свои взоры на Prometheus, у вас уже могут иметься другие системы мониторинга, и вы вряд ли сможете в один прекрасный день выполнить большой переход на Prometheus.

Самое замечательное, что среди сотен экспортёров для Prometheus есть экспортёры, способные преобразовывать данные из других систем мониторинга в формат Prometheus. Конечно, идеальной конечной целью для вас мог бы быть полный переход на Prometheus, но такой переход нечасто удается совершить одномоментно, и в этом случае вам помогут экспортёры, подобные описанным в этой главе, позволив выполнить такой переход постепенно.

Другие системы мониторинга

Системы мониторинга различаются по степени совместимости с Prometheus; одни требуют серьезных усилий для их интеграции с Prometheus, тогда как другие не требуют почти ничего. Например, InfluxDB имеет модель данных, очень похожую на Prometheus, поэтому можно заставить приложение передавать линейный протокол InfluxDB в экспортёр InfluxDB Exporter (<https://oreil.ly/NPmXE>), который затем может быть опрошен сервером Prometheus.

Другие системы, такие как collectd, не имеют меток, но их метрики можно автоматически преобразовывать в метрики Prometheus без дополнительной настройки с помощью Collectd Exporter (<https://oreil.ly/ErtjW>). Начиная с версии 5.7, система collectd даже включает этот экспортёр в плагин Write Prometheus (<https://oreil.ly/mWhRg>).

Но не все системы мониторинга имеют модели данных, которые можно автоматически преобразовать в метрики Prometheus. Например, Graphite не поддерживает метки вида ключ–значение, но некоторые метки можно

извлечь с помощью Graphite Exporter¹ (<https://oreil.ly/6ah0Q>). Система StatsD имеет похожую модель строк с точками, что и Graphite, но StatsD использует события, а не метрики, поэтому StatsD Exporter (<https://oreil.ly/ECuBA>) преобразует события в метрики, а также может извлекать метки.

В пространстве Java/JVM для представления метрик часто используется стандарт JMX (Java Management eXtensions), но порядок использования этого механизма сильно отличается для разных приложений. JMX Exporter (<https://oreil.ly/A549a>) имеет хорошие настройки по умолчанию, но, учитывая отсутствие стандартизации структуры mBean, единственный разумный способ настроить его – использовать регулярные выражения. Хорошая новость состоит в том, что имеется множество примеров конфигураций и JMX Exporter предназначен для работы в роли Java-агента, поэтому вам не придется управлять отдельным процессом экспортёра.

SNMP имеет модель данных, очень близкую к модели Prometheus, и с помощью MIB² метрики SNMP могут автоматически создаваться экспортёром SNMP Exporter (<https://oreil.ly/HW3wu>). Однако этот экспортёр имеет два существенных недостатка. Во-первых, MIB от поставщиков часто отсутствуют в свободном доступе, поэтому вам придется приобрести MIB самостоятельно и использовать *генератор*, входящий в состав SNMP Exporter, для преобразования MIB в форму, понятную экспортёру SNMP Exporter. Во-вторых, многие поставщики следуют букве спецификации SNMP, но не ее духу, поэтому иногда требуется дополнительная настройка и/или переделка с помощью PromQL. Экспортёр SNMP Exporter действует в стиле Blackbox/SNMP, как обсуждалось в разделе «Blackbox» главы 10, поэтому, в отличие от многих других экспортёров, этот экспортёр запускается в единственном экземпляре для каждого сервера Prometheus, а не для каждого экземпляра приложения.

 Протокол SNMP очень многословный. Поэтому рекомендуется располагать экспортёры SNMP Exporter в сети как можно ближе к сетевым устройствам, за которыми они наблюдают. Кроме того, многие SNMP-устройства могут использовать протокол SNMP, но не возвращают метрики за какое-либо разумное время. Возможно, вам придется ограничить себя в выборе метрик для извлечения и проявить щедрость при подборе значения для параметра `scrape_interval`.

Существуют также экспортёры, которые можно использовать для извлечения метрик из различных систем мониторинга SaaS (Software as a Service – программное обеспечение как услуга), в том числе CloudWatch Exporter (<https://oreil.ly/IQ9Fe>), New Relic Exporter (<https://oreil.ly/YWPcw>), Pingdom Exporter (<https://oreil.ly/UU4br>) и Stackdriver Exporter (<https://oreil.ly/JH2T9>). Обратите внимание, что эти экспортёры могут иметь ограниченную скорость работы и требовать финансовых затрат на использование API, к которым они обращаются.

¹ В версию Graphite 1.1.0 добавлена поддержка тегов, которые извлекаются экспортёром Graphite Exporter в виде меток.

² Management Information Base (база данных управляющей информации), фактически схема для объектов SNMP.

NRPE Exporter (<https://oreil.ly/BBkqg>) – это экспортер в стиле SNMP/Black-box, позволяющий выполнять проверки NRPE. Аббревиатура NRPE расшифровывается как Nagios Remote Program Execution и определяет способ запуска проверок Nagios на удаленных машинах. Хотя многие существующие проверки в стиле Nagios можно заменить метриками из Node Exporter и других экспортёров, у вас могут иметься некоторые нестандартные проверки, перенести которые будет довольно сложно. Экспортёр NRPE Exporter предлагает для таких случаев вариант перехода, позволяющий позже преобразовать эти проверки в другое решение, такое как сборщик текстовых файлов, как описано в разделе «Сборщик текстовых файлов» главы 7.

Интеграция с другими системами мониторинга не ограничивается запуском отдельных экспортёров; есть также возможность интеграции с популярными системами инструментирования, такими как Dropwizard¹. Клиент Java (<https://oreil.ly/lIdH0>), например, может извлекать метрики из Dropwizard, используя механизм отчетов, которые затем будут отображаться вместе с любыми напрямую инструментированными метриками в `/metrics`.



Dropwizard также может экспортirовать свои метрики через JMX. Однако если есть такая возможность (т. е., если вы контролируете исходный код приложения), то старайтесь отдавать предпочтение интеграции с Dropwizard через клиент Java вместо JMX, потому что использование JMX сопряжено с большими накладными расходами и требует дополнительной настройки.

InfluxDB

Экспортёр InfluxDB Exporter принимает линейный протокол InfluxDB, добавленный в версию InfluxDB 0.9.0. Этот протокол работает поверх HTTP, поэтому один и тот же TCP-порт может использоваться как для приема записей, так и для обслуживания конечной точки `/metrics`. Чтобы запустить InfluxDB Exporter, выполните действия, описанные в примере 11.1.

Пример 11.1 ♦ Загрузка и запуск InfluxDB Exporter

```
hostname $ wget https://github.com/prometheus/influxdb_exporter/releases/download/
    v0.10.0/influxdb_exporter-0.10.0.linux-amd64.tar.gz
hostname $ tar -xzf influxdb_exporter-0.10.0.linux-amd64.tar.gz
hostname $ cd influxdb_exporter-0.10.0.linux-amd64/
hostname $ ./influxdb_exporter
msg="Starting influxdb_exporter" version="(version=0.10.0, branch=
    HEAD, revision=6ce7ff5e3f584eb9c2019be71ecb9e586ba3d83e)"
msg="Build context" context="(go=go1.18.3, user=root@de8ee7c667
    c4, date=20220708-19:34:59)"
```

Затем можно потребовать от своих существующих приложений, поддерживающих линейный протокол InfluxDB, использовать InfluxDB Exporter. Вот как можно вручную отправить метрику с метками:

¹ Ранее известная как Yammer.

```
curl -XPOST 'http://localhost:9122/write' --data-binary \
'example_metric,foo=bar value=43 1517339868000000000'
```

Если теперь открыть в браузере страницу <http://localhost:9122/metrics>, то среди всего прочего вы увидите:

```
# HELP example_metric InfluxDB Metric
# TYPE example_metric untyped
example_metric{foo="bar"} 43
```

Можно заметить, что отметка времени, отправленная экспортеру, не отображается. Для экспортации отметок времени через `/metrics` не так много допустимых вариантов, потому что процедура опроса предусматривает только синхронный сбор метрик, описывающих текущее состояние приложения. Это относится не ко всем системам мониторинга, и в них использование отметок времени было бы допустимо. На момент написания этих строк только клиентская библиотека Java поддерживала передачу отметок времени пользовательским сборщикам. Когда метрики экспортируются без отметок времени, Prometheus использует время выполнения опроса. Это распространенные проблемы, с которыми приходится сталкиваться при переходе от модели принудительной передачи со стороны клиента к модели получения метрик по опросу. С другой стороны, переход от опроса к принудительной передаче довольно прост, как показано в примере 4.13.

Экспортер InfluxDB Exporter можно опрашивать, как любые другие экспортёры (пример 11.2).

Пример 11.2 ❖ `prometheus.yml` с конфигурацией опроса локального экспортёра InfluxDB Exporter

```
global:
  scrape_interval: 10s
scrape_configs:
- job_name: application_name
  static_configs:
    - targets:
      - localhost:9122
```

StatsD

StatsD принимает события и агрегирует их в метрики. Отправку события в StatsD можно представить как вызов `inc` для счетчика или `observe` для сводной метрики. Именно так и работает StatsD Exporter. Он преобразует события StatsD в метрики клиентской библиотеки Prometheus и инструментированные вызовы.

Чтобы запустить StatsD Exporter, выполните действия, описанные в примере 11.3.

Пример 11.3 ✧ Загрузка и запуск StatsD Exporter

```
hostname $ wget https://github.com/prometheus/statsd_exporter/releases/download/
v0.22.8/statsd_exporter-0.22.8.linux-amd64.tar.gz
hostname $ tar -xzf statsd_exporter-0.22.8.linux-amd64.tar.gz
hostname $ cd statsd_exporter-0.22.8.linux-amd64/
hostname $ ./statsd_exporter
msg="Starting StatsD -> Prometheus Exporter" version="(version=0.22.8, branch=
HEAD, revision=aecad1a2faf31d4a6c27323a29ca8c7a23d88f6b)"
msg="Build context" context="(go=go1.18.6, user=root@56d5d8c6d
3d1, date=20220913-14:49:05)"
msg="Accepting StatsD Traffic" udp=:9125 tcp=:9125 unixgram=
msg="Accepting Prometheus Requests" addr=:9102
```

Поскольку StatsD использует протоколы TCP и UDP, для отправки событий вам потребуются другие порты, отличные от тех, что используются для опроса `/metrics`.

Послать метрику-датчик можно вручную¹:

```
echo 'example_gauge:123|g' | nc localhost 9125
```

которая появится на странице `http://localhost:9102/metrics` как

```
# HELP example_gauge Metric autogenerated by statsd_exporter.
# TYPE example_gauge gauge
example_gauge 123
```

Аналогично можно отправлять приращения счетчика и наблюдения сводных метрик и гистограмм:

```
echo 'example_counter_total:1|c' | nc localhost 9125
echo 'example_latency_total:20|ms' | nc localhost 9125
```

Протокол StatsD определен не полностью; многие реализации поддерживают только целочисленные значения. И хотя StatsD Exporter не имеет этого ограничения, многие метрики нельзя экспортить в базовых единицах, привычных для Prometheus.

Также есть возможность извлекать метки, потому что StatsD часто используется с нотацией *точечных строк* Graphite, согласно которой позиция определяет значение. Например, `app.http.requests.eu-west-1./foo` может быть преобразована в `app_http_requests_total{region="euwest-1",path="/foo"}` в Prometheus. Чтобы иметь возможность преобразовывать такие строки, необходимо предоставить файл `mapping.yml`, описывающий преобразования, например:

```
mappings:
- match: app.http.requests.*.*
  name: app_http_requests_total
```

¹ `nc` – это сетевая утилита, полное имя которой – *netcat*. Возможно, вам придется установить ее, если она отсутствует в вашей системе.

```
labels:  
  region: "${1}"  
  path:  "${2}"
```

А затем передать его StatsD Exporter при запуске:

```
./statsd_exporter -statsd.mapping-config mapping.yml
```

Если теперь отправить запрос в StatsD Exporter по шаблону, показанному выше, то он вернет соответствующие имена и метки:

```
echo 'app.http.requests.eu-west-1./foo:1|c' | nc localhost 9125  
echo 'app.http.requests.eu-west-1./bar:1|c' | nc localhost 9125
```

В этом можно убедиться, открыв страницу <http://localhost:9102/metrics>:

```
# HELP app_http_requests_total Metric autogenerated by statsd_exporter.  
# TYPE app_http_requests_total counter  
app_http_requests_total{path="/bar",region="eu-west-1"} 1  
app_http_requests_total{path="/foo",region="eu-west-1"} 1
```

Graphite Exporter имеет аналогичный механизм преобразования строк с точками в метки.

Наконец, StatsD Exporter можно использовать даже после завершения перехода на Prometheus для мониторинга веб-приложений, реализованных на таких языках, как PHP и Perl. Как упоминалось в разделе «Многозадачность с Gunicorn» главы 4, Prometheus поддерживает многопоточную модель с долгоживущими процессами. Но такие языки, как PHP, часто используются для реализации процессов, запускаемых только для обработки единственного HTTP-запроса. Хотя к приложениям на PHP можно применить подход, подобный тому, который используется клиентом Python для многозадачных развертываний, вы можете обнаружить, что применение StatsD Exporter более практично. В этом пространстве также есть `prom-aggregation-gateway` (<https://oreil.ly/qrgYk>).

Мы бы рекомендовали при использовании таких экспортёров, как InfluxDB Exporter, Graphite Exporter, StatsD Exporter и Collectd Exporter, преобразующих модель принудительной передачи в модель опроса (push-to-pull), запускать по одному экспортёру для каждого экземпляра приложения с жизненным циклом, совпадающим с жизненным циклом приложения. Желательно запускать, останавливать и перезапускать экспортёр одновременно с запуском, остановкой и перезапуском экземпляра приложения. Такой способ упрощает управление, позволяет избежать проблем со сменой меток и не позволяет экспортёру стать узким местом¹.

Несмотря на то что в настоящее время доступны сотни самых разных экспортёров, вам все же может понадобиться написать свой или расширить существующий. В следующей главе мы покажем, как писать экспортёры.

¹ Одна из причин появления Prometheus связана с проблемами масштабирования, имевшимися у SoundCloud, когда многие приложения отправляли данные в одну систему StatsD.

Глава 12

Разработка экспортеров

Иногда невозможно ни инструментировать приложение напрямую, ни подобрать подходящий существующий экспортер. В таких случаях не остается ничего другого, как написать свой экспортер. Однако сделать это совсем нетрудно. Самое сложное – выяснить, что означают метрики, экспортруемые приложениями. Единицы измерения часто неизвестны, а документация может содержать лишь расплывчатые описания. В этой главе вы узнаете, как написать свой экспортер.

Consul Telemetry

Для демонстрации процесса разработки мы напишем небольшой экспортер для Consul. Мы уже познакомились с Consul и Consul Exporter в разделе «Consul» главы 10, поэтому создадим простой экспортер, поставляющий метрики из API телеметрии¹.

Экспортеры можно писать на любом языке программирования, но большинство из них написано на Go, и именно этот язык мы используем. Однако вы можете найти небольшое количество экспортеров, написанных на Python, и еще меньшее – на Java.

Если агент Consul еще не запущен, то запустите его снова, следуя инструкциям в примере 8.8. Открыв страницу <http://localhost:8500/v1/agent/metrics>, вы увидите выходные данные в формате JSON, с которыми мы будем работать (пример 12.1). В комплекте с Consul поставляется библиотека на Go, которую мы сможем использовать для создания своего экспортера, поэтому нам не придется беспокоиться о реализации парсинга JSON.

Пример 12.1 ♦ Пример вывода метрик агента Consul

```
{  
  "Timestamp": "2018-01-31 14:42:10 +0000 UTC",
```

¹ Эти метрики также экспортруются Consul, но мы представим, что пишем этот экспортер в то время, когда эти метрики еще не экспортровались механизмом обнаружения служб Consul.

```

"Gauges": [
  {
    "Name": "consul.autopilot.failure_tolerance",
    "Value": 0,
    "Labels": {}
  }
],
"Points": [],
"Counters": [
  {
    "Name": "consul.raft.apply",
    "Count": 1,
    "Sum": 2, "Min": 1, "Max": 1, "Mean": 1, "Stddev": 0,
    "Labels": {}
  }
],
"Samples": [
  {
    "Name": "consul.fsm.coordinate.batch-update",
    "Count": 1,
    "Sum": 0.13156799972057343,
    "Min": 0.13156799972057343, "Max": 0.13156799972057343,
    "Mean": 0.13156799972057343, "Stddev": 0,
    "Labels": {}
  }
]
}

```

Нам очень повезло, что Consul автоматически разделил счетчики и датчики¹. Метрики в разделе `Samples` тоже выглядят так, будто можно использовать `Count` и `Sum` в сводной метрике. При взгляде на метрики `Samples` возникает подозрение, что они отслеживают задержку. Изучение документации подтверждает, что это *таймеры* – сводные метрики в Prometheus (см. раздел «Сводные метрики» главы 3). Значения всех таймеров выражены в миллисекундах, поэтому мы можем преобразовать их в секунды². Несмотря на то что в данных JSON есть поле `Labels`, ни одно из них не используется, поэтому мы можем игнорировать его. Также нам нужно вычистить все недопустимые символы из имен метрик.

Теперь, зная логику, которую необходимо применить к метрикам, экспортируемым агентом Consul, можно написать свой экспортёр, как показано в примере 12.2.

¹ Название «счетчик» (counter) еще не означает, что перед нами действительно счетчик. Например, в Dropwizard есть счетчики, которые могут считать в обратную сторону, поэтому в зависимости от фактического использования в терминах Prometheus метрика может быть счетчиком, датчиком или нетипизированной метрикой.

² Если бы таймерами были только некоторые из метрик `Samples`, то вам пришлось бы выбирать между передачей их «как есть» или описывать, какие метрики являются задержками, а какие нет.

Пример 12.2 ❖ consul_metrics.go, экспортер метрик Consul, написанный на Go

```

package main

import (
    "log"
    "net/http"
    "regexp"

    "github.com/hashicorp/consul/api"
    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

var (
    up = prometheus.NewDesc(
        "consul_up",
        "Was talking to Consul successful.",
        nil, nil,
    )
    invalidChars = regexp.MustCompile("[^a-zA-Z0-9:_]")
)

type ConsulCollector struct {
}

// Реализация prometheus.Collector.
func (c ConsulCollector) Describe(ch chan<- *prometheus.Desc) {
    ch <- up
}

// Реализация prometheus.Collector.
func (c ConsulCollector) Collect(ch chan<- prometheus.Metric) {
    consul, err := api.NewClient(api.DefaultConfig())
    if err != nil {
        ch <- prometheus.MustNewConstMetric(up, prometheus.GaugeValue, 0)
        return
    }

    metrics, err := consul.Agent().Metrics()
    if err != nil {
        ch <- prometheus.MustNewConstMetric(up, prometheus.GaugeValue, 0)
        return
    }
    ch <- prometheus.MustNewConstMetric(up, prometheus.GaugeValue, 1)

    for _, g := range metrics.Gauges {
        name := invalidChars.ReplaceAllLiteralString(g.Name, "_")
        desc := prometheus.NewDesc(name, "Consul metric "+g.Name, nil,
            g.Labels)
        ch <- prometheus.MustNewConstMetric(
            desc, prometheus.GaugeValue, float64(g.Value))
    }

    for _, c := range metrics.Counters {
}

```

```

        name := invalidChars.ReplaceAllLiteralString(c.Name, "_")
        desc := prometheus.NewDesc(name+"_total", "Consul metric "+c.Name,
                                    nil, c.Labels)
        ch <- prometheus.MustNewConstMetric(
            desc, prometheus.CounterValue, float64(c.Count))
    }

    for _, s := range metrics.Samples {
        // Все метрики в разделе Samples - это таймеры, измеряющие
        // время в миллисекундах, мы преобразуем их в секунды.
        name := invalidChars.ReplaceAllLiteralString(s.Name, "_") +
            "_seconds"
        countDesc := prometheus.NewDesc(
            name+"_count", "Consul metric "+s.Name, nil, s.Labels)
        ch <- prometheus.MustNewConstMetric(
            countDesc, prometheus.CounterValue, float64(s.Count))
        sumDesc := prometheus.NewDesc(
            name+"_sum", "Consul metric "+s.Name, nil, s.Labels)
        ch <- prometheus.MustNewConstMetric(
            sumDesc, prometheus.CounterValue, s.Sum/1000)
    }
}

func main() {
    c := ConsulCollector{}
    prometheus.MustRegister(c)
    http.Handle("/metrics", promhttp.Handler())
    log.Fatal(http.ListenAndServe(":8000", nil))
}

```

При наличии среды разработки на Go можно запустить экспортатор:

```

go get -d -u github.com/hashicorp/consul/api
go get -d -u github.com/prometheus/client_golang/prometheus
go run consul_metrics.go

```

Если теперь открыть в браузере страницу <http://localhost:8000/metrics>, то можно увидеть следующие метрики:

```

# HELP consul_autopilot_failure_tolerance Consul metric
  consul.autopilot.failure_tolerance
# TYPE consul_autopilot_failure_tolerance gauge
consul_autopilot_failure_tolerance 0
# HELP consul_raft_apply_total Consul metric consul.raft.apply
# TYPE consul_raft_apply_total counter
consul_raft_apply_total 1
# HELP consul_fsm_coordinate_batch_update_seconds_count Consul metric
  consul.fsm.coordinate.batch-update
# TYPE consul_fsm_coordinate_batch_update_seconds_count counter
consul_fsm_coordinate_batch_update_seconds_count 1
# HELP consul_fsm_coordinate_batch_update_seconds_sum Consul metric
  consul.fsm.coordinate.batch-update
# TYPE consul_fsm_coordinate_batch_update_seconds_sum counter
consul_fsm_coordinate_batch_update_seconds_sum 1.3156799972057343e-01

```

Это все хорошо, но как заставить этот код работать? Мы расскажем об этом в следующем разделе.

Пользовательские сборщики

При прямом инструментировании клиентская библиотека принимает события и отслеживает значения метрик с течением времени. Клиентские библиотеки предоставляют для этого такие метрики, как счетчики, датчики, сводные метрики и гистограммы, являющиеся примерами *сборщиков*. Процедура опроса *выбирает* данные из каждого зарегистрированного сборщика, т. е. запрашивает у него его метрики. Затем эти метрики возвращаются через конечную точку */metrics*. Счетчики и метрики трех других стандартных типов всегда возвращают только одно семейство метрик.

Если вместо прямого инструментирования вы решили экспортить данные из некоторого другого источника, то вам потребуется *пользовательский сборщик* – любой сборщик, не входящий в группу из четырех стандартных сборщиков. Пользовательские сборщики могут возвращать любое количество семейств метрик. Извлечение метрик происходит при каждом опросе конечной точки */metrics*, и каждый опрос дает в результате непротиворечивый снимок метрик из сборщика.

В Go сборщики должны реализовать интерфейс `prometheus.Collector`. То есть сборщики должны быть объектами с методами `Describe` и `Collect` с определенной сигнатурой.

Метод `Describe` возвращает описание создаваемых метрик, в частности имя метрики, имена меток и строку справки. Метод `Describe` вызывается во время регистрации и используется, чтобы исключить дублирование регистрируемых метрик.

Экспортер может иметь два типа метрик: метрики, имена которых ему известны заранее, и метрики, имена и метки которых определяются только во время опроса. В этом примере метрика `consul_up` известна заранее, поэтому можно создать ее описание `Desc` один раз с помощью `NewDesc` и экспортить через `Describe`. Все остальные метрики генерируются динамически во время опроса, поэтому их нельзя включить:

```
var (
    up = prometheus.NewDesc(
        "consul_up",
        "Was talking to Consul successful.",
        nil, nil,
    )
)
// Реализует prometheus.Collector.
func (c ConsulCollector) Describe(ch chan<- *prometheus.Desc) {
    ch <- up
}
```

-  Клиент Go требует, чтобы `Describe` предоставлял хотя бы одно описание `Desc`. Если все ваши метрики создаются динамически, то можно указать фиктивное описание, чтобы удовлетворить это требование.

В основе пользовательского сборщика лежит метод `Collect`. Он извлекает все необходимые данные из экземпляра приложения, обрабатывает их при необходимости, а затем возвращает метрики клиентской библиотеке. Здесь нужно подключиться к агенту Consul и получить его метрики. В случае ошибки в `consul_up` возвращается значение 0; в противном случае если извлечение данных завершилось успехом, то в ней будет возвращено значение 1. В PromQL иногда трудно работать только с возвращаемыми метриками¹. Наличие метрики `consul_up` позволяет предупреждать о проблемах, возникших при попытке взаимодействия с Consul.

Чтобы вернуть `consul_up`, здесь используется `prometheus.MustNewConstMetric`. Этот метод принимает описание `Desc`, тип и значение метрики:

```
// Реализует prometheus.Collector.
func (c ConsulCollector) Collect(ch chan<- prometheus.Metric) {
    consul, err := api.NewClient(api.DefaultConfig())
    if err != nil {
        ch <- prometheus.MustNewConstMetric(up, prometheus.GaugeValue, 0)
        return
    }

    metrics, err := consul.Agent().Metrics()
    if err != nil {
        ch <- prometheus.MustNewConstMetric(up, prometheus.GaugeValue, 0)
        return
    }
    ch <- prometheus.MustNewConstMetric(up, prometheus.GaugeValue, 1)
}
```

Возможны три значения: `GaugeValue`, `CounterValue` и `UntypedValue`. Значения `GaugeValue` и `CounterValue` вы уже знаете, а `UntypedValue` предназначено для случаев, когда нет уверенности, является ли метрика счетчиком или датчиком. Такое невозможно при прямом инструментировании, но нередко типы метрик из других систем мониторинга и инструментирования бывает очень трудно идентифицировать однозначно.

Теперь, получив метрики из Consul, можно обработать датчики. Недопустимые символы в именах метрик, такие как точки и дефисы, преобразуются в символы подчеркивания. `Desc` создается динамически и сразу же используется в `MustNewConstMetric`:

```
for _, g := range metrics.Gauges {
    name := invalidChars.ReplaceAllLiteralString(g.Name, "_")
    desc := prometheus.NewDesc(name, "Consul metric "+g.Name, nil, g.Labels)
    ch <- prometheus.MustNewConstMetric(
        desc, prometheus.GaugeValue, float64(g.Value))
}
}
```

¹ См. раздел «Оператор or» главы 15.

i В последнем параметре в вызов `prometheus.NewDesc` мы передаем `g.Labels`. Это набор меток, назначенных агентом Consul, таких как `datacenter`. Мы должны передать их, потому что кардинальность некоторых датчиков больше единицы, и без этих меток страница `/metrics` выдаст ошибку.

Счетчики обрабатываются аналогично, за исключением того, что к имени метрики добавляется суффикс `_total`:

```
for _, c := range metrics.Counters {
    name := invalidChars.ReplaceAllLiteralString(c.Name, "_")
    desc := prometheus.NewDesc(name+"_total", "Consul metric "+c.Name, nil,
        c.Labels)
    ch <- prometheus.MustNewConstMetric(
        desc, prometheus.CounterValue, float64(s.Count))
}
```

`metrics.Samples` имеет более сложное содержимое. Это обычные сводные метрики Prometheus, однако в настоящее время клиент Go не поддерживает метрики для `MustNewConstMetric`. Чтобы обойти эту проблему, их можно эмулировать с помощью двух счетчиков. К имени метрики добавляется суффикс `_seconds`, а сумма делится на одну тысячу, чтобы преобразовать миллисекунды в секунды:

```
for _, s := range metrics.Samples {
    // Все метрики содержат время в миллисекундах, мы преобразуем их в секунды.
    name := invalidChars.ReplaceAllLiteralString(s.Name, "_") + "_seconds"
    countDesc := prometheus.NewDesc(
        name+"_count", "Consul metric "+s.Name, nil, s.Labels)
    ch <- prometheus.MustNewConstMetric(
        countDesc, prometheus.CounterValue, float64(s.Count))
    sumDesc := prometheus.NewDesc(
        name+"_sum", "Consul metric "+s.Name, nil, s.Labels)
    ch <- prometheus.MustNewConstMetric(
        sumDesc, prometheus.CounterValue, s.Sum/1000)
}
```

! Здесь `s.Sum` – это число типа `float64`, но будьте осторожны при делении на целое число, чтобы не потерять точность. Если бы метрика `Sum` была целым числом, то операнды `float64(sum)/1000` сначала были бы преобразованы в число с плавающей точкой, а затем было бы выполнено деление, что вам и нужно. С другой стороны, выражение `float64(sum/1000)` сначала разделит целочисленное значение на одну тысячу, теряя при этом три цифры после запятой.

Наконец, создается пользовательский объект-сборщик и регистрируется в реестре по умолчанию, подобно метрике, полученной методом прямого инструментирования:

```
c := ConsulCollector{}
prometheus.MustRegister(c)
```

Экспортирование выполняется обычным образом, как описывается в разделе «Go» главы 4:

```
http.Handle("/metrics", promhttp.Handler())
log.Fatal(http.ListenAndServe(":8000", nil))
```

Это, конечно, упрощенный пример. На самом деле у вас должен иметься некоторый способ настроить сервер Consul для взаимодействий, например флаг командной строки, чтобы не зависеть от настроек клиента по умолчанию. Вы также можете повторно использовать клиент между циклами опроса и разрешить указывать различные параметры аутентификации клиента.

 Метрики `min`, `max`, `mean` и `stddev` были исключены из вывода, поскольку они почти бесполезны. Среднее значение (`mean`), например, можно вычислить на основе суммы и количества. А метрики `min`, `max` и `stddev`, в свою очередь, не могут быть агрегированы, так как неизвестно, за какой период времени они были получены.

Благодаря использованию реестра по умолчанию в результат включаются метрики `go_` и `process_`. Они содержат информацию о производительности самого экспортёра и могут пригодиться для обнаружения таких проблем, как утечка файловых дескрипторов, с помощью `process_open_fds`. Это избавляет от необходимости создавать отдельные экспортёры для этих метрик.

Единственный случай, когда можно не использовать реестр по умолчанию для регистрации экспортёра, – когда экспортёр реализуется в стиле Blackbox/SNMP и требуется выполнить интерпретацию параметров URL, поскольку сборщики не имеют доступа к параметрам URL. В таких случаях вы также будете опрашивать конечную точку `/metrics` экспортёра для мониторинга его самого.

Для сравнения в примере 12.3 показан эквивалентный экспортёр, написанный на Python 3. Экспортёр на Python во многом повторяет реализацию на Go. Единственное заметное отличие – `SummaryMetricFamily` может представлять сводную метрику, избавляя от необходимости эмулировать ее с помощью двух отдельных счетчиков. В клиенте на Python не так много проверок работоспособности, как в клиенте на Go, поэтому с ним нужно быть немного осторожнее.

Пример 12.3 ❖ consul_metrics.py, экспортёр метрик Consul на Python 3.

```
import json
import re
import time
from urllib.request import urlopen
from prometheus_client.core import GaugeMetricFamily, CounterMetricFamily
from prometheus_client.core import SummaryMetricFamily, REGISTRY
from prometheus_client import start_http_server

def sanitize_name(s):
    return re.sub(r"[^a-zA-Z0-9:_]", "_", s)

class ConsulCollector(object):
    def collect(self):
        out = urlopen("http://localhost:8500/v1/agent/metrics").read()
        metrics = json.loads(out.decode("utf-8"))

        for g in metrics["Gauges"]:
            yield GaugeMetricFamily(sanitize_name(g["Name"]),

```

```

    "Consul metric " + g["Name"], g["Value"])

for c in metrics["Counters"]:
    yield CounterMetricFamily(sanitize_name(c["Name"]) + "_total",
        "Consul metric " + c["Name"], c["Count"])

for s in metrics["Samples"]:
    yield SummaryMetricFamily(sanitize_name(s["Name"]) + "_seconds",
        "Consul metric " + s["Name"],
        count_value=c["Count"], sum_value=s["Sum"] / 1000)

if __name__ == '__main__':
    REGISTRY.register(ConsulCollector())
    start_http_server(8000)
    while True:
        time.sleep(1)

```

Метки

В предыдущем примере вы видели только метрики без меток. Чтобы добавить метки, нужно указать имена меток в `Desc` и передать соответствующие им значения в `MustNewConstMetric`.

Добавить метрику с временными рядами `example_gauge{foo="bar", baz="small"}` и `example_gauge{foo="quu", baz="far"}` можно с помощью клиентской библиотеки Prometheus для Go:

```

func (c MyCollector) Collect(ch chan<- prometheus.Metric) {
    desc := prometheus.NewDesc(
        "example_gauge",
        "A help string.",
        []string{"foo", "baz"}, nil,
    )
    ch <- prometheus.MustNewConstMetric(
        desc, prometheus.GaugeValue, 1, "bar", "small")
    ch <- prometheus.MustNewConstMetric(
        desc, prometheus.GaugeValue, 2, "quu", "far")
}

```

Каждый временной ряд можно экспортить отдельно. Реестр позаботится об объединении всех временных рядов, принадлежащих одному семейству метрик, в выходных данных `/metrics`.

 Строки с описанием метрик с одинаковыми именами должны совпадать. Если строки будут отличаться, то это приведет к сбою во время опроса.

Клиент на Python работает немного иначе; он должен сконструировать семейство метрик, а затем вернуть его. Может показаться, что такой подход требует много лишних усилий, но в действительности трудозатраты получаются примерно такими же, как и при использовании языка Go:

```

class MyCollector(object):
    def collect(self):

```

```
mf = GaugeMetricFamily("example_gauge", "A help string.",
    labels=["foo", "baz"])
mf.add_metric(["bar", "small"], 1)
mf.add_metric(["quu", "far"], 2)
yield mf
```

Методические рекомендации

В отличие от прямого инструментирования разработка экспортёров, как правило, сложнее, что связано с необходимостью идти на технические компромиссы. У вас есть возможность потратить массу времени на получение идеальных метрик или на создание чего-то хорошего и не требующего обслуживания? Тогда займитесь разработкой экспортёров, но имейте в виду, что написание экспортёров – это больше искусство, чем наука.

Страйтесь строго следовать правилам именования метрик, в частности избегайте суффиксов `_count`, `_sum`, `_total`, `_bucket` и `_info`, если только временные ряды не являются частью метрики, которая должна содержать их.

Часто невозможно или почти невозможно определить, к какой категории относятся метрики – датчикам, счетчикам или их комбинациям. Если метрики являются комбинациями датчиков и счетчиков, отмечайте их как *нетипизированные* (*untyped*). Если метрика является счетчиком, не забудьте добавить суффикс `_total`.

Там, где это целесообразно, указывайте единицы измерения и не забудьте указать единицы измерения в именах метрик. Необходимость определять единицы измерения по метрикам, как было показано в примере 12.1, – сомнительное удовольствие, поэтому страйтесь снять это бремя с пользователей вашего экспортёра. Секунды и байты всегда предпочтительнее.

Использование меток в экспортёрах имеет свои подводные камни, на которые следует обратить внимание. Экспортёры, как и метрики, полученные прямым инструментированием, страдают проблемой кардинальности, обсуждавшейся в разделе «Кардинальность» главы 5. Страйтесь избегать метрик с высокой кардинальностью в их метках.

Метки метрики должны быть осмысленными. Если вы берете сумму или среднее значение по метрике, то они должны иметь осмысленное значение, как описано в разделе «Когда использовать метки» главы 5. В частности, обращайте внимание на любые временные ряды, которые являются простыми суммами всех других значений в метрике, и удалите их. Если, разрабатывая экспортёра, вы не уверены, имеет ли смысл использовать некоторую метку, то безопаснее не использовать ее, но при этом не забывайте, о чём говорилось во врезке «Исключение таблицы» в главе 5. Как и при прямом инструментировании, не используйте такие метки, как `env="prod"`, для всех метрик, поступающих из вашего экспортёра, потому что это область ответственности целевых меток, как описано в разделе «Целевые метки» главы 8.

Лучше передать необработанные метрики в Prometheus, чем выполнить вычисления на стороне приложения. Например, нет необходимости экспортировать частоту за 5-минутный интервал, если есть счетчик, потому что

в этом случае можно использовать функцию `rate` для вычисления частоты за любой период. То же верно для отношений: откажитесь от них в пользу отдельных метрик, представляющих числитель и знаменатель. Если у вас есть величина в процентах без составляющих ее числителя и знаменателя, то хотя бы преобразуйте ее в отношение¹.

Помимо умножения и деления для стандартизации единиц измерения, избегайте математических вычислений в экспортерах, потому что необработанные данные лучше обрабатывать в PromQL. Условия гонки между событиями инструментирования метрик могут привести к искажениям, особенно когда метрики вычисляются на основе друг друга. Добавление метрик для уменьшения кардинальности вполне допустимо, но если это счетчики, то убедитесь, что не появятся искаженные значения из-за исчезновения некоторых из них.

Некоторые метрики почти бесполезны, учитывая особенности Prometheus. Многие приложения экспортируют такие метрики, как объем оперативной памяти компьютера, количество процессоров и емкость диска. Не следует предусматривать передачу подобных метрик машинного уровня в своем экспортере, так как это обязанность Node Exporter². Агрегировать минимумы, максимумы и стандартные отклонения не имеет смысла, поэтому от них тоже следует отказаться.

Для каждого экземпляра приложения запускайте отдельный экземпляр экспортара³ и извлекайте метрики синхронно без какого-либо кеширования. Это сохранит обязанности по обнаружению служб и планированию опроса за Prometheus. Также не забывайте, что одновременно может выполняться несколько циклов опроса⁴.

Подобно тому, как Prometheus добавляет метрику `scrape_duration_seconds` при выполнении опроса, вы можете добавить метрику `muexporter_scrape_duration_seconds`, сообщающую, сколько времени требуется экспортёру для извлечения данных из своего приложения. Она поможет при отладке производительности, так как позволит увидеть, что замедляет работу приложения или вашего экспортёра. Также могут пригодиться дополнительные метрики, такие как счетчик обработанных метрик.

Иногда, в дополнение к пользовательским сборщикам, может быть полезно добавить прямое инструментирование в экспортёры, чтобы обеспечить возможность мониторинга их основных функций. Например, CloudWatch Exporter имеет счетчик `cloudwatch_requests_total`, сообщающий количество вызовов API, поскольку каждый вызов API стоит денег. Но обычно прямое инструментирование полезно только для экспортёров в стиле Blackbox/SNMP.

Теперь, узнав, как получать метрики из ваших приложений или из стороннего кода, в следующей главе мы начнем знакомиться с использованием PromQL для работы с метриками.

¹ И обязательно убедитесь, что это действительно отношение/процент. Нередко метрики путают их.

² Или Windows Exporter в ОС Windows.

³ Это не относится к экспортёрам в стиле Blackbox/SNMP.

⁴ Такое возможно, когда экспортёры опрашиваются несколькими серверами.

Часть IV

PromQL

Язык запросов Prometheus (Prometheus Query Language, PromQL) позволяет выполнять все виды агрегирования, анализа и арифметические операции, чтобы лучше понять работу контролируемых систем по полученным метрикам.

В этой части мы вернемся к настроенному окружению Prometheus и Node Exporter, созданному в главе 2, и будем использовать браузер выражений для выполнения запросов.

Глава 13 посвящена основам PromQL и описывает, как можно использовать HTTP API для вычисления выражений.

В главе 14 подробно рассматриваются приемы агрегирования.

В главе 15 мы поговорим о таких операторах, как сложение и сравнение, а также о способах соединения различных метрик.

В главе 16 рассматривается широкий спектр функций, предлагаемых PromQL, от определения времени суток до предсказания момента, когда наступит переполнение жесткого диска.

Глава 17 охватывает правила записи, которые позволяют заранее вычислять метрики для ускорения сложных запросов PromQL.

Глава 13

Введение в PromQL

PromQL – это язык запросов Prometheus (Prometheus Query Language). Хотя его название заканчивается на *QL*, вы увидите, что он совсем не похож на язык SQL, которому, как правило, не хватает выразительной силы для реализации вычислений с временными рядами.

Метки играют важную роль в PromQL. Их можно использовать не только для произвольного агрегирования, но также для объединения различных метрик и выполнения арифметических операций. Вам доступен широкий спектр функций, от предсказаний до операций с датами и математических вычислений.

В этой главе вы познакомитесь с основными концепциями PromQL, включая агрегирование, основные типы данных и HTTP API.

Основы агрегирования

Начнем с простых запросов агрегирования. Представленные здесь запросы охватывают большинство типичных применений PromQL. Хотя язык PromQL обладает невероятной мощностью¹, в большинстве случаев вы будете применять его для выполнения простых запросов.

Датчики

Датчики – это моментальные снимки состояния, и обычно под их агрегированием понимается получение суммы, среднего значения, минимума или максимума.

Рассмотрим метрику `node_filesystem_size_bytes` из Node Exporter, которая сообщает размер каждой из смонтированных файловых систем и имеет мет-

¹ Брайан двумя разными способами доказал (<https://oreil.ly/TQWlz>), что PromQL является полным по Тьюрингу. Но, пожалуйста, не пытайтесь воспроизвести его доказательства в промышленном окружении.

ки `device`, `fstype` и `mountpoint`. Вот как можно узнать общий размер файловых систем на каждой машине:

```
sum without(device, fstype, mountpoint)(node_filesystem_size_bytes)
```

Здесь `without` сообщает функции агрегирования `sum`, что она должна суммировать все с одинаковыми метками, игнорируя эти три. То есть если у вас есть временные ряды

```
node_filesystem_free_bytes{device="/dev/sda1",fstype="vfat",
  instance="localhost:9100",job="node",mountpoint="/boot/efi"} 70300672
node_filesystem_free_bytes{device="/dev/sda5",fstype="ext4",
  instance="localhost:9100",job="node",mountpoint="/" } 30791843840
node_filesystem_free_bytes{device="tmpfs",fstype="tmpfs",
  instance="localhost:9100",job="node",mountpoint="/run"} 817094656
node_filesystem_free_bytes{device="tmpfs",fstype="tmpfs",
  instance="localhost:9100",job="node",mountpoint="/run/lock"} 5238784
node_filesystem_free_bytes{device="tmpfs",fstype="tmpfs",
  instance="localhost:9100",job="node",mountpoint="/run/user/1000"} 826912768
```

то результат должен выглядеть так:

```
{instance="localhost:9100",job="node"} 32511390720
```

Обратите внимание, что метки `device`, `fstype` и `mountpoint` исчезли. Имя метрики тоже отсутствует, так как это уже не метрика `node_filesystem_free_bytes`, потому что с ней были выполнены математические операции. Поскольку Prometheus опрашивает только одного экспортёра Node Exporter, результат будет только один, но если бы мы опрашивали несколько экспортёров, то и результатов бы получили несколько, по одному для каждого экспортёра Node Exporter.

Можно сделать еще один шаг и удалить метку `instance`:

```
sum without(device, fstype, mountpoint, instance)(node_filesystem_size_bytes)
```

Этот запрос, как и следовало ожидать, удаляет метку `instance`, но значение результата от этого не изменяется, потому что в агрегировании метрик участвует только один Node Exporter:

```
{job="node"} 32511390720
```

Аналогичный подход можно использовать с другими функциями агрегирования. Функция `max`, например, сообщит вам размер самой большой смонтированной файловой системы на каждой машине:

```
max without(device, fstype, mountpoint)(node_filesystem_size_bytes)
```

Этот запрос выведет те же метки, что и `sum`:

```
{instance="localhost:9100",job="node"} 30792601600
```

Эта предсказуемость возвращаемых меток важна для сопоставления векторов с операторами, о которых рассказывается в главе 15.

Обратите внимание, что агрегировать метрики можно не только по типу задания. Например, вот как можно найти среднее количество файловых дескрипторов, открытых для всех заданий:

```
avg without(instance, job)(process_open_fds)
```

Счетчики

Счетчики отслеживают количество или размеры событий, а значения, экспортируемые вашими приложениями в `/metrics`, являются общими, накопленными с момента запуска. Но значения счетчиков сами по себе мало полезны. Чаще бывает желательно знать, как быстро счетчик увеличивается с течением времени. Обычно для этого используется функция `rate`, но к счетчикам можно также применять функции `increase` и `irate`.

Например, вот как можно рассчитать объем сетевого трафика, получаемого в секунду:

```
rate(node_network_receive_bytes_total[5m])
```

[`5m`] говорит, что `rate` должна использовать данные за последние 5 мин, поэтому возвращаемое значение будет средним за последние 5 мин:

```
{device="lo",instance="localhost:9100",job="node"} 1859.389655172414
{device="wlan0",instance="localhost:9100",job="node"} 1314.5034482758622
```

Значения получаются дробными, потому что 5-минутное окно, которое просматривает `rate`, не точно совпадает с циклами извлечения метрик. Для заполнения пробелов между имеющимися точками данных и границами диапазона используется некоторая оценка.

Результатом `rate` является датчик, поэтому к нему применимы те же агрегирующие функции, что и к обычным датчикам. Метрика `node_network_receive_bytes_total` имеет метку `device`, поэтому если агрегировать по ней, то можно получить общее количество байтов, получаемых машиной в секунду:

```
sum without(device)(rate(node_network_receive_bytes_total[5m]))
```

Вот какой результат возвращает этот запрос:

```
{instance="localhost:9100",job="node"} 3173.8931034482762
```

Есть возможность отфильтровать анализируемые временные ряды, чтобы, например, получить данные только по интерфейсу `eth0`, а затем агрегировать его по всем машинам по метке `instance`:

```
sum without(instance)(rate(node_network_receive_bytes_total{device="eth0"}[5m]))
```

Этот запрос удаляет из результата метку `instance`, но метка `device` остается, потому что мы не просили ее удалить:

```
{device="eth0",job="node"} 3173.8931034482762
```

Метки никак не упорядочиваются и не имеют какой-то определенной организации, что позволяет выполнить агрегирование любому количеству меток.

Сводные метрики

Сводные метрики обычно содержат временные ряды `_sum` и `_count`, а иногда и без суффикса с меткой `quantile`. Ряды `_sum` и `_count` являются счетчиками.

Prometheus предоставляет сводную метрику `http_response_size_bytes` с объемом данных, возвращаемых некоторыми его HTTP API¹. `http_response_size_bytes_count` подсчитывает количество запросов, и, поскольку это счетчик, перед агрегированием по метке `handler` необходимо использовать `rate`:

```
sum without(handler)(rate(http_response_size_bytes_count[5m]))
```

Этот запрос вернет количество HTTP-запросов в секунду, и, поскольку Node Exporter тоже возвращает эту метрику, вы увидите в результате оба задания:

```
{instance="localhost:9090",job="prometheus"} 0.26868836781609196
{instance="localhost:9100",job="node"} 0.1
```

Точно так же метрика `http_response_size_bytes_sum` – это счетчик количества байтов, возвращаемых каждым обработчиком, поэтому к ней применим тот же шаблон:

```
sum without(handler)(rate(http_response_size_bytes_sum[5m]))
```

Этот запрос вернет результат с теми же метками, что и предыдущий, но значения будут больше, потому что в ответах обычно возвращается больше информации:

```
{instance="localhost:9090",job="prometheus"} 796.0015958275862
{instance="localhost:9100",job="node"} 1581.6103448275862
```

Преимущество сводных метрик заключается в том, что они позволяют рассчитать средний размер события, в данном случае среднее количество байтов, возвращаемых в каждом ответе. Если бы у нас было три ответа размером 1, 4 и 7, то среднее значение было бы равно их сумме, деленной на их количество, т. е. 12, деленное на 3. То же относится к сводным метрикам. Чтобы получить среднее значение за период времени, нужно разделить `_sum` на `_count` (после вызова `rate`):

```
sum without(handler)(rate(http_response_size_bytes_sum[5m]))
/
sum without(handler)(rate(http_response_size_bytes_count[5m]))
```

¹ В Prometheus 2.3.0 она была переименована в `prometheus_http_response_size_bytes_count`.

Оператор деления выбирает временные ряды с одинаковыми метками и выполняет деление, возвращая те же два временных ряда, но со средним размером ответа за последние 5 мин в качестве значения:

```
{instance="localhost:9090",job="prometheus"} 2962.54580091246150133317
{instance="localhost:9100",job="node"} 15816.10344827586200000000
```

При вычислении среднего значения важно сначала выполнить агрегирование `_sum` и `_count` и только потом вычислять частное. В противном случае вы усредните средние значения, что с точки зрения статистики не имеет смысла.

Например, вот как можно получить средний размер ответа по всем экземплярам задания¹:

```
sum without(instance)(
    sum without(handler)(rate(http_response_size_bytes_sum[5m]))
)
/
sum without(instance)(
    sum without(handler)(rate(http_response_size_bytes_count[5m]))
)
```

Однако следующее решение было бы неправильным:

```
avg without(instance)(
    sum without(handler)(rate(http_response_size_bytes_sum[5m]))
)
/
sum without(handler)(rate(http_response_size_bytes_count[5m]))
```

Неправильно усреднять средние значения, так как деление и `avg` здесь будут вычислять средние значения.

i Агрегирование квантилей сводных метрик (временные ряды с меткой `quantile`) не имеет смысла с точки зрения статистики.

Гистограммы

Метрики-гистограммы позволяют исследовать распределение размеров событий и вычислять квантили. Например, с помощью гистограммы можно вычислить 0.9-й квантиль задержки (известной также как 90-й процентиль).

Prometheus 2.37.1 экспортирует метрику-гистограмму `prometheus_tsdb_compaction_duration_seconds`, сообщающую, сколько секунд занимает уплотнение базы данных временных рядов. Эта метрика имеет временной ряд с суффиксом `_bucket` – `prometheus_tsdb_compaction_duration_seconds_bucket`. Каждый сегмент в этой метрике-гистограмме имеет метку `le` – счетчик с количе-

¹ Тот же результат можно вычислить проще, как `sum without(instance, handler)(...)`, но, учитывая правила записи, описываемые в главе 17, такое выражение может быть разбито на несколько выражений.

ством событий, размер которых меньше или равен границе сегмента. Но это лишь деталь реализации, о которой редко приходится беспокоиться, так как функция `histogram_quantile` сама сделает все необходимое при вычислении квантилей. Например, квантиль 0.90 можно получить так:

```
histogram_quantile(
  0.90,
  rate(prometheus_tsdb_compaction_duration_seconds_bucket[1d]))
```

Поскольку `prometheus_tsdb_compaction_duration_seconds_bucket` является счетчиком, сначала нужно применить функцию `rate`. Уплотнение обычно происходит только раз в два часа, поэтому здесь используется временной диапазон, равный одним суткам. Вот примерно какой результат вернет этот запрос:

```
{instance="localhost:9090",job="prometheus"} 7.720000000000001
```

Он сообщает, что 90-й процентиль задержек уплотнения составляет около 7.72 с. Поскольку в течение суток производится всего 12 уплотнений, 90-й процентиль говорит о том, что 10 % операций уплотнения занимают больше времени, чем эти 7.72 с, а это одна или две операции уплотнения. Это обстоятельство следует учитывать при вычислении квантилей. Например, если вы хотите рассчитать квантиль 0.999, то у вас должно быть несколько тысяч точек данных, чтобы получить достаточно точный ответ. Если точек данных меньше, то отдельные выбросы могут сильно повлиять на результат, и в таких случаях следует подумать о возможности использования более низких квантилей, чтобы избежать ошибочных заявлений о работе системы.



Обычно при работе с гистограммами функции `rate` передаются 5- или 10-минутные интервалы. Все временные ряды сегментов в сочетании с любыми метками и большим диапазоном `rate` могут привести к созданию большого количества выборок для обработки. Будьте осторожны с выражениями PromQL, использующими диапазоны, равные часам или дням, поскольку их вычисление может оказаться относительно дорогим удовольствием¹.

Как и при получении средних значений, вызов `histogram_quantile` должен быть последним шагом в выражении запроса. Со статистической точки зрения квантили бессмысленно агрегировать или применять к ним арифметические операции. Соответственно, когда требуется получить гистограмму агрегированного значения, сначала выполните агрегирование с использованием `sum`, а затем применяйте `histogram_quantile`:

```
histogram_quantile(
  0.90,
  sum without(instance)(rate(prometheus_tsdb_compaction_duration_bucket[1d])))
```

¹ Диапазон длинной в сутки используется здесь только из-за ограниченного количества гистограмм, которые Prometheus и Node Exporter предлагают для использования в качестве примеров.

Этот запрос вычисляет квантиль 0.9 продолжительности уплотнения для всех серверов Prometheus и дает результат без метки `instance`:

```
{job="prometheus"} 7.720000000000001
```

Метрики-гистограммы также включают ряды `_sum` и `_count`, которые работают точно так же, как аналогичные ряды сводных метрик. Их можно использовать для расчета средних размеров событий, таких как средняя продолжительность уплотнения:

```
sum without(instance)(rate(prometheus_tsdb_compaction_duration_sum[1d]))  
/  
sum without(instance)(rate(prometheus_tsdb_compaction_duration_count[1d]))
```

Этот запрос вернет примерно такой результат:

```
{job="prometheus"} 3.1766430400714287
```

Селекторы

Работа с разными временными рядами метрики, имеющими разные значения меток, может оказаться сложной задачей и вызывать путаницу, если метрика поступает с нескольких серверов разных типов¹. Обычно требуется сузить временные ряды, участвующие в расчетах. В большинстве случаев желательно ограничиться меткой `job`, и в зависимости от конкретных требований вы можете захотеть, например, исследовать только один экземпляр (`instance`) или один обработчик (`handler`).

Такое ограничение по меткам осуществляется с помощью *селекторов*. Вы уже видели селекторы во всех показанных примерах, и теперь мы объясним их более подробно. Например,

```
process_resident_memory_bytes{job="node"}
```

это селектор, возвращающий все временные ряды с именем `process_resident_memory_bytes` и значением `node` в метке `job`. Этот конкретный селектор правильнее называть *селектором мгновенного вектора*, поскольку он возвращает значения из данного временного ряда, соответствующие данному моменту времени. Под словом *вектор* здесь подразумевается одномерный список, потому что селектор может вернуть ноль или более временных рядов, и каждый временной ряд будет иметь один образец.

`job="node"` называется *сопоставителем* (*matcher*), и в одном селекторе может быть указано несколько сопоставителей, которые объединяются по И (AND).

¹ Например, `process_cpu_seconds_total`, которая предоставляется большинством экспортёров и клиентских библиотек.

Сопоставители

Есть четыре сопоставителя (вы уже видели *сопоставитель равенства*, который также является самым часто используемым сопоставителем).

=

Это *сопоставитель равенства*; например, `job="node"`. Он позволяет указать, что возвращаемый временной ряд должен иметь заданную метку с заданным значением. Если задать пустое значение для метки, то это будет равносильно отсутствию метки, поэтому можно использовать выражение `foo=""`, чтобы указать, что метка `foo` отсутствует.

!=

Это *сопоставитель неравенства*; например, `job!="node"`. Он позволяет указать, что возвращаемый временной ряд должен иметь заданную метку со значением, отличным от заданного.

=~

Это *сопоставитель соответствия регулярному выражению*; например, `job=~"n.*"`. Он позволяет указать, что возвращаемый временной ряд должен иметь заданную метку, значение которой соответствует заданному регулярному выражению. Регулярное выражение неявно получает якорные символы, т. е. регулярное выражение `a`, например, будет соответствовать только строке `a`, но не будет соответствовать строкам `xa` или `ax`. Перед регулярным выражением и/или после него можно добавить комбинацию `.*`, если не требуется такое строгое поведение¹. Как и в механизме изменения меток, здесь используется движок регулярных выражений RE2, как описано в разделе «Регулярные выражения» главы 8.

!~

Это *сопоставитель несоответствия регулярному выражению*. RE2 не поддерживает негативные опережающие проверки (negative lookahead), поэтому был добавлен этот сопоставитель, предоставляющий альтернативный способ исключения значений меток на основе регулярного выражения.

Допускается задавать несколько сопоставителей с одним и тем же именем метки в селекторе для имитации негативной опережающей проверки. Например, вот как можно найти размер всех файловых систем, смонтированных в `/run`, но не в `/run/user`²:

```
node_filesystem_size_bytes{job="node",mountpoint=~"/run/.*",  
                         mountpoint!~/run/user/.*"}

---


```

¹ Такая строгость обусловлена стремлением исключить вероятность случайного совпадения. Благодаря этому вы сразу же сможете заметить ошибку, если разрабатываемое регулярное выражение не находит совпадения там, где оно должно обнаруживаться. Если бы регулярное выражение не получало неявно якорных символов, то возможность случайного совпадения не там, где нужно, могла бы вызвать трудноуловимые проблемы в будущем.

² Node Exporter поддерживает флаг `--collector.filesystem.ignored-mount-points`, который можно использовать для той же цели, если по какой-то причине нежелательно экспортить информацию обо всех файловых системах.

Внутренне имя метрики хранится в метке `__name__` (как обсуждалось во врезке «Зарезервированные метки и `__name__`» в главе 5), поэтому `process_resident_memory_bytes{job="node"}` фактически является синтаксическим сахаром для `{name="process_resident_memory_bytes", job="node"}`. Вместо имен меток даже можно использовать регулярные выражения, но это имеет смысл только для отладки работы сервера Prometheus.



Использование сопоставителей с регулярными выражениями считается дурным тоном. Поэтому, обнаружив, что часто используете их для какой-то одной метки, подумайте, не следует ли объединить совпадающие значения метки в одно. Например, анализируя коды состояния HTTP, вместо `code~="4.."`, чтобы перехватить 401, 404, 405 и т. д., можно объединить их в значение метки `4xx` и использовать сопоставитель равенства `code="4xx"`.

Селектор `{}` возвращает ошибку, что является мерой предосторожности, позволяющей избежать случайного возврата всех временных рядов внутри сервера Prometheus, так как это может стоить очень дорого. Точнее говоря, хотя бы одно из совпадений в селекторе не должно совпадать с пустой строкой. Соответственно, `{foo=""}` и `{foo=~".*"}` вернут ошибку, а `{foo="", bar="x"}, {foo!=""}` или `{foo=~".+"}` выполняются нормально¹.

Мгновенный вектор

Селектор мгновенного вектора возвращает мгновенный вектор с самыми последними значениями, полученными к моменту выполнения запроса, т. е. список из нуля или более временных рядов. Каждый из этих временных рядов будет иметь один образец, а образец будет содержать значение и отметку времени. Мгновенный вектор, возвращаемый селектором мгновенного вектора, будет содержать отметки времени из исходных данных², но любые мгновенные векторы, возвращаемые другими операциями или функциями, будут содержать отметки времени, соответствующие моменту выполнения запроса.

При запросе текущего объема использованной памяти нежелательно, чтобы в вектор включались образцы, полученные несколько дней назад, из-за устаревания информации. В Prometheus 1.x эта проблема решалась путем возврата временных рядов с экземплярами, полученными не более чем за 5 мин до момента выполнения запроса. Это вполне работоспособное решение, но оно имело такие недостатки, как двойной подсчет, если экземпляр перезапускался с новой меткой `instance` в течение этого 5-минутного окна.

Prometheus 2.x реализует более сложный подход. Если временной ряд исчезает после одного из циклов опроса или если цель больше не возвращается механизмом обнаружения служб, то в конец временного ряда до-

¹ Если вам понадобится вернуть все временные ряды, то используйте `{__name__=~".+"}`, но помните о возможных затратах, которые влечет это выражение.

² Отметку времени можно извлечь из образца с помощью функции `timestamp`.

бавляется образец специального типа, называемый *маркером устаревания*¹. При оценке селектора мгновенного вектора сначала выполняется поиск всех временных рядов, удовлетворяющих всем сопоставителям, и по-прежнему извлекается самый последний образец из 5-минутного интервала, предшествующего времени выполнения запроса. Если образец не является маркером устаревания, то он помещается в возвращаемый мгновенный вектор, но если это маркер устаревания, то такой временной ряд не включается в мгновенный вектор.

В результате при использовании селектора мгновенных векторов устаревшие временные ряды не возвращаются.

- i** Если у вас есть экспортер, предоставляющий отметки времени, как описано в разделе «Отметки времени» главы 4, то маркеры устаревания и соответствующая логика Prometheus 2.x не применяются. Вместо этого такие временные ряды будут обрабатываться старой логикой, которая просто выбирает образцы из последнего 5-минутного интервала.

Вектор диапазона

Существуют также селекторы второго типа, называемые *селекторами векторов диапазона*. Их вы тоже уже видели. В отличие от селектора мгновенного вектора, возвращающего единственный образец из каждого соответствующего временного ряда, селектор вектора диапазона может вернуть множество образцов². Векторы диапазонов всегда используются с функцией `rate`, например:

```
rate(process_cpu_seconds_total[1m])
```

[1м] превращает селектор мгновенного вектора в селектор вектора диапазона и сообщает движку PromQL, что он должен вернуть все образцы из всех временных рядов, соответствующих селектору, полученные за минуту до времени выполнения запроса. Если выполнить только `process_cpu_seconds_total[1m]` на вкладке **Console** браузера выражений, то результат будет выглядеть примерно так, как показано на рис. 13.1.

В данном случае в каждом временном ряду оказалось по шесть образцов, полученных за последнюю минуту. Обратите внимание, что образцы в каждом временном ряду располагаются с шагом 10 с³, согласно настроенному интервалу опроса, но отметки времени двух временных рядов не совпадают.

¹ Внутренне маркеры устаревания реализованы как особый тип значения NaN. Но это внутренняя деталь реализации, и она недоступна напрямую через API запросов, используемых движком PromQL. Однако эти значения можно увидеть, если заглянуть непосредственно в хранилище сервера Prometheus, например, через конечную точку Prometheus для удаленного чтения.

² Его также иногда называют *матрицей*, так как это двумерная структура данных.

³ Это очень малонагруженный сервер Prometheus, поэтому разбросов по времени не наблюдается.

ют друг с другом. Один временной ряд имеет образец с отметкой времени 1517925155.087, а другой – 1517925156.245.

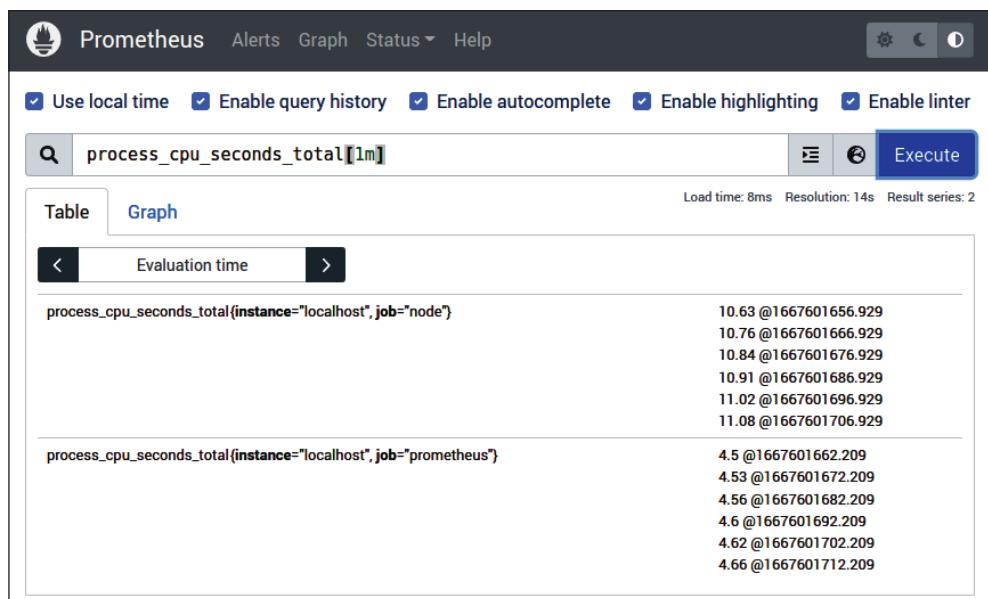


Рис. 13.1 ♦ Вектор диапазона на вкладке **Console** (Консоль) в браузере выражений

Это связано с тем, что векторы диапазонов сохраняют фактические отметки времени образцов, а моменты опроса разных целей распределяются для более равномерной нагрузки. Имейте в виду, что вы можете контролировать частоту опроса и оценки правил, но не можете контролировать конкретные моменты их выполнения. Если задать 10-секундный интервал опроса сотни целей, то все эти цели будут опрашиваться в разные моменты в заданном 10-секундном окне. Иначе говоря, все временные ряды будут иметь немного разный возраст. На практике это не имеет большого значения, но может приводить к некоторому недопониманию. Запомните, что системы мониторинга, основанные на метриках, такие как Prometheus, дают (довольно хорошие) оценки, но не точные ответы.

Вам редко придется исследовать векторы диапазонов напрямую. Обычно необходимость в этом возникает только при отладке, когда необходимо видеть исходные образцы. Почти всегда вы будете использовать векторы диапазонов с такими функциями, как `rate` и `avg_over_time`, принимающими вектор диапазона в качестве аргумента.

Понятие устаревания неприменимо к векторам диапазонов, и маркеры устаревания не включаются в них; вы получите все образцы в заданном вами диапазоне. Любые маркеры устаревания, присутствующие в этом диапазоне не возвращаются селектором вектора диапазона.

Интервалы времени

Интервалы времени в запросах PromQL и в конфигурационных файлах Prometheus можно задавать с использованием разных единиц измерения. С минутами (`m`) вы уже знакомы.

Суффикс	Значение
ms	Миллисекунды
s	Секунды, 1 секунда = 1000 миллисекунд
m	Минуты, 1 минута = 60 секунд
h	Часы, 1 час = 60 минут
d	День (сутки), 1 день = 24 часа
w	Неделя, 1 неделя = 7 дней
y	Год, 1 год = 365 дней

Допускается комбинировать несколько единиц измерения с целыми числами. Единственное условие: они должны следовать в порядке увеличения точности. Например, `90m`, `1h30m` и `1.5h` – допустимые значения, а `30m1h` – недопустимое.

Високосные годы не поддерживаются; `1y` всегда содержит $60*60*24*365$ секунд.

Подзапросы

Несмотря на то что векторы диапазонов применяются к временным рядам, их нельзя использовать в сочетании с функциями.

Если потребуется объединить `max_over_time` с `rate`, то в таком случае можно использовать правила записи, которые будут записывать результат функции `rate` и передавать его векторной функции, или подзапрос.

Подзапрос – это часть запроса, позволяющая выполнять запрос диапазона внутри запроса. Подзапросы должны заключаться в квадратные скобки, подобно селекторам диапазонов. Но для этого нужно указать два разных интервала: диапазон и разрешение.

Диапазон – это интервал, возвращаемый подзапросом, а разрешение действует как шаг:

```
max_over_time( rate(http_requests_total[5m])[30m:1m] )
```

Предыдущий запрос будет вызывать `rate(http_requests_total[5m])` каждую минуту (`1m`) в течение последних 30 мин (`30m`), а затем передаст результат в функцию `max_over_time()`.

Разрешение можно не указывать, например `[30m:]`. В этом случае в качестве разрешения используется глобальный интервал оценки.

Смещение

При работе с любыми векторными селекторами можно также использовать дополнительный модификатор смещения `offset`. Он может сдвинуть время вычисления запроса назад во времени для каждого селектора. Например:

```
process_resident_memory_bytes{job="node"} offset 1h
```

даст потребление памяти за час до момента выполнения запроса.

Модификатор `offset` обычно не используется в таких простых запросах, как этот, так как гораздо проще изменить начальное время для всего запроса. Но он может пригодиться, когда требуется настроить только один селектор в выражении запроса. Например:

```
process_resident_memory_bytes{job="node"}  
-  
  process_resident_memory_bytes{job="node"} offset 1h
```

вернет изменение потребления памяти в Node Exporter за последний час¹.

Тот же подход можно применить к векторам диапазонов:

```
rate(process_cpu_seconds_total{job="node"})[5m]  
-  
  rate(process_cpu_seconds_total{job="node"})[5m] offset 1h
```

`offset` позволяет заглянуть не только в прошлое, но и в будущее, если использовать отрицательное смещение. Эту особенность можно использовать при прогнозировании или когда образец метрики не соответствует действительности:

```
rate(process_cpu_seconds_total{job="node"})[5m] offset -1h  
-  
  rate(process_cpu_seconds_total{job="node"})[5m]
```

Обратите внимание, что этот запрос, скорее всего, ничего не вернет за последний час.



В Grafana поддерживается возможность смещения панели в другой временной диапазон, отличный от временного диапазона всего дашборда, частью которого она является. В Grafana 5.0.0 соответствующие настройки можно найти на вкладке **Time range** (Временной диапазон) редактора панели.

¹ Этот прием восприимчив к выбросам, так как используются только две точки данных; функция `deriv`, описанная в разделе «`deriv`» главы 16, дает более надежные результаты.

Модификатор @

Помимо модификатора `offset`, PromQL поддерживает еще один похожий модификатор – `@`. Он позволяет получить результат селекторов векторов, селекторов диапазонов и подзапросов, соответствующий конкретному моменту времени.

Модификатору `@` можно передать отметку времени Unix. Запрос `http_requests_total @ 1667491200` вернет значение `http_requests_total` на момент 2022-11-03T16:00:00+00:00. Запрос `rate(http_requests_total[5m] @ 1667491200)` вернет частоту `http_requests_total` за 5-минутный интервал, начинающийся в тот же момент времени.

Также модификатору `@` можно передать функции `start()` и `end()`. В запросе диапазона они разрешаются как начало и конец диапазона соответственно. В запросе мгновенного значения обе функции разрешаются как время запуска запроса.

На практике модификатор `@` можно использовать для построения графика изменения `http_request_total`, когда частота увеличивается в конце интервала оценки:

```
rate(http_requests_total[1m])
and
topk(5, rate(http_requests_total[1h] @ end()))
```

`topk(5, rate(http_requests_total[1h] @ end()))` действует как функция ранжирования, фильтруя более высокие значения в конце интервала оценки.

HTTP API

Prometheus предлагает несколько HTTP API. В основном вы будете работать с `query` и `query_range`, открывающими доступ к PromQL. Их можно использовать в дашбордах или в собственных сценариях отчетов.

Все конечные точки находятся в `/api/v1/` и, помимо выполнения запросов PromQL, позволяют также просматривать метаданные временных рядов и выполнять административные действия, такие как создание моментальных снимков и удаление временных рядов. Эти другие API в основном представляют интерес для систем создания дашбордов, таких как Grafana, которые могут использовать метаданные для улучшения пользовательского интерфейса, и для тех, кто занимается администрированием Prometheus, но не выполняют запросов PromQL.

query

Конечная точка `query`, или, более формально, `/api/v1/query`, выполняет выражение PromQL в заданное время и возвращает результат. Например, `http://`

`localhost:9090/api/v1/query?query=process_resident_memory_bytes` вернет такие результаты¹:

```
{
  "status": "success",
  "data": {
    "resultType": "vector",
    "result": [
      {
        "metric": {
          "__name__": "process_resident_memory_bytes",
          "instance": "localhost:9090",
          "job": "prometheus"
        },
        "value": [1517929228.782, "91656192"]
      },
      {
        "metric": {
          "__name__": "process_resident_memory_bytes",
          "instance": "localhost:9100",
          "job": "node"
        },
        "value": [1517929228.782, "15507456"]
      }
    ]
  }
}
```

Поле `status` имеет значение `success`, сообщающее, что запрос выполнен успешно. В случае неудачи в поле `status` было бы возвращено значение `error`, а в поле `error` было бы представлено подробное описание проблемы.

Этот конкретный результат представляет собой мгновенный вектор, что можно определить по `"resultType": "vector"`. Для каждого из образцов в результате метки находятся в словаре `metric`, а значения образцов – в списке `value`. Первое число в списке `value` – это отметка времени образца в секундах, а второе – фактическое значение. Значение представлено в виде строки, потому что JSON не может представлять несуществующие значения, такие как `NaN` и `+Inf`.

Для всех образцов устанавливается время запроса, даже если выражение состоит из единственного селектора мгновенного вектора. В этом примере время запроса по умолчанию равно текущему времени, но есть возможность указать другое время с помощью параметра URL `time`, в котором можно передать отметку времени Unix в секундах или время в формате RFC 3339. Например, `http://localhost:9090/api/v1/query?query=process_resident_memory_bytes&time=1514764800` вернет результат, соответствующий полуночи 1 января 2018 года².

¹ Мы немного отформатировали результат для удобства чтения.

² Если ваш сервер Prometheus не работал в то время, то запрос вернет пустой результат.

Из конечной точки `query` также можно получить векторы диапазонов. Например, `http://localhost:9090/api/v1/query?query=prometheus_tsdb_head_samples_appended_total[1m]` вернет такой результат:

```
{
  "status": "success",
  "data": {
    "resultType": "matrix",
    "result": [
      {
        "metric": {
          "__name__": "process_resident_memory_bytes",
          "instance": "localhost:9090",
          "job": "prometheus"
        },
        "values": [
          [1518008453.662, "87318528"],
          [1518008463.662, "87318528"],
          [1518008473.662, "87318528"]
        ]
      },
      {
        "metric": {
          "__name__": "process_resident_memory_bytes",
          "instance": "localhost:9100",
          "job": "node"
        },
        "values": [
          [1518008444.819, "17043456"],
          [1518008454.819, "17043456"],
          [1518008464.819, "17043456"]
        ]
      }
    ]
  }
}
```

Он отличается от предыдущего результата с мгновенным вектором, о чем свидетельствует поле `resultType` со значением `matrix` и наличие нескольких значений в каждом временном ряду. Когда запрос к конечной точке `query` содержит селектор векторов диапазонов, она возвращает необработанные образцы¹, но будьте осторожны, запрашивая слишком много данных за один раз, потому что на стороне сервера или клиента может не хватить памяти.

Существует еще один тип результата, называемый *скаляром*. У скаляров нет меток, это просто числа². Например, запрос `http://localhost:9090/api/v1/query?query=42` вернет:

¹ Без маркеров устаревания.

² Это не то же самое, что пустые фигурные скобки {}, которые идентифицируют временной ряд без меток.

```
{
  "status": "success",
  "data": {
    "resultType": "scalar",
    "result": [1518008879.023, "42"]
  }
}
```

query_range

Конечная точка *query_range* (точнее, */api/v1/query_range*) – это наиболее часто используемая конечная точка Prometheus HTTP API. Обычно она используется для построения графиков. По сути, *query_range* – это синтаксический сахар (плюс некоторые оптимизации производительности), замещающий несколько вызовов конечной точки *query*.

В дополнение к параметру URL *query* необходимо также передать параметр *query_range* с временем начала (*start*), временем окончания (*end*) и шагом (*step*). Сначала выполняется запрос с временем *start*. Затем с временем через *step* секунд после *start*. Затем с временем через $2 * step$ секунд после *start* и т. д., пока не будет превышено время *end*. Все полученные мгновенные векторы¹ объединяются в вектор диапазона и возвращаются.

Например, чтобы получить количество образцов, загруженных Prometheus за первые 15 мин 2018 года, можно выполнить запрос *http://localhost:9090/api/v1/query_range?query=rate(prometheus_tsdb_head_samples_appended_total[5m])&start=1514764800&end=1514765700&step=60*, который вернет следующий результат:

```
{
  "status": "success",
  "data": {
    "resultType": "matrix",
    "result": [
      {
        "metric": {
          "instance": "localhost:9090",
          "job": "prometheus"
        },
        "values": [
          [1514764800, "85.07241379310345"],
          [1514764860, "102.6793103448276"],
          [1514764920, "120.30344827586208"],
          [1514764980, "137.93103448275863"],
          [1514765040, "146.7586206896552"],
          [1514765100, "146.7793103448276"],
          [1514765160, "146.8"]
        ]
      }
    ]
  }
}
```

¹ Скалярный результат преобразуется в мгновенный вектор с одним временным рядом без меток с одинаковыми значениями образцов, как если бы использовалась функция *vector*. Результаты в виде вектора диапазона не поддерживаются.

```
[1514765220, "146.8"],
[1514765280, "146.8"],
[1514765340, "146.8"],
[1514765400, "146.8"],
[1514765460, "146.8"],
[1514765520, "146.8"],
[1514765580, "146.8"],
[1514765640, "146.8"],
[1514765700, "146.8"],
]
}
]
}
}
```

Однако, выполняя такие запросы, вы должны учитывать несколько аспектов. Во-первых, отметки времени образцов будут совпадать с временем `start` и изменяться с шагом `step`, потому что каждый образец является результатом другого запроса, возвращающего мгновенный вектор, а мгновенные результаты всегда содержат время, заданное в запросе.

Во-вторых, последний образец здесь имеет время `end`, т. е. диапазон является inkluzivным, и последняя точка данных будет иметь конечное время `end`, если она совпадает с шагом.

В-третьих, в функции `rate` мы указали диапазон 5 мин, который больше шага `step`. Причина такого поступка в том, что `query_range` выполняет мгновенные запросы, не сохраняя состояния между ними. Указав диапазон меньше шага, мы бы пропустили данные. Например, 1-минутный диапазон с 5-минутным шагом приведет к потере 80 % образцов. Чтобы предотвратить это, диапазоны должны быть больше шага хотя бы на величину одного или двух интервалов опроса.

! При использовании векторов диапазонов с `query_range` используйте диапазон длиннее шага `step`, чтобы не пропустить данные.

В-четвертых, некоторые образцы не особенно круглые, что обусловлено особенностями настройки выборки значений. При работе с метриками вы редко будете получать идеальные данные; разные цели опрашиваются в разное время, и опрос некоторых целей может откладываться. При выполнении запросов, не полностью согласованных с базовыми данными, или при агрегировании по нескольким хостам вы редко будете получать круглые результаты. Кроме того, характер вычислений с плавающей точкой может привести к тому, что числа будут почти, но не полностью круглыми.

В этом примере для каждого шага имеется образец. Если для какого-то шага в заданном временном ряду нет результата, то соответствующий образец будет отсутствовать в конечном результате.

i Если диапазон для `query_range` насчитывает более 11 000 шагов, то Prometheus отклонит запрос и вернет признак ошибки. Это сделано для предотвращения случайной отправки чрезвычайно больших запросов, таких как с диапазоном в неделю и с ша-

том в 1 с. Поскольку мониторы с горизонтальным разрешением более 11 000 пикселей встречаются редко, вы вряд ли столкнетесь с этой проблемой при построении графиков.

При разработке сценариев отчетов можно разделить запросы `queu_range`, превышающие это ограничение. Это ограничение не будет превыщено при попытке получить данные за неделю с разрешением в 1 мин или за год с разрешением в 1 час, поэтому в большинстве случаев можно не опасаться его превышения.

Согласование данных

При использовании таких инструментов, как Grafana, запросы к `queu_range` обычно согласовываются с текущим временем, поэтому получаемые результаты не будут идеально согласовываться с минутами, часами или днями. Это обычное дело для информационных дашбордов, но часто нежелательно для сценариев отчетов.

`queu_range` не имеет параметров согласования, зато позволяет указать параметр `start` с правильно выбранным временем начала. Например, чтобы получать данные каждый час за час, в Python можно использовать выражение `(time.time() // 3600) * 3600`, возвращающее начало текущего часа¹, которое можно использовать в качестве значений параметров URL `start` и `end`, и задать параметр `step=3600`.

Теперь, познакомившись с основами PromQL и приемами выполнения запросов через HTTP API, перейдем к агрегированию.

¹ Оператор `//` выполняет целочисленное деление в Python.

Глава 14

Операторы агрегирования

Вы уже познакомились с агрегированием в разделе «Основы агрегирования» главы 13; однако это лишь малая часть имеющихся возможностей. Агрегирование играет важную роль. Для приложений с тысячами или даже десятками экземпляров нецелесообразно просеивать метрики каждого экземпляра по отдельности. Агрегирование позволяет суммировать метрики не только в рамках одного приложения, но и по всем приложениям.

В PromQL есть 12 функций агрегирования, поддерживающих два необязательных модификатора: `without` и `by`. В этой главе вы познакомитесь с некоторыми разными приемами агрегирования.

Группировка

Прежде чем начать обсуждение функций агрегирования, давайте посмотрим, как группируются временные ряды. Функции агрегирования работают только с мгновенными векторами и возвращают только мгновенные векторы.

Допустим, у вас есть следующие временные ряды:

```
node_filesystem_size_bytes{device="/dev/sda1",fstype="vfat",
  instance="localhost:9100",job="node",mountpoint="/boot/efi"} 100663296
node_filesystem_size_bytes{device="/dev/sda5",fstype="ext4",
  instance="localhost:9100",job="node",mountpoint="/" } 90131324928
node_filesystem_size_bytes{device="tmpfs",fstype="tmpfs",
  instance="localhost:9100",job="node",mountpoint="/run" } 826961920
node_filesystem_size_bytes{device="tmpfs",fstype="tmpfs",
  instance="localhost:9100",job="node",mountpoint="/run/lock" } 5242880
node_filesystem_size_bytes{device="tmpfs",fstype="tmpfs",
  instance="localhost:9100",job="node",mountpoint="/run/user/1000" } 826961920
node_filesystem_size_bytes{device="tmpfs",fstype="tmpfs",
  instance="localhost:9100",job="node",mountpoint="/run/user/119" } 826961920
```

Здесь есть три инструментированные метки: `device`, `fstype` и `mountpoint`, а также две целевые метки: `job` и `instance`. Вы уже знаете, чем отличаются целевые и инструментированные метки, но PromQL их никак не различает. Для PromQL все метки одинаковы независимо от их происхождения.

without

Как правило, инструментированные метки известны всегда, так как они редко меняются. Но это не относится к целевым меткам, так как написанное вами выражение может быть использовано кем-то другим для изучения метрик, полученных с применением разных конфигураций опроса, или серверами Prometheus, которые могут добавить другие целевые метки к заданиям, например `env` или `cluster`. В какой-то момент вы даже можете сами добавить такие целевые метки, и было бы неплохо не обновлять все свои выражения.

При агрегировании метрик обычно желательно сохранить такие целевые метки, поэтому при агрегировании следует использовать модификатор `without`, чтобы указать метки, которые нужно удалить. Например, запрос

```
sum without(fstype, mountpoint)(node_filesystem_size_bytes)
```

объединит временные ряды в три группы, игнорируя метки `fstype` и `mountpoint`:

```
# Group {device="/dev/sda1",instance="localhost:9100",job="node"}
node_filesystem_size_bytes{device="/dev/sda1",fstype="vfat",
    instance="localhost:9100",job="node",mountpoint="/boot/efi"} 100663296

# Group {device="/dev/sda5",instance="localhost:9100",job="node"}
node_filesystem_size_bytes{device="/dev/sda5",fstype="ext4",
    instance="localhost:9100",job="node",mountpoint="/" } 90131324928

# Group {device="tmpfs",instance="localhost:9100",job="node"}
node_filesystem_size_bytes{device="tmpfs",fstype="tmpfs",
    instance="localhost:9100",job="node",mountpoint="/run"} 826961920
node_filesystem_size_bytes{device="tmpfs",fstype="tmpfs",
    instance="localhost:9100",job="node",mountpoint="/run/lock"} 5242880
node_filesystem_size_bytes{device="tmpfs",fstype="tmpfs",
    instance="localhost:9100",job="node",mountpoint="/run/user/1000"} 826961920
node_filesystem_size_bytes{device="tmpfs",fstype="tmpfs",
    instance="localhost:9100",job="node",mountpoint="/run/user/119"} 826961920
```

и к каждой применит агрегирующую функцию `sum`, чтобы сложить значения в каждом временном ряду, и вернет по одному образцу для каждой группы:

```
{device="/dev/sda1",instance="localhost:9100",job="node"} 100663296
{device="/dev/sda5",instance="localhost:9100",job="node"} 90131324928
{device="tmpfs",instance="localhost:9100",job="node"} 2486128640
```

Обратите внимание, что метки `instance` и `job` сохраняются, как и любые другие имеющиеся метки. Это полезно, потому что любые созданные вами оповещения, включающие это выражение, сохранят дополнительные целе-

вые метки, такие как `env` или `cluster`, что обеспечит контекст для предупреждений и сделает их более полезными.

Имя метрики было удалено, так как агрегирование выполняется не для исходной метрики, а для `node_filesystem_size_bytes`. Когда оператор или функция PromQL может изменить значение или смысл временного ряда, имя метрики удаляется.

Также модификатор `without` можно использовать без меток. Например:

```
sum without()(node_filesystem_size_bytes)
```

Это выражение даст тот же результат, что и

```
node_filesystem_size_bytes
```

Единственное отличие – в первом случае имя метрики удаляется.

by

В дополнение к `without` есть также модификатор `by`. Но, в отличие от `without`, который указывает, какие метки удалить, модификатор `by` указывает, какие метки оставить. Соответственно, при использовании `by` требуется некоторая осторожность, чтобы не удалить целевые метки, которые вы собирались рассылать в предупреждениях или использовать в дашбордах. Имейте в виду, что в одном выражении агрегирования нельзя использовать сразу оба модификатора, `by` и `without`.

Запрос

```
sum by(job, instance, device)(node_filesystem_size_bytes)
```

даст тот же результат, что и запрос с модификатором `without` в предыдущем разделе:

```
{device="/dev/sda1",instance="localhost:9100",job="node"} 100663296
{device="/dev/sda5",instance="localhost:9100",job="node"} 90131324928
{device="tmpfs",instance="localhost:9100",job="node"} 2486128640
```

Если бы метки `instance` или `job` не были указаны, то для них не были бы определены и сформированы группы в выводе. Обычно по этой причине предпочтительнее использовать `without`.

Однако в двух случаях модификатор `by` действительно можете быть очень полезен. Во-первых, в отличие от `without`, модификатор `by` сохраняет метку `__name__`, если она указана явно. Это позволяет использовать такие выражения, как

```
sort_desc(count by(__name__)({__name__=~".+"}))
```

чтобы выяснить, сколько временных рядов имеют одинаковые имена метрик¹.

¹ Это потенциально дорогостоящий запрос, поскольку затрагивает все активные временные ряды; используйте его с осторожностью.

Во-вторых, `by` удобно использовать, когда требуется удалить все метки, о которых вы не знаете. Например, информационные метрики, как обсуждалось в разделе «Информационные метрики» главы 5, со временем будут добавлять все больше меток. Чтобы подсчитать, сколько машин работает с каждой версией ядра, можно использовать такой запрос:

```
count by(release)(node_uname_info)
```

который в нашем тестовом окружении с одной машиной возвращает

```
{release="4.4.0-101-generic"} 1
```

Вы можете использовать функцию `sum` с пустым модификатором `by` и даже опустить `by`. То есть

```
sum by()(node_filesystem_size_bytes)
```

или

```
sum(node_filesystem_size_bytes)
```

Оба запроса действуют совершенно идентично и дают одинаковый результат, такой как:

```
[] 92718116864
```

Это единственный временной ряд, и он не имеет меток.

Если выполнить выражение

```
sum(non_existent_metric)
```

то результатом будет мгновенный вектор без временных рядов, отображаемый на вкладке **Console** (Консоль) в браузере выражений как «*no data*» (нет данных).



Если функции агрегирования передать пустой мгновенный вектор, то она вернет пустой мгновенный вектор. Поэтому `count by(foo)(non_existent_metric)` вернет пустое значение, а не 0, так как `count` не получила никаких меток для подсчета. `count(non_existent_metric)` действует точно так же и тоже возвращает пустой мгновенный вектор.

Операторы

Все 11 операторов агрегирования используют одинаковую логику группировки, работой которой можно управлять с помощью модификаторов `without` и `by`. Но они по-разному обрабатывают сгруппированные данные.

sum

`sum` – самый распространенный оператор агрегирования; он суммирует все значения в группе и возвращает полученную сумму. Например,

```
sum without(fstype, mountpoint, device)(node_filesystem_size_bytes)
```

вернет общий размер файловых систем на каждой из машин.

При работе со счетчиками¹ важно не забывать вызывать `rate` перед агрегированием с помощью `sum`:

```
sum without(device)(rate(node_disk_read_bytes_total[5m]))
```

Прямое суммирование счетчиков вернет бессмысленный результат, так как разные счетчики могли быть инициализированы в разное время в зависимости от момента запуска или перезапуска экспортера или когда впервые были использованы конкретные потомки.

count

Оператор `count` подсчитывает количество временных рядов в группе и возвращает полученное число. Например,

```
count without(device)(node_disk_read_bytes_total)
```

вернет количество дисковых устройств на машине. Наша тестовая машина имеет только один диск, поэтому мы получаем:

```
{instance="localhost:9100",job="node"} 1
```

Здесь можно не использовать `rate`, так как здесь нас интересует наличие временного ряда, а не его значения.

Уникальные значения меток

Оператор `count` также можно использовать для подсчета уникальных значений меток. Например, вот как можно подсчитать количество процессоров на каждой из машин:

```
count without(cpu)(count without (mode)(node_cpu_seconds_total))
```

Вложенный вызов `count`² удаляет инструментированную метку `mode`, возвращая по одному временному ряду для каждого процессора каждого экземпляра:

```
{cpu="0",instance="localhost:9100",job="node"} 8
{cpu="1",instance="localhost:9100",job="node"} 8
{cpu="2",instance="localhost:9100",job="node"} 8
{cpu="3",instance="localhost:9100",job="node"} 8
```

¹ Включая `_sum`, `_count` и `_bucket` гистограмм и сводных метрик.

² Внутреннее агрегирование не обязательно должно производиться с помощью `count`; с тем же успехом можно использовать любой другой агрегатор, возвращающий тот же набор временных рядов, например `sum`. Это связано с тем, что внешний вызов `count` счетчик игнорирует значения этих временных рядов.

Затем внешний вызов `count` возвращает количество процессоров в каждом экземпляре:

```
{instance="localhost:9100",job="node"} 4
```

Если не нужна разбивка по машинам, например если вы изучаете, имеют ли определенные метки высокую кардинальность, то можно использовать модификатор `by`, чтобы просмотреть только одну метку:

```
count(count by(cpu)(node_cpu_seconds_total))
```

Это выражение вернет единственный образец без меток:

```
{} 4
```

avg

Оператор `avg` возвращает среднее значение¹ временных рядов в группе. Например,

```
avg without(cpu)(rate(node_cpu_seconds_total[5m]))
```

даст среднее потребление процессора в каждом режиме каждым экземпляром Node Exporter:

```
{instance="localhost:9100",job="node",mode="idle"} 0.9095948275861836
{instance="localhost:9100",job="node",mode="iowait"} 0.005543103448275879
{instance="localhost:9100",job="node",mode="irq"} 0
{instance="localhost:9100",job="node",mode="nice"} 0.0013620689655172522
{instance="localhost:9100",job="node",mode="softirq"} 0.0001465517241379329
{instance="localhost:9100",job="node",mode="steal"} 0
{instance="localhost:9100",job="node",mode="system"} 0.015836206896552414
{instance="localhost:9100",job="node",mode="user"} 0.06054310344827549
```

Это тот же результат, который возвращает запрос:

```
sum without(cpu)(rate(node_cpu_seconds_total[5m]))
/
count without(cpu)(rate(node_cpu_seconds_total[5m]))
```

но запрос с `avg` более краткий и действует более эффективно.

При использовании `avg` иногда можно обнаружить, что присутствие значения `NaN` во входных данных приводит к возврату `NaN` в результате. Это связано с тем, что любые арифметические операции с плавающей запятой, получающие `NaN`, в результате возвращают `NaN`.

Вы можете задаться вопросом, как отфильтровать эти значения `NaN` из входных данных, но это неправильный вопрос. Обычно это происходит из-за

¹ Технически это среднее арифметическое. В редких случаях, когда может понадобиться среднее геометрическое, используйте функции `ln` и `exp` в сочетании с агрегатором `avg`.

попытки усреднить средние значения, и один из знаменателей первых средних значений оказался равным 0¹. Усреднение средних значений не имеет смысла с точки зрения статистики, вместо этого следует выполнить суммирование с помощью `sum` и затем деление, как показано в разделе «Сводные метрики» главы 13.

group

Оператор `group` возвращает 1 для каждого временного ряда в группе. Например:

```
count by (instance)(  
    group by (fstype,instance) (node_filesystem_files)  
)
```

Этот запрос вернет количество файловых систем разных типов в каждом экземпляре.

В данном случае вместо `group` можно использовать любой другой агрегатор (`sum`, `count`). Однако использование `group` помогает понять всем, кто читает запрос, что нас интересует группировка и сами результирующие метки, а не значение, полученное внутренним оператором агрегирования.

stddev и stdvar

Стандартное отклонение – это статистическая мера разброса набора чисел. Например, для набора чисел [2, 4, 6] стандартное отклонение равно 1.633². Набор чисел [3, 4, 5] имеет то же среднее значение 4, но его стандартное отклонение равно 0.816.

Основное применение стандартного отклонения в мониторинге – обнаружение выбросов. В случае нормально распределенных данных можно ожидать, что около 68 % образцов будут находиться в пределах одного стандартного отклонения от среднего значения, а 95 % – в пределах двух стандартных отклонений³. Если один экземпляр в задании имеет метрику, отличающуюся на несколько стандартных отклонений от среднего, то это верный признак того, что с ним что-то не так.

Например, вот как можно найти все экземпляры, превышающие среднее значение на два стандартных отклонения или больше:

¹ Это как $1/0 = \text{NaN}$.

² Prometheus вычисляет *стандартное отклонение генеральной совокупности*, а не *стандартное отклонение выборки*, потому что обычно вы будете просматривать все интересующие вас значения, а не случайное их подмножество.

³ Для данных с ненормальным распределением *неравенство Чебышева* дает более узкие границы.

```

some_gauge
> ignoring (instance) group_left()
(
    avg without(instance)(some_gauge)
    +
    2 * stddev without(instance)(some_gauge)
)

```

При этом используется сопоставление векторов «один ко многим», которое мы обсудим в разделе «Многие к одному и group_left» главы 15. Если все значения плотно сгруппированы, то этот запрос может вернуть некоторые временные ряды, превышающие среднее более чем на два стандартных отклонения, но достаточно близкие к среднему и характеризующие нормальную работу. Для защиты от подобных ситуаций можно добавить дополнительный фильтр, требующий, чтобы значение было по крайней мере на 20 % выше среднего. Это также один из тех редких случаев, когда имеет смысл получать среднее значение от средних, например для получения средней задержки.

Стандартная дисперсия – это квадрат стандартного отклонения¹ и широко используется в статистике.

min и max

Агрегаторы `min` и `max` возвращают минимальное и максимальное значение в группе соответственно. При работе с ними применяются те же правила группировки, что и везде, поэтому выходные временные ряды будут иметь метки группы². Например,

```
max without(device, fstype, mountpoint)(node_filesystem_size_bytes)
```

вернет размер наибольшей файловой системы в каждом экземпляре:

```
{instance="localhost:9100",job="node"} 90131324928
```

Агрегаторы `max` и `min` будут возвращать `NaN`, только если все значения в группе равны `NaN`³.

¹ Если бы оператор возведения в степень существовал в то время, когда мы добавляли `stdvar` и `stddev`, то оператор `stdvar`, вероятно, не был бы добавлен.

² Если нужно, чтобы возвращались входные временные ряды, используйте `topk` или `bottomk`.

³ В арифметике с плавающей точкой любое сравнение с `NaN` всегда возвращает ложное значение. Помимо некоторых странностей, таких как `NaN != NaN`, наивная реализация `min` и `max` застряла бы (и когда-то застревала) на `NaN`, если бы это было первое проверяемое значение.

topk и bottomk

`topk` и `bottomk` имеют три важных отличия от других агрегаторов, обсуждавшихся до сих пор. Во-первых, метки временных рядов, которые они возвращают для группы, не являются метками группы; во-вторых, для каждой группы они могут возвращать несколько временных рядов; и в-третьих, они принимают дополнительный параметр.

`topk` возвращает к временных рядов с наибольшими значениями, например

```
topk without(device, fstype, mountpoint)(2, node_filesystem_size_bytes)
```

вернет до двух¹ временных рядов для каждой группы:

```
node_filesystem_size_bytes{device="/dev/sda5",fstype="ext4",
  instance="localhost:9100",job="node",mountpoint="/" } 90131324928
node_filesystem_size_bytes{device="tmpfs",fstype="tmpfs",
  instance="localhost:9100",job="node",mountpoint="/run" } 826961920
```

Как видите, `topk` возвращает входные временные ряды со всеми их метками, включая метку `__name__`, которая содержит имя метрики. Результаты дополнительно сортируются.

`bottomk` действует так же, как `topk`, но возвращает к временных рядов с наименьшими значениями. Оба агрегатора стараются избегать возврата временных рядов со значениями NaN.

При использовании этих агрегаторов с конечной точкой HTTP API `query_range` необходимо учитывать один нюанс. Как обсуждалось в разделе «`query_range`» главы 13, вычисления на каждом шаге выполняются независимо. Поэтому может случиться так, что при использовании `topk(5, some_gauge)` для `query_range` с 1000 шагов может в худшем случае вернуть 5000 различных временных рядов.

Чтобы избежать этого, используйте модификатор `@`, как обсуждалось в разделе «Модификатор `@`» главы 13.

quantile

Агрегатор `quantile` возвращает указанный квантиль значений группы. Подобно `topk`, оператор `quantile` принимает дополнительный параметр.

Например, с помощью следующего запроса можно узнать 90-й процентиль потребления процессора в режиме `system` на каждой из наших машин:

```
quantile without(cpu)(0.9, rate(node_cpu_seconds_total{mode="system"}[5m]))
```

В нашем тестовом окружении он вернул

```
{instance="localhost:9100",job="node",mode="system"} 0.024558620689654007
```

¹ В данном случае k равно 2.

Этот результат сообщает, что 90 % наших процессоров проводят в режиме system не менее 0.02 с за секунду. Этот запрос был бы намного полезнее, имей мы десятки процессоров, а не четыре, как на самом деле.

В дополнение к среднему с помощью `quantile` можно получить медиану, 25-й и 75-й процентили¹. Например, вот запросы, сообщающие потребление процессорного времени процессами:

```
# арифметическое среднее
avg without(instance)(rate(process_cpu_seconds_total[5m]))

# 0.25-й квантиль, 25-й процентиль, 1-й или нижний quartиль
quantile without(instance)(0.25, rate(process_cpu_seconds_total[5m]))

# 0.5-й квантиль, 50-й процентиль, 2-й quartиль, медиана
quantile without(instance)(0.5, rate(process_cpu_seconds_total[5m]))

# 0.75-й квантиль, 75-й процентиль, 3-й или верхний quartиль
quantile without(instance)(0.75, rate(process_cpu_seconds_total[5m]))
```

Эти значения дадут вам представление о поведении разных экземпляров при выполнении задания без необходимости графически отображать каждый экземпляр отдельно. Это поможет сохранить читаемость дашбордов с ростом числа экземпляров. На своем опыте мы убедились, что при просмотре графиков для отдельных экземпляров в дашбордах анализируются не более трех-пяти экземпляров.

quantile, histogram_quantile и quantile_over_time

Как вы, возможно, уже заметили, в PromQL существует несколько функций и операторов со словом «`quantile`» в имени.

Агрегатор `quantile` работает с мгновенным вектором, представляющим агрегированную группу.

Функция `quantile_over_time` работает с одним временным рядом за раз векторе диапазонов.

Функция `histogram_quantile` работает с сегментами одного дочернего элемента метрики-гистограммы в мгновенном векторе.

count_values

Последний оператор агрегирования – `count_values`. Подобно `topk`, он принимает дополнительный параметр и может вернуть несколько временных рядов из группы. Он строит гистограмму частот значений временного ряда в группе со счетчиком каждого значения в качестве значения выходного временного ряда и исходным значением в качестве новой метки.

¹ Также известны как 1-й и 3-й квартили.

Это описание выглядит немного запутанным, поэтому давайте рассмотрим пример. Допустим, у вас есть временной ряд `software_version` со следующими значениями:

```
software_version{instance="a",job="j"} 7
software_version{instance="b",job="j"} 4
software_version{instance="c",job="j"} 8
software_version{instance="d",job="j"} 4
software_version{instance="e",job="j"} 7
software_version{instance="f",job="j"} 4
```

Применив к этомуциальному ряду запрос

```
count_values without(instance)(“version”, software_version)
```

вы получите такой результат:

```
{job="j",version="7"} 2
{job="j",version="8"} 1
{job="j",version="4"} 3
```

В группе есть два временных ряда со значением 7, поэтому для временного ряда с `version="7"` и метками группы было возвращено значение 2. Аналогичный результат возвращается для других временных рядов.

При создании частотной гистограммы группировка не используется; используются точные значения временного ряда. Поэтому этот прием по-настоящему полезен только с целочисленными значениями и там, где не слишком много уникальных значений.

В частности, его с успехом можно использовать для проверки номеров версий¹ или количества объектов определенного типа, которые видят каждый экземпляр вашего приложения. Если у вас одновременно развернуто слишком много версий или разные приложения продолжают видеть разное количество объектов, то, возможно, что-то где-то не может продвинуться вперед.

`count_values` можно комбинировать с `count` для вычисления количества уникальных значений в данной группе. Например, вот как можно рассчитать количество развернутых версий программного обеспечения:

```
count without(version)(
  count_values without(instance)(“version”, software_version)
)
```

В нашем случае это выражение вернуло

```
{job="j"} 3
```

Также, комбинируя `count_values` и `count`, можно подсчитать, сколько ваших машин имеет то или иное количество дисковых устройств:

¹ Для версий, которые нельзя представить в виде значений с плавающей точкой, можно использовать информационную метрику, как описано в разделе «Информационные метрики» главы 5.

```
count_values without(instance)(  
    "devices",  
    count without(device) (node_disk_io_now)  
)
```

В нашем случае мы получили

```
{devices="5",job="node"} 1
```

Теперь, познакомившись с агрегаторами, перейдем к знакомству двухместных операторов, таких как сложение и вычитание, и посмотрим, как работает сопоставление векторов.

Глава 15

Двухместные операторы

Часто с метриками требуется сделать нечто большее, чем простое агрегирование, и именно здесь на помощь вам придут *двуихместные операторы*. Двухместные операторы – это операторы, принимающие два операнда¹, такие как операторы сложения и сравнения.

Двухместные операторы позволяют выполнять не только простые арифметические действия с мгновенными векторами, но также могут применяться к двум мгновенным векторам с группировкой на основе меток. Именно здесь проявляется настоящая мощь языка PromQL – возможность выполнять виды анализа, предлагаемые лишь немногими другими системами мониторинга.

PromQL имеет три набора двухместных операторов: арифметические операторы, операторы сравнения и логические операторы. Эта глава покажет вам, как их использовать.

Работа со скалярами

В дополнение к мгновенным векторам и векторам диапазонов существует еще один тип значений, известный как *скаляр*². Скаляры – это одиночные числа без размерности. Например, `0` – это скаляр с нулевым значением, а `[] 0` – мгновенный вектор, содержащий один образец с нулевым значением и без меток³.

¹ В отличие от одноместных (унарных) операторов, которые принимают только один операнд. В PromQL есть также унарные операторы `+` и `-`.

² В PromQL также имеется строковый тип, но он используется только для аргумента `count_values`, `label_replace` и `label_join`.

³ Вам также может встретиться соглашение `{}: 0` для представления одного образца.

Арифметические операторы

Скаляры можно использовать в арифметических операциях с мгновенным вектором, чтобы изменить значения в нем. Например,

```
process_resident_memory_bytes / 1024
```

вернет значения:

```
{instance="localhost:9090",job="prometheus"} 21376
{instance="localhost:9100",job="node"} 13316
```

описывающие потребление процессами памяти в килобайтах¹. Обратите внимание, что оператор деления применяется ко всем временным рядам в мгновенном векторе, возвращаемом селектором `process_resident_memory_bytes`, а имя метрики было удалено, так как новое значение больше не является метрикой `process_resident_memory_bytes`.

- i** Даже если арифметические операторы используются так, что не меняют значения, имя метрики все равно удаляется для согласованности. Например, результат `some_gauge + 0` не будет иметь имени метрики.

Все шесть арифметических операций работают одинаково, с семантикой, обычно ожидаемой от других языков программирования:

- + – сложение;
- – вычитание;
- * – умножение;
- / – деление;
- % – деление по модулю (остаток от деления);
- ^ – возведение в степень.

Оператор *деления по модулю* возвращает результат с плавающей точкой и может вернуть нецелочисленный результат, если ему передать нецелочисленные входные данные. Например,

```
5 % 1.5
```

вернет

```
0.5
```

Как показывает этот пример, двухместные арифметические операторы можно также применять к операндам-скалярам. Результатом в этом случае будет скаляр. Это полезно для придания удобочитаемости, так как гораздо легче понять назначение $(1024 * 1024 * 1024)$, чем 1073741824 .

¹ Если вы используете инструмент создания дашбордов, такой как Grafana, то обычно лучше позволить ему создавать удобочитаемые единицы измерения для метрик, которые уже представлены в базовых единицах, таких как байты.

Кроме того, скалярный операнд можно поместить слева от оператора, а мгновенный вектор справа, например:

```
1e9 - process_resident_memory_bytes
```

Это выражение вычтет из миллиарда объем памяти, использованной процессом.

Арифметические операторы также можно применять к двум мгновенным векторам, как описано в разделе «Сопоставление векторов» ниже.

Тригонометрический оператор

Оператор `atan2` возвращает арктангенс частного двух векторов, используя их знаки для определения квадранта возвращаемого значения:

```
x atan2 y
```

Этот оператор позволяет применить функцию `atan2` к двум векторам, используя сопоставление векторов, что недоступно в обычных функциях. Он действует так же, как арифметические операторы (+, -, *, ...).

Операторы сравнения

В PromQL поддерживаются следующие *операторы сравнения*:

- == – равно;
- != – не равно;
- > – больше;
- < – меньше;
- >= – больше или равно;
- <= – меньше или равно.

Отличительная особенность операторов сравнения в PromQL – они являются *фильтрами*, т. е. если у вас есть образцы

```
process_open_fds{instance="localhost:9090",job="prometheus"} 14  
process_open_fds{instance="localhost:9100",job="node"} 7
```

и выполняется сравнение мгновенного вектора со скаляром

```
process_open_fds > 10
```

то вы получите результат:

```
process_open_fds{instance="localhost:9090",job="prometheus"} 14
```

Поскольку операторы сравнения не изменяют значений, имя метрики сохраняется. При сравнении скаляра и мгновенного вектора, независимо от того, с какой стороны находится скаляр, всегда возвращаются элементы мгновенного вектора.

i Поскольку PromQL поддерживает числа с плавающей точкой, использование операторов `==` и `!=` требует некоторой осторожности. Вычисления с плавающей точкой могут давать немного разные результаты, в зависимости от того, в каком порядке выполняются операции.

Сравнивая нецелочисленные значения, лучше сравнивать их разность с некоторым небольшим числом, которое называется *эпсилон*. Например, вот как можно сравнить датчик со значением 1 с погрешностью в одну миллионную:

```
(some_gauge - 1) < 1e-6 > -1e-6
```

Выполнить фильтрующее сравнение между двумя скалярами невозможно, потому что для согласования с арифметическими операциями между двумя скалярами оно должно было бы возвращать скаляр. Однако в результате фильтрации не может получиться пустой скаляр, в отличие от пустого мгновенного вектора.

Модификатор `bool`

Фильтрующее сравнение в первую очередь полезно в правилах формирования предупреждений, как обсуждается в главе 18, и в общем случае его следует избегать в других местах¹. Мы покажем вам почему.

Продолжая предыдущий пример, допустим, что мы хотим узнать, сколько процессов для каждого задания имеет более 10 дескрипторов открытых файлов. Очевидный способ сделать это:

```
count without(instance)(process_open_fds > 10)
```

который может вернуть

```
{job="prometheus"} 1
```

Этот запрос правильно сообщает, что существует один процесс Prometheus с более чем 10 дескрипторами открытых файлов. Но он не сообщает, что Node Exporter не имеет таких процессов. Это очень тонкий нюанс, потому что, пока один временной ряд не отфильтрован, все кажется в порядке.

На самом деле нам нужен способ выполнить сравнение без фильтрации. Именно такую возможность дает модификатор `bool`; для каждого сравнения он возвращает 0 (обозначающий `false`) или 1 (обозначающий `true`).

Например,

```
process_open_fds > bool 10
```

вернет результат

```
{instance="localhost:9090",job="prometheus"} 1
{instance="localhost:9100",job="node"} 0
```

который, как и ожидалось, имеет по одному экземпляру для каждого входного мгновенного вектора.

¹ Фильтрацию можно использовать при осторожном применении оператора `of`, но это сложно и чревато ошибками.

Полученные результаты можно суммировать, чтобы получить количество процессов для каждого задания, которые имеют более 10 дескрипторов открытых файлов:

```
sum without(instance)(process_open_fds > bool 10)
```

и получить результат, который, собственно, и был нужен:

```
{job="prometheus"} 1  
{job="node"} 0
```

Аналогичный подход можно использовать для поиска доли машин с пятью дисковыми устройствами и более:

```
avg without(instance)(  
    count without(device)(node_disk_io_now) > bool 4  
)
```

Это выражение сначала выполнит оператор агрегирования `count`, чтобы получить количество дисков, возвращаемое каждым экспортёром Node Exporter, затем это количество сравнивается с числом 4, и, наконец, результат усредняется по количеству машин, чтобы получить долю. Хитрость здесь в том, что все значения, возвращаемые модификатором `bool`, равны 0 и 1, поэтому `count` возвращает общее количество машин, а `sum` – это количество машин, удовлетворяющих критериям. `avg` – это количество, деленное на сумму, что дает отношение, или пропорцию.

Модификатор `bool` – единственный способ сравнить скаляры, например

```
42 <= bool 13
```

вернет

```
0
```

где 0 означает `false`.

Сопоставление векторов

Применение операторов к скалярам и мгновенным векторам охватывает многие наши потребности, но в применении операторов к двум мгновенным векторам раскрывается истинная мощь PromQL.

Имея скаляр и мгновенный вектор, мы можем применить скаляр к каждому образцу в векторе. Но при применении к двум мгновенным векторам какие образцы из векторов применяются друг к другу? Ответ на этот вопрос дает *сопоставление векторов*.

Один к одному

В простейших случаях между двумя векторами имеется взаимно однозначное соответствие. Пусть у нас есть следующие образцы:

```
process_open_fds{instance="localhost:9090",job="prometheus"} 14
process_open_fds{instance="localhost:9100",job="node"} 7
process_max_fds{instance="localhost:9090",job="prometheus"} 1024
process_max_fds{instance="localhost:9100",job="node"} 1024
```

Тогда выражение

```
/process_open_fds
/
process_max_fds
```

вернет результат:

```
{instance="localhost:9090",job="prometheus"} 0.013671875
{instance="localhost:9100",job="node"} 0.0068359375
```

Здесь образцы с одинаковыми метками, за исключением имени метрики в метке `__name__`, были поставлены в соответствие друг другу, т. е. были сопоставлены два образца с метками `{instance="localhost:9090",job="prometheus"}` и два образца с метками `{instance="localhost:9100",job="node"}`.

В этом случае имело место идеальное совпадение, когда совпали все образцы в векторах с обеих сторон от оператора. Если бы для образца с одной стороны не нашлось соответствия с другой стороны, то он не попал бы в результат, так как двухместным операторам требуется два операнда.



Если вопреки ожиданиям двухместный оператор возвращает пустой мгновенный вектор, то, вероятно, это обусловлено несовпадением меток образцов в операндах. Часто это происходит из-за того, что на одной стороне оператора имеется метка, а на другой нет.

Иногда может понадобиться сопоставить два мгновенных вектора, метки которых не совпадают точно. Подобно тому, как агрегирование позволяет указать, какие метки имеют значение (см. раздел «Группировка» главы 14), сопоставление векторов также предлагает возможность сообщить, какие метки следует учитывать.

Вы можете использовать модификатор `ignoring`, чтобы игнорировать определенные метки, как это делает модификатор `without` при агрегировании. Допустим, вы используете метрику `node_cpu_seconds_total`, которая имеет инструментированные метки `cpu` и `mode`, и хотели бы узнать, какую часть времени тратит каждый экземпляр в режиме `idle`. Для этого можно использовать такое выражение:

```
sum without(cpu)(rate(node_cpu_seconds_total{mode="idle"}[5m]))
/
ignoring(mode)
sum without(mode, cpu)(rate(node_cpu_seconds_total[5m]))
```

Оно вернет такой результат:

```
{instance="localhost:9100",job="node"} 0.8423353718871361
```

Здесь первый вызов `sum` возвращает мгновенный вектор с меткой `mode="idle"`, а второй – мгновенный вектор без метки `mode`. Обычно в таких случаях механизм сопоставления векторов терпит неудачу, но в этом примере оператор `ignoring(mode)` отбрасывает метку `mode`, и сопоставление выполняется успешно. Поскольку метка `mode` отсутствовала в сопоставляемых группах, она также отсутствует и в результате¹.

 Даже ничего не зная о лежащих в основе временных рядах, мы можем смело сказать, что предыдущее выражение верное с точки зрения сопоставления векторов. Метка `cri` удаляется с обеих сторон, и `ignoring(mode)` обрабатывает одну сторону, имеющую метку `mode`, а другую – нет.

Ситуация может осложниться, когда в сопоставлении участвуют разные временные ряды с разными метками, но просмотр выражений с точки зрения распределения меток поможет обнаружить ошибки.

Оператор `on` позволяет учесть только определенные метки. Он действует подобно оператору `by` в операциях агрегирования. Выражение

```
sum by(instance, job)(rate(node_cpu_seconds_total{mode="idle"}[5m]))  
/ on(instance, job)  
sum by(instance, job)(rate(node_cpu_seconds_total[5m]))
```

вернет тот же результат, что и предыдущее выражение², но, как и в случае с `by`, оператор `on` имеет недостаток – вы должны заранее знать все метки, имеющиеся в данный момент во временном ряду или способные появиться в будущем в других контекстах.

Значение, возвращаемое арифметическими операторами, является результатом вычислений, но вам может быть интересно, что происходит с операторами сравнения, когда в операции участвуют два мгновенных вектора. При сравнении двух мгновенных векторов возвращаются образцы из вектора слева. Например, выражение

```
process_open_fds  
>  
(process_max_fds * .5)
```

вернет экземпляры из `process_open_fds`, в которых количество дескрипторов открытых файлов превышает половину от максимума³.

¹ Метка `cri` была агрегирована обоими вызовами `sum`, поэтому она тоже отсутствует в результате.

² Здесь можно исключить `on(instance, job)`, так как обе стороны, левая и правая, имеют только метки `instance` и `job`.

³ Исчерпание файловых дескрипторов может привести к аварийному завершению приложений, поэтому желательно, чтобы в распоряжении ваших приложений всегда было достаточно дескрипторов.

Если вместо этого использовать выражение

```
(process_max_fds * .5)
<
process_open_fds
```

то в результате вы получите половину максимального количества файловых дескрипторов. Результат будет иметь те же метки, но само это значение может быть менее полезным семантически для рассылки уведомлений¹ или при использовании в дашборде! В общем случае текущие значения более информативны, чем пределы, поэтому старайтесь структурировать вычисления так, чтобы интересующие вас значения находились слева от оператора сравнения.

Многие к одному и group_left

Если удалить сопоставитель mode из первой половины выражения в предыдущем разделе и попытаться вычислить

```
sum without(cpu)(rate(node_cpu_seconds_total[5m]))
/ ignoring(mode)
  sum without(mode, cpu)(rate(node_cpu_seconds_total[5m]))
```

то вы получите сообщение об ошибке:

```
multiple matches for labels:
many-to-one matching must be explicit (group_left/group_right)
```

Это связано с тем, что теперь между образцами нет прямого соответствия – каждому образцу справа соответствует несколько образцов с разными метками mode слева. Эта ошибка может проявляться не сразу, так как временной ряд, нарушающий работу такого выражения, может появиться позже. Заметить эту потенциальную проблему можно, если обратить внимание, что в потоке меток слева нет ничего, что ограничивало бы метку mode одним потенциальным значением².

Подобные ошибки обычно возникают из-за неправильно составленных выражений, поэтому PromQL по умолчанию не пытается их исправлять. Вместо этого вы должны прямо указать, что должно выполняться сопоставление «многие к одному», использовав модификатор `group_left`.

`group_left` сообщает движку PromQL, что в группе, возвращаемой левым операндом, может иметься несколько образцов, соответствующих одному образцу в группе справа³. Например,

¹ Шаблоны предупреждений имеют доступ к значению выражения PromQL. Об этом говорится в разделе «Аннотации и шаблоны» главы 18.

² Операция агрегирования, удаляющая метку `mode`, на самом деле оставляет эту метку с единственным значением, равным пустой строке.

³ В группе справа по-прежнему может быть только один образец, соответствующий нескольким образцам слева, потому что `group_left` разрешает только сопоставление «многие к одному», но не «многие ко многим».

```
sum without(cpu)(rate(node_cpu_seconds_total[5m]))
/ ignoring(mode) group_left
  sum without(mode, cpu)(rate(node_cpu_seconds_total[5m]))
```

создаст один выходной образец для каждой метки mode в каждой группе слева:

```
{instance="localhost:9100",job="node",mode="irq"} 0
{instance="localhost:9100",job="node",mode="nice"} 0
{instance="localhost:9100",job="node",mode="softirq"} 0.00005226389784152013
{instance="localhost:9100",job="node",mode="steal"} 0
{instance="localhost:9100",job="node",mode="system"} 0.01720353303949279
{instance="localhost:9100",job="node",mode="user"} 0.10345203045243238
{instance="localhost:9100",job="node",mode="idle"} 0.8608691486211044
{instance="localhost:9100",job="node",mode="iowait"} 0.01842302398912871
```

group_left всегда берет все метки из образцов в операнде слева. Это гарантирует сохранение дополнительных меток слева стороны, требующих сопоставления «многие к одному»¹.

Это намного проще, чем составлять выражение с сопоставлением «один к одному», предусматривающим все возможные метки mode: модификатор group_left сделает все это за вас в одном выражении. Этот подход можно использовать для определения доли, которую представляет каждое значение метки в метрике, как показано в предыдущем примере, или для сравнения метрики, полученной из лидера кластера, с метриками из реплик.

Существует еще одно применение group_left – добавление меток из информационных метрик в целевые метрики. Порядок инструментирования информационных метрик описан в разделе «Информационные метрики» главы 5. Задача информационных метрик состоит в том, чтобы добавить метки, которые могут пригодиться в других метриках, но вызвали бы их чрезмерное загромождение, если бы использовались как обычные метки.

Метрика prometheus_build_info, например, предоставляет информацию о сборке Prometheus:

```
prometheus_build_info{branch="HEAD",governsion="go1.10",
  instance="localhost:9090",job="prometheus",
  revision="bc6058c81272a8d938c05e75607371284236adc",version="2.2.1"}
```

Ее можно объединить с такими метриками, как up:

```
up
* on(instance) group_left(version)
  prometheus_build_info
```

что приведет к такому результату:

```
{instance="localhost:9090",job="prometheus",version="2.2.1"} 1
```

Как видите, метка version была скопирована из правого операнда в левый в соответствии с модификатором group_left(version). При желании в group_

¹ Если бы использовались метки с правой стороны, то вы бы получили одинаковые метки для каждого образца из групп слева, что привело бы к конфликту.

`left` можно указать любое количество меток, но обычно ограничиваются одной или двумя¹. Этот подход работает независимо от количества инструментированных меток слева, потому что сопоставление векторов производится по принципу «многие к одному».

Предыдущее выражение с модификатором `on(instance)` опирается на тот факт, что каждая метка `instance` используется только для одной цели. Часто это действительно так, но не всегда, поэтому также может понадобиться добавить в модификатор `on` другие метки, такие как `job`.

`prometheus_build_info` применяется ко всей цели. Но есть также информационные метрики², такие как `node_hwmon_sensor_label`, упомянутые в разделе «Сборщик информации об аппаратной части» главы 7, которые применяются к дочерним элементам других метрик:

```
node_hwmon_sensor_label{chip="platform_coretemp_0",instance="localhost:9100",
    job="node",label="core_0",sensor="temp2"} 1
node_hwmon_sensor_label{chip="platform_coretemp_0",instance="localhost:9100",
    job="node",label="core_1",sensor="temp3"} 1

node_hwmon_temp_celsius{chip="platform_coretemp_0",instance="localhost:9100",
    job="node",sensor="temp1"} 42
node_hwmon_temp_celsius{chip="platform_coretemp_0",instance="localhost:9100",
    job="node",sensor="temp2"} 42
node_hwmon_temp_celsius{chip="platform_coretemp_0",instance="localhost:9100",
    job="node",sensor="temp3"} 41
```

У метрики `node_hwmon_sensor_label` есть дочерние элементы, соответствующие некоторым (но не всем) времененным рядам в `node_hwmon_temp_celsius`. В данном случае мы знаем, что есть только одна дополнительная метка (с именем `label`), поэтому мы можем использовать `ignoring` с `group_left`, чтобы добавить эту метку в образцы `node_hwmon_temp_celsius`:

```
node_hwmon_temp_celsius
* ignoring(label) group_left(label)
  node_hwmon_sensor_label
```

и получить в результате:

```
{chip="platform_coretemp_0",instance="localhost:9100",
    job="node",label="core_0",sensor="temp2"} 42
{chip="platform_coretemp_0",instance="localhost:9100",
    job="node",label="core_1",sensor="temp3"} 41
```

Обратите внимание, что здесь нет образца с `sensor="temp1"`, потому что такого образца не было в `node_hwmon_sensor_label` (сопоставление разреженных мгновенных векторов мы рассмотрим в разделе «Оператор `or`» ниже).

¹ Нет никакой возможности запросить копирование всех меток, так как тогда вы не будете знать, какие метки имеет выходная метрика.

² Соглашение о том, должна ли информационная метрика с единственной меткой иметь суффикс `_info`, еще не стало общепринятым.

Существует также модификатор `group_right`, действующий подобно `group_left`, за исключением того, что стороны «один» и «многие» меняются местами, т. е. сторона «многие» теперь является правым операндом. Любые метки, указанные в модификаторе `group_right`, копируются слева направо. Однако ради согласованности мы рекомендуем использовать только `group_left`.

Многие ко многим и логические операторы

Существует три логических оператора, которые также называют операторами множеств:

- `or` – объединение;
- `and` – пересечение;
- `unless` – разность.

Здесь нет оператора `not`, но его роль играет функция `absent`, описанная в разделе «Отсутствующие серии, `absent` и `absent_over_time`» главы 16.

Все логические операторы работают по принципу «многие ко многим», и они – единственные операторы, работающие по этому принципу. В отличие от арифметических операторов и операторов сравнения логические операторы не выполняют никаких математических операций; значение имеет лишь присутствие образца в группе.

Оператор `or`

В предыдущем разделе в метрике `node_hwmon_sensor_label` образцы присутствовали не для всех значений `node_hwmon_temp_celsius`, поэтому результаты возвращались только для образцов, имеющихся в обоих мгновенных векторах. Метрики с несогласованными или отсутствующими дочерними элементами сложны в работе, но эту проблему легко решить с помощью оператора `or`.

Оператор `or` возвращает образцы из каждой группы слева, если они есть, в противном случае он возвращает образцы из группы справа. Если вы знакомы с SQL, то можете считать этот оператор подобием SQL-функции `COALESCE`, только применяется она к меткам, а не к данным.

Продолжим пример из предыдущего раздела и используем `or` для замены временных рядов, отсутствующих в `node_hwmon_sensor_label`. Для этого нужен какой-то другой временной ряд с требуемыми метками, в данном случае это `node_hwmon_temp_celsius`. Метрика `node_hwmon_temp_celsius` не имеет метки `label`, но все остальные метки совпадают, поэтому просто игнорируем эту метку с помощью `ignoring`:

```
node_hwmon_sensor_label
or ignoring(label)
  (node_hwmon_temp_celsius * 0 + 1)
```

Сопоставление векторов дало три группы меток. Первые две группы имеют образцы из `node_hwmon_sensor_label`, поэтому будут возвращены именно они, включая имя метрики. Однако в третьей группе, куда входит `sensor="temp1"`, имеется образец, для которого не нашлось соответствия в левой части, по-

этому будут взяты значения из группы справа. А так как к значению применяются арифметические операторы, имя метрики будет удалено.

- ✓ $x * 0 + 1$ заменит все¹ значения мгновенного вектора x на 1. Этот прием также может пригодиться при использовании `group_left` для копирования меток, так как 1 – это значение идентичности для операции умножения – оно не изменяет значения, которое вы умножаете на 1.

Теперь это выражение можно использовать вместо `node_hwmon_sensor_label`:

```
node_hwmon_temp_celsius
* ignoring(label) group_left(label)
(
    node_hwmon_sensor_label
    or ignoring(label)
    (node_hwmon_temp_celsius * 0 + 1)
)
```

Оно вернет

```
{chip="platform_coretemp_0",instance="localhost:9100",
 job="node",sensor="temp1"} 42
{chip="platform_coretemp_0",instance="localhost:9100",
 job="node",label="core_0",sensor="temp2"} 42
{chip="platform_coretemp_0",instance="localhost:9100",
 job="node",label="core_1",sensor="temp3"} 41
```

Образец с `sensor="temp1"` теперь присутствует в полученном результате. У него нет метки с именем `label`, что равносильно наличию метки `label` с пустой строкой в качестве значения.

В более простых случаях вы будете работать с метриками без каких-либо инструментированных меток. Например, вы можете использовать сборщик текстовых файлов, описанный в разделе «Сборщик текстовых файлов» главы 7, и ожидать, что он вернет метрику с именем `node_custom_metric`, а в отсутствие такой метрики вы хотели бы возвращать 0. В таких случаях можно использовать метрику `up`, связанную с каждой целью:

```
node_custom_metric
or
up * 0
```

Этот прием имеет небольшую проблему – он будет возвращать значение даже если попытка опроса потерпела неудачу, хотя извлекаемые метрики действуют иначе². Он также будет возвращать результаты для других задачий. Исправить проблему можно, добавив сопоставление с фильтрацией:

¹ Значение `Nan` останется значением `Nan`, но на практике будет получен другой временной ряд с теми же метками и без значений `Nan`, которые вы могли бы использовать.

² `up` – это не извлекаемая метрика; Prometheus добавляет ее после каждого цикла опроса независимо от успеха или неудачи операции.

```
node_custom_metric  
ор  
(up{job="node"} == 1) * 0
```

Другой способ использования оператора `ор` – возврат наибольшего из двух рядов:

```
(a >= b) or b
```

Если `a` больше, оно будет возвращено оператором сравнения, а затем и оператором `ор`, потому что группа слева – непустая. Если, напротив, `b` больше, то оператор сравнения ничего не вернет и `ор` вернет `b`, так как группа слева – пустая.

Оператор unless

Оператор `unless` выполняет сопоставление векторов так же, как оператор `ор`, принимая решение в зависимости от того, являются ли группы справа или слева пустыми. Оператор `unless` возвращает группу слева, только если группа справа не имеет образцов, иначе возвращается группа справа.

Вы можете использовать `unless` для выбора возвращаемых временных рядов на основе выражений. Например, чтобы узнать среднее потребление процессорного времени процессами, за исключением тех, которые используют менее 100 Мбайт оперативной памяти, можно использовать такое выражение:

```
rate(process_cpu_seconds_total[5m])  
unless  
process_resident_memory_bytes < 100 * 1024 * 1024
```

`unless` можно также использовать, чтобы определить присутствие метрики в цели. Например,

```
up{job="node"} == 1  
unless  
node_custom_metric
```

вернет образец для каждого экземпляра, в котором отсутствует метрика `node_custom_metric`. Подобные выражения можно использовать для рассылки оповещений.

По умолчанию, подобно всем двухместным операторам, `unless` просматривает все метки при группировке. Если `node_custom_metric` будет иметь инструментированные метки, то вы сможете использовать `ор` или `ignoring` для проверки существования хотя бы одного соответствующего временного ряда, даже не зная значений других меток:

```
up == 1  
unless on (job, instance)  
node_custom_metric
```

Даже если в группе окажется несколько образцов из правого операнда, это нормально, так как `unless` использует сопоставление «многие ко многим».

Оператор and

Оператор `and` противоположен оператору `unless`. Он возвращает группу из левого операнда, только если соответствующая группа справа имеет образцы, в противном случае он не возвращает образцов для этой группы. Его можно рассматривать как оператор `if1`.

Чаще всего оператор `and` используется, чтобы предупредить о выполнении сразу нескольких условий. Например, вы можете захотеть вернуться, когда задержка слишком высока и имеется больше, чем несколько пользовательских запросов. Вот как это можно реализовать при использовании обработчиков Prometheus, на выполнение которых уходит одна секунда и они получают не менее одного запроса каждую секунду:

```
(  
    rate(http_request_duration_seconds_sum{job="prometheus"}[5m])  
    /  
    rate(http_request_duration_seconds_count{job="prometheus"}[5m])  
) > 1  
and  
rate(http_request_duration_seconds_count{job="prometheus"}[5m]) > 1
```

Это выражение вернет образец для каждого отдельного обработчика в каждом задании `rgometheus`, что может вызвать некоторое замедление даже с ограничением один запрос в секунду. Обычно при рассылке уведомлений бывает желательно агрегировать данные по заданию.

Вместе с оператором `and`, как и с другими двухместными операторами, можно использовать модификаторы `on` и `ignoring`. В частности, с помощью `on()` можно выявлять случаи, когда два операнда вообще не имеют общих меток. Этот прием можно использовать, например, чтобы ограничить время дня, для которого выражение будет возвращать результаты:

```
(  
    rate(http_request_duration_microseconds_sum{job="prometheus"}[5m])  
    /  
    rate(http_request_duration_microseconds_count{job="prometheus"}[5m])  
) > 1000000  
and  
rate(http_request_duration_microseconds_count{job="prometheus"}[5m]) > 1  
and on()  
hour() >= 9 < 17
```

Функция `hour` описана в разделе «Функции `minute`, `hour`, `day_of_week`, `day_of_month`, `day_of_year`, `days_in_month`, `month` и `year`» главы 16; она возвращает мгновенный вектор с одним образцом без меток и часом дня UTC в момент выполнения запроса.

¹ До выхода версии Prometheus 2.x в PromQL имелось ключевое слово `IF`, которое использовалось для формирования оповещений, поэтому, хотя Брайан и думал, что было бы неплохо переименовать оператор `and` в `if`, осуществить это на практике было невозможно.

Приоритет операторов

При оценке выражений с несколькими двухместными операторами PromQL не просто выполняет их слева направо, но учитывает их приоритеты, которые во многом совпадают с приоритетами аналогичных операторов в языках программирования:

- 1) $^$;
- 2) $* / \% \text{atan}2$;
- 3) $+ -$;
- 4) $== != > < >= <=$;
- 5) `unless and`;
- 6) `or`.

Например, $a \text{ or } b * c + d$ – это то же самое, что $a \text{ or } ((b * c) + d)$.

Все операторы, кроме $^$, левоассоциативны. Это означает, что $a/b*c$ – это то же самое, что $(a/b)*c$, но a^b*c – это a^b*c .

Изменить порядок вычисления операторов можно с помощью круглых скобок. Мы также рекомендуем добавлять круглые скобки, если порядок вычисления выражения может быть неочевиден, так как не все помнят приоритеты операторов.

Теперь, познакомившись с агрегаторами и операторами, рассмотрим заключительную часть PromQL: функции.

Глава 16

Функции

В версии PromQL 2.37.0 имеется 69 функций, в число которых входят как обычные математические функции, так и функции, специально предназначенные для работы со счетчиками и гистограммами. В этой главе мы расскажем, как работают все эти функции и как их можно использовать.

Почти все функции PromQL возвращают мгновенные векторы, и только три (`time`, `pi` и `scalar`) возвращают скаляры. В PromQL нет ни одной функции, возвращающей векторы диапазонов, хотя многие, включая `rate` и `avg_over_time`, принимают векторы диапазонов в виде аргументов.

Иначе говоря, функции обычно работают либо с образцами одного временного ряда, либо с образцами мгновенного вектора. Чтобы обработать весь вектор диапазона сразу, вам придется использовать подзапросы.

PromQL – статически типизированный язык, и его функции не меняют свои возвращаемые значения в зависимости от типов входных данных. На самом деле типы входных аргументов каждой функции также фиксированы. Если функция должна обрабатывать значения двух разных типов, то определяются две функции с разными именами. Например, агрегатор `avg` применяется к мгновенным векторам, а `avg_over_time` – к векторам диапазонов.

Функции не делятся официально на какие-либо категории, но мы группировали родственные функции вместе.

Изменение типа

Иногда вам требуется получить скаляр из вектора или наоборот. Сделать это вам помогут две функции: `vector` и `scalar`.

`vector`

Функция `vector` принимает скалярное значение и преобразует его в мгновенный вектор с одним образцом с заданным значением и без метки. Например, выражение,

```
vector(1)
```

вернет

```
{} 1
```

Она может пригодиться, например, чтобы гарантировать возврат результата независимо от наличия какого-либо конкретного временного ряда. Например,

```
sum(some_gauge) or vector(0)
```

всегда будет возвращать один образец, даже если в `some_gauge` отсутствуют образцы. В зависимости от варианта использования иногда вместо оператора `or` (раздел «Оператор `or`» главы 15) предпочтительнее использовать модификатор `bool`, описанный в разделе «Модификатор `bool`» главы 15.

scalar

Функция `scalar` принимает мгновенный вектор с одним образцом и преобразует его в скаляр со значением входного образца. Если вектор содержит несколько образцов или не содержит ни одного, то `scalar` вернет `NaN`.

Эта функция может пригодиться при работе со скалярными константами, когда требуется использовать математические функции, работающие только с мгновенными векторами. Например, если вам понадобится получить натуральный логарифм двойки в виде скаляра, то вместо того, чтобы набирать `0.6931471805599453` и надеяться, что любой, кто прочитает это значение, поймет его смысл, можно записать такой вызов:

```
scalar(ln(vector(2)))
```

Также эта функция помогает упростить некоторые выражения. Например, чтобы узнать, какие серверы были запущены в текущем году, можно использовать такое выражение:

```
year(process_start_time_seconds)
== scalar(year())
```

вместо:

```
year(process_start_time_seconds)
== on() group_left
    year()
```

потому что скалярные сравнения более понятны, чем сопоставления векторов с `group_left`, и мы знаем, что `year` здесь вернет только один образец.

Но не стоит злоупотреблять функцией `scalar`, потому что при ее использовании теряются все метки, а вместе с ними и возможность выполнять со-поставление векторов. Например,

```
sum(rate(node_cpu_seconds_total{mode!="idle",instance="localhost:9090"}[5m])
/
scalar(count(node_cpu_seconds_total{mode="idle",instance="localhost:9090"}))
```

вернет долю времени, в течение которого процессор машины не простоявал, но вам придется изменить и вычислить это выражение для каждого отдельного экземпляра.

Воспользовавшись всей мощью PromQL, можно записать выражение

```
sum without (cpu, mode)(
  rate(node_cpu_seconds_total{mode!="idle"}[5m])
)
/
count without(cpu, mode)(node_cpu_seconds_total{mode="idle"})
```

и вычислить ту же самую долю, но сразу для всех машин одновременно.

Математические функции

Математические функции выполняют стандартные математические операции над мгновенными векторами, такие как вычисление абсолютных значений или логарифмирование. Они применяются к каждому образцу в мгновенном векторе независимо и удаляют имя метрики из возвращающегося результата.

abs

`abs` принимает мгновенный вектор и возвращает абсолютные значения для всех его образцов, т. е. любые отрицательные числа заменяются положительными.

Выражение

```
abs(process_open_fds - 15)
```

вернет, насколько далеко количество дескрипторов открытых файлов в каждом процессе отстоит от 15. Для значений счетчиков 5 и 25 будет возвращено одно и то же значение 10.

ln, log2 и log10

Функции `ln`, `log2` и `log10` принимают мгновенный вектор и возвращают логарифмы значений с основаниями e (число Эйлера), 2 и 10 соответственно. `ln` также известен как *натуральный логарифм*.

Эти функции можно использовать для получения представления о различных порядках величин. Например, вот как можно подсчитать число девяток¹ в количестве успешных обращений к конечной точке API за последний час:

```
log10(
  sum without(instance)(rate(requests_failed_total[1h]))
/
  sum without(instance)(rate(requests_total[1h]))
) * -1
```

- ✓ Если вам потребуется вычислить логарифм с другим основанием, то используйте формулу изменения основания. Например, вот как можно вычислить логарифм по основанию 3 для мгновенного вектора x :

$\ln(x) / \ln(3)$

Функции логарифмов также могут пригодиться для построения графиков, когда обычные линейные графики не способны надлежащим образом представить большие различия в значениях. Однако часто лучше положиться на встроенные параметры построения графиков с логарифмической шкалой в таких инструментах, как Grafana, потому что они, как правило, изящнее обрабатывают пограничные случаи, такие как отрицательные логарифмы, возвращающие NaN.

exp

`exp` вычисляет значение натуральной показательной функции, обратной к функции `ln`. Например,

```
exp(vector(1))
```

вернет

```
{} 2.718281828459045
```

число Эйлера.

sqrt

Функция `sqrt` возвращает квадратный корень значений в мгновенном векторе. Например,

¹ 99 % – это две девятки.

```
sqrt(vector(9))
```

вернет

```
{} 3
```

`sqrt` появилась раньше оператора возведения в степень `^`, поэтому тот же результат теперь можно вычислить так:

```
vector(9) ^ 0.5
```

Для вычисления других корней можно использовать тот же подход. Например, кубический корень (корень третьей степени) можно вычислить так:

```
vector(9) ^ (1/3)
```

ceil и floor

`ceil` и `floor` позволяют округлить значения в мгновенном векторе. `ceil` всегда округляет вверх до ближайшего целого числа, а `floor` – вниз. Например,

```
ceil(vector(0.1))
```

вернет

```
{} 1
```

round

`round` округляет значения в мгновенном векторе до ближайшего целого числа. Если передать значение, находящееся ровно посередине между двумя целыми числами, то оно округлится в большую сторону. Например,

```
round(vector(5.5))
```

вернет

```
{} 6
```

Кроме того, `round` принимает второй необязательный аргумент – скаляр. Получив этот второй аргумент, `round` округлит значение до ближайшего кратного этому второму аргументу:

```
round(vector(2446), 1000)
```

вернет

```
{} 2000
```

Этот вызов эквивалентен выражению

```
round(vector(2446) / 1000) * 1000
```

но выглядит проще и понятнее.

clamp, clamp_max и clamp_min

Иногда вам будут встречаться метрики, возвращающие ложные значения, выходящие далеко за пределы нормального диапазона например значение датчика, которое всегда должно быть положительным, иногда может оказаться отрицательным. `clip_max` и `clip_min` позволяют установить верхнюю и нижнюю границы соответственно для значений в мгновенном векторе.

Например, если вы уверены, что ваши процессы не могут иметь менее 10 дескрипторов открытых файлов, то можете использовать выражение

```
clamp_min(process_open_fds, 10)
```

которое даст такой результат:

```
{instance="localhost:9090",job="prometheus"} 46  
{instance="localhost:9100",job="node"} 10
```

`clamp` позволяет указать сразу две границы, верхнюю и нижнюю. На тех же данных следующий запрос

```
clamp(process_open_fds, 10, 20)
```

вернет:

```
{instance="localhost:9090",job="prometheus"} 20  
{instance="localhost:9100",job="node"} 10
```

sgn

`sgn` возвращает вектор, каждый образец в котором представляет знак соответствующего образца в исходном векторе. Знак определяется следующим образом: 1, если значение положительное, -1, если значение отрицательное, и 0, если значение равно нулю.

```
sgn(vector(100))
```

вернет

```
{} 1
```

Тригонометрические функции

В PromQL доступно 12 тригонометрических функций. Они работают в радианах:

- `acos` вычисляет арккосинус значений;
- `acosh` вычисляет гиперболический арккосинус значений;
- `asin` вычисляет арксинус значений;

- `asinh` вычисляет гиперболический арксинус значений;
- `atan` вычисляет арктангенс значений;
- `atanh` вычисляет гиперболический арктангенс значений;
- `cos` вычисляет косинус значений;
- `cosh` вычисляет гиперболический косинус значений;
- `sin` вычисляет синус значений.
- `sinh` вычисляет гиперболический синус значений;
- `tan` вычисляет тангенс значений;
- `tanh` вычисляет гиперболический тангенс значений.

Имеются также три дополнительные функции, полезные для преобразования значений между градусами в радианами:

- `deg` преобразует значения в радианах в градусы;
- `pi` возвращает число пи;
- `rad` преобразует значения в градусах в радианы.

```
sin(vector(pi()/2))
```

вернет

```
{} 1
```

Время и дата

Prometheus предлагает несколько функций для работы с датой и временем, очень удобных и избавляющих от необходимости реализовывать свою логику для операций с такими значениями. Prometheus работает исключительно в формате UTC и не имеет представления о часовых поясах.

time

Функция `time` – самая простая из набора функций для работы с временем. Она возвращает время выполнения запроса в секундах от начала эпохи Unix¹ в виде скаляра. Например,

```
time()
```

может вернуть

```
1652911202.529
```

Если использовать `time` с конечной точкой `query_range`, то все результаты получатся разными, потому что каждый шаг будет вычисляться в свой момент времени.

¹ Полночь 1 января 1970 года по всемирному координированному времени (UTC).

Лучшей практикой в Prometheus считается экспорт времени Unix в секундах, когда произошло какое-то интересное событие, а не количество времени, прошедшего с момента этого события. Это надежнее, потому что при обновлении метрики не возникает сбоев. Затем с помощью функции `time` можно вычислить длительности интересующих интервалов. Например, вот как можно узнать продолжительность работы процессов:

```
time() - process_start_time_seconds
```

Этот вызов может вернуть, например, такие результаты:

```
{instance="localhost:9090",job="prometheus"} 313.5699999332428
{instance="localhost:9100",job="node"} 322.2599999046326
```

Здесь Node Exporter и Prometheus работают чуть дольше 5 мин. Если у вас было пакетное задание, успешно отправившее метрики через Pushgateway, как обсуждалось в разделе «Pushgateway» главы 4, то вы сможете выявить задания, которые не завершились успехом за последний час:

```
time() - my_job_last_success_seconds > 3600
```

Функции `minute`, `hour`, `day_of_week`, `day_of_month`, `day_of_year`, `days_in_month`, `month` и `year`

`time` охватывает большинство потребностей, но иногда может понадобиться логика, основанная на часах или календаре. Преобразование времени в минуты и часы реализуется довольно просто¹, но, помимо этого, нужно также учитывать такие проблемы, как високосные дни.

Все эти функции, перечисленные в заголовке раздела, возвращают значение времени выполнения запроса в виде мгновенного вектора с одним образцом и без меток. Мы пишем эти строки в 13:37 субботы, 5 ноября 2022 года, по часовому поясу UTC. Вот что возвращают эти функции в данный момент:

Выражение	Результат
<code>minute()</code>	<code>[] 37</code>
<code>hour()</code>	<code>[] 13</code>
<code>day_of_week()</code>	<code>[] 6</code>
<code>day_of_month()</code>	<code>[] 5</code>
<code>day_of_year()</code>	<code>[] 309</code>
<code>days_in_month()</code>	<code>[] 30</code>
<code>month()</code>	<code>[] 11</code>
<code>year()</code>	<code>[] 2022</code>

¹ Минуты, например, – это `floor(vector(time()) / 60 % 60)`.

`day_of_week` начинает отсчет дней недели с воскресенья и с 0, поэтому 6 означает субботу. Чтобы проверить, является ли текущая дата последним днем месяца, можно сравнить результаты `day_of_month` и `days_in_month`.

Возможно, вам интересно, почему эти функции не возвращают скаляры, с которыми удобнее работать. Дело в том, что все эти функции принимают необязательный аргумент¹, в котором можно передать мгновенный вектор. Например, вот как можно узнать, в каком году были запущены ваши процессы:

```
year(process_start_time_seconds)
```

Этот запрос вернет такой результат:

```
{instance="localhost:9090",job="prometheus"} 2022
{instance="localhost:9100",job="node"} 2022
```

Следующий пример демонстрирует, как подсчитать количество процессов, запущенных в текущем месяце:

```
sum(
    (year(process_start_time_seconds) == bool scalar(year()))
    *
    (month(process_start_time_seconds) == bool scalar(month())))
)
```

Здесь используется тот факт, что оператор умножения действует подобно *оператору and* со значениями 1 вместо `true` и 0 вместо `false`.

timestamp

Функция `timestamp` отличается от других функций для работы с датой и временем тем, что получает отметку времени из образцов в мгновенном векторе, а не из значений. Как упоминалось в разделах «Мгновенный вектор» и «query» главы 13, отметки времени для образцов, возвращаемых всеми операторами, функциями, а также HTTP API `query_range` и `query`, когда они возвращают мгновенные векторы, будут содержать время выполнения запроса.

Однако отметка времени образцов в мгновенном векторе, полученном применением селектора мгновенного вектора, будет фактической отметкой времени². Функция `timestamp` позволяет получить к ним доступ. Например, вот как можно узнать время начала последнего цикла опроса каждой цели:

```
timestamp(up)
```

потому что по умолчанию отметка времени для данных опроса – это время начала опроса. Точно так же отметка времени для образцов из правил записи, как описано в главе 17, является временем выполнения группы правил.

¹ Значение по умолчанию этого аргумента – `vector(time())`.

² Как и отметки времени образцов, если в HTTP API `query` передать селектор вектора диапазона.

Если вам понадобится просмотреть исходные данные, получаемые с об разцами, например для отладки, то используйте селектор вектора диапазона с HTTP API query. Тем не менее timestamp имеет несколько практических применений. Например,

```
node_time_seconds - timestamp(node_time_seconds)
```

вернет разницу между тем, когда Prometheus начал опрос экспортера Node Exporter, и временем, которое Node Exporter считает текущим. Пусть и не со 100%-ной гарантией (многое зависит от загрузки на машины), но этот прием позволит вам узнать, не рассинхронизировалось ли время между машинами более чем на несколько секунд.

Метки

В идеальном мире имена и значения меток, используемые различными частями системы, должны быть согласованы; например, не должно быть метки customer в одном месте и cust в другом. Такие несоответствия лучше всего устранять в исходном коде или с помощью конфигурации metric_relabel_configs, как описано в разделе «metric_relabel_configs» главы 8. Однако это возможно не всегда, и в таких случаях вам на помощь придут две функции для работы с метками.

label_replace

label_replace позволяет выполнять подстановку значений меток с применением регулярных выражений. Например, если нужно переименовать метку device метрики node_disk_read_bytes_total в dev, чтобы обеспечить сопоставление векторов так, как вам нужно, то это можно сделать так¹:

```
label_replace(node_disk_read_bytes_total, "dev", "{$1}", "device", "(.*)")
```

Этот вызов вернет примерно такой результат:

```
node_disk_read_bytes_total{dev="sda",device="sda",instance="localhost:9100",
job="node"} 4766305792
```

В отличие от большинства функций label_replace не удаляет имя метрики, так как предполагается, что вы делаете что-то необычное, коль скоро вам пришлось прибегнуть к label_replace, и удаление имени метрики может усложнить задачу.

Функция label_replace принимает мгновенный вектор, имя метки на выходе, выражение замены, имя исходной метки и регулярное выражение. label_replace работает аналогично действию replace механизма изменения меток,

¹ На самом деле, поскольку node_disk_read_bytes_total является счетчиком, следует сначала применить rate, а уж потом label_replace.

но позволяет указать только одну исходную метку. Если регулярное выражение не соответствует данному образцу, то образец возвращается без изменений.

label_join

`label_join` позволяет объединить значения меток аналогично тому, как механизм изменения меток обрабатывает `source_labels`. Например, вот как можно объединить метки `job` и `instance` в новую метку:

```
label_join(node_disk_read_bytes_total, "combined", "-", "instance", "job")
```

Это выражение вернет такой результат:

```
node_disk_read_bytes_total{combined="localhost:9100-node",device="sda",
  instance="localhost:9100",job="node"} 4766359040
```

Подобно `label_replace`, функция `label_join` не удаляет имя метрики и принимает мгновенный вектор, имя метки на выходе, разделитель, а также ноль или более имен меток для объединения.

Функцию `label_join` можно комбинировать с `label_replace`, чтобы получить полноценную замену действия `replace` механизма изменения меток, но вообще, если это возможно, предпочтительнее использовать конфигурацию `metric_relabel_configs` или изменить исходный код инструментированных меток.

Отсутствующие серии, absent и absent_over_time

Как упоминалось в разделе «Многие ко многим и логические операторы» главы 15, функция `absent` играет роль оператора `not`. Если передать непустой мгновенный вектор в виде аргумента `absent`, то функция вернет пустой мгновенный вектор, а если передать пустой мгновенный вектор, то она вернет мгновенный вектор с одним образцом, имеющим значение 1.

Можно было бы ожидать, что в этом образце нет меток, однако `absent` действует немного умнее: если в аргументе передан селектор мгновенного вектора, то функция добавит метки из имеющихся сопоставителей равенства.

Выражение	Результат
<code>absent(up)</code>	пустой мгновенный вектор
<code>absent(up{job="prometheus"})</code>	пустой мгновенный вектор
<code>absent(up{job="missing"})</code>	<code>{job="missing"} 1</code>
<code>absent(up{job=~"missing"})</code>	<code>[] 1</code>
<code>absent(non_existent)</code>	<code>[] 1</code>
<code>absent(non_existent{job="foo",env="dev"})</code>	<code>{job="foo",env="dev"} 1</code>
<code>absent(non_existent{job="foo",env="dev"} * 0)</code>	<code>[] 1</code>

`absent` удобно использовать для определения отсутствия задания в механизме обнаружения служб. Оповещение о `up == 0` не будет работать в отсутствие целей, экспортирующих метрики `up!` Даже при использовании `static_configs` бывает целесообразно добавить такое предупреждение на случай, если в процессе генерирования вашего файла `prometheus.yml` что-то пойдет не так.

Если же вам нужно просто посыпать оповещение при отсутствии определенных метрик в цели, то используйте `unless`, как описано в разделе «Оператор `unless`» главы 15.

Помимо `absent`, имеется также аналогичная функция для работы с векторами диапазонов: `absent_over_time`. Она возвращает пустой вектор, если переданный ей вектор диапазона имеет какие-либо элементы, и одноделементный вектор со значением 1, если переданный вектор диапазона не имеет элементов.

Эту функцию можно использовать для оповещения об отсутствии временных рядов с данной комбинацией из имени метрики и метки в течение определенного периода времени.

Следующий запрос

```
absent_over_time(up{job="myjob"}[1h])
```

сообщит, что задание `myjob` не имеет цели в течение как минимум одного часа.

Сортировка с помощью `sort` и `sort_desc`

PromQL обычно не определяет порядок элементов в мгновенном векторе, поэтому он может меняться от случая к случаю. Но, используя `sort` или `sort_desc` в выражении PromQL, можно отсортировать мгновенный вектор по значению. Например,

```
sort(node_filesystem_size_bytes)
```

может вернуть:

```
node_filesystem_free_bytes{device="tmpfs",fstype="tmpfs",
  instance="localhost:9100",job="node",mountpoint="/run/lock"} 5238784
node_filesystem_free_bytes{device="/dev/sda1",fstype="vfat",
  instance="localhost:9100",job="node",mountpoint="/boot/efi"} 70300672
node_filesystem_free_bytes{device="tmpfs",fstype="tmpfs",
  instance="localhost:9100",job="node",mountpoint="/run"} 817094656
node_filesystem_free_bytes{device="tmpfs",fstype="tmpfs",
  instance="localhost:9100",job="node",mountpoint="/run/user/1000"} 826912768
node_filesystem_free_bytes{device="/dev/sda5",fstype="ext4",
  instance="localhost:9100",job="node",mountpoint="/"} 30791843840
```

Эти функции дают чисто косметический эффект, но могут помочь сэкономить силы и время при написании отчетов. Значения `NaN` всегда помещаются

в конец, поэтому нельзя сказать, что `sort` и `sort_desc` полностью противоположны друг другу.

- ✓ Мгновенные векторы, возвращаемые агрегаторами `topk` и `bottomk`, уже отсортированы по группам агрегирования.

Гистограммы с `histogram_quantile`

Функция `histogram_quantile` уже упоминалась в разделе «Гистограммы» главы 13. Она в чем-то похожа на агрегатор, потому что группирует образцы, как это сделал бы оператор `without(le)`, а затем вычисляет квантиль на основе их значений. Например,

```
histogram_quantile(
    0.90,
    rate(prometheus_tsdb_compaction_duration_seconds_bucket[1d])
)
```

вычислит квантиль 0.9 (также известный как 90-й процентиль) задержки уплотнения базы данных временных рядов Prometheus за прошедший день. Значения вне диапазона от нуля до единицы не имеют смысла для квантилей и ведут к бесконечности.

Как обсуждалось во врезке «Кумулятивные гистограммы» в главе 3, значения в сегментах должны быть кумулятивными, и должен иметься сегмент `+Inf`.

Всегда следует сначала применить `rate` к сегментам, представленным метриками-гистограммами Prometheus, как показано в разделе «Гистограммы» главы 3, так как `histogram_quantile` принимает датчики. Но есть очень небольшое количество экспортёров, которые экспортируют временные ряды, похожие на гистограммы, где сегменты являются датчиками, а не счетчиками. К таким временным рядам можно применять `histogram_quantile` напрямую.

Счетчики

Счетчики включают не только метрику-счетчик, но и временные ряды `_sum`, `_count` и `_bucket`. Значения счетчиков могут только увеличиваться. Когда приложение запускается или перезапускается, счетчики инициализируются значением 0, и функции для работы со счетчиками автоматически учитывают это.

Значения счетчиков сами по себе малополезны, и на практике они почти всегда преобразуются в датчики с использованием одной из функций, описываемых в этом разделе.

Все функции, работающие со счетчиками, принимают вектор диапазона и возвращают мгновенный вектор. Каждый временной ряд в векторе диапа-

зона обрабатывается отдельно, и для каждого возвращается не более одного образца. Если для одного из временных рядов в представленном диапазоне имеется только один образец, то эти функции ничего не вернут для него.

rate

Функция `rate` чаще других используется при работе со счетчиками, и, вероятно, это основная функция, которую вы будете использовать в PromQL. `rate` возвращает скорость увеличения счетчика в секунду для каждого временного ряда в переданном векторе диапазона. Вы уже видели много примеров применения `rate`, таких как:

```
rate(process_cpu_seconds_total[1m])
```

возвращающих, например, такие результаты:

```
{instance="localhost:9090",job="prometheus"} 0.0018000000000000683  
{instance="localhost:9100",job="node"} 0.005
```

`rate` автоматически обрабатывает события сброса счетчика, считая сбросом любое его уменьшение. Так, например, если имеется временной ряд со значениями [5, 10, 4, 6], то он будет рассматриваться как [5, 10, 14, 16]. `rate` предполагает, что цели, за которыми она наблюдает, являются относительно долгоживущими, по сравнению с интервалом опроса, так как нет никакой возможности обнаружить несколько событий сброса за короткий период времени. Если у вас есть цели, которые, как ожидается, будут действовать менее нескольких интервалов опроса, то подумайте о возможности их мониторинга на основе журнала.

`rate` должна обрабатывать такие сценарии, как появление и исчезновение временных рядов, например когда один из экземпляров запустился, а затем произошел сбой. Так, если счетчик в одном из экземпляров увеличивался на 10 каждую секунду, но экземпляр работал всего полчаса, то `rate(x_total[1h])` вернет результат около 5 в секунду.

Значения редко бывают точными. Со временем могут возникать отклонения, потому что опрос разных целей происходит в разное время, и шаги вызова `query_range` редко полностью совпадают с интервалом опроса, к тому же ожидается, что процедура опроса периодически будет терпеть неудачу. Функция `rate` должна учитывать все эти проблемы и давать верный результат в среднем по времени.

`rate` не предназначена для захвата каждого отдельного приращения, так как есть вероятность потери приращений, например если экземпляр остановится между опросами. Это может вызвать неточности, особенно если счетчики изменяются очень редко, например всего несколько раз в час. `rate` также может обрабатывать только изменения в счетчиках, потому что, если временной ряд счетчика появляется со значением 100, `rate` не знает, произошло ли это приращение только что, или цель работала на протяжении многих лет, и только сейчас стали поступать данные о ней.

При применении `rate` к вектору диапазона рекомендуется указывать интервал, который по меньшей мере в четыре раза превышает интервал опроса. Это гарантирует наличие хотя бы двух образцов для анализа, даже если опрос выполняется медленно, чтение данных занимает длительное время и имело место одна неудачная попытка опросить цель. Все эти проблемы неизбежны в реальных системах, поэтому так важна устойчивость. Например, для 1-минутного интервала опроса можно задать 4-минутный интервал анализа в вызове `rate`, но обычно в таких случаях интервал округляется до 5 мин¹.

Как правило, для единобразия желательно стремиться использовать один и тот же интервал во всех вызовах функции `rate`, потому что результаты вызовов `rate` с разными интервалами несопоставимы, и их, как правило, трудно отслеживать.

Прочитав обо всех этих тонкостях, кто-то из вас наверняка подумает: можно ли изменить `rate` так, чтобы сделать ее проще. Есть несколько вариантов решения этой проблемы, но все они имеют не только преимущества, но и недостатки. Устранив одну очевидную проблему, вы вызовете появление другой. Функция `rate` поддерживает достаточно хороший баланс между всеми этими проблемами и обеспечивает надежное решение, подходящее для оперативного мониторинга. Если вы столкнетесь с ситуацией, когда функция, подобная `rate`, не дает нужного результата, то мы рекомендуем подумать о возможности мониторинга на основе журналов, который не имеет этих особых проблем и может дать точные ответы за счет больших затрат.

increase

`increase` – это просто синтаксический сахар поверх `rate`. Вызов `increase(x_total[5m])` вернет тот же результат, что и вызов `rate(x_total[5m]) * 300`, т. е. произведение скорости на интервал. В остальном они действуют совершенно идентично.

Секунды являются базовой единицей для Prometheus, поэтому `increase` следует использовать только при отображении значений в дашбордах или отчетах. В ваших правилах записи и оповещениях лучше использовать `rate` для согласованности.

Одним из следствий высокой устойчивости `rate` и `increase` является возврат ими нецелочисленных результатов для целочисленных входных данных. Представьте, что у вас есть следующие точки данных внутри временного ряда:

```
21 в 2 с
22 в 7 с
24 в 12 с
```

и требуется вычислить `increase(x_total[15s])` в момент времени выполнения запроса 15 с. За 10 с значение увеличилось на 3, поэтому резонно было бы ожидать результата 3. Однако скорость оценивается за 15-секундный период, поэтому, чтобы избежать недооценки правильного ответа, имеющиеся 10-секундные данные экстраполируются на интервал в 15 с, что дает результат 4.5.

¹ Например, `rate(x_total[5m])`.

`rate` и `increase` предполагают, что тенденция изменения временного ряда продолжится за границей диапазона, если первый и последний образцы находятся в пределах 110 % от среднего значения. Иначе предполагается, что временной ряд простирается не более чем на 50 % от имеющегося интервала, но со значением не ниже нуля.

irate

`irate` похожа на `rate` – она тоже возвращает скорость увеличения счетчика в секунду. Однако она использует более простой алгоритм, учитывая только два последних образца из полученного вектора диапазона. Преимущество такого решения – в более высокой чувствительности к изменениям, благодаря чему вам не нужно беспокоиться о взаимосвязи между диапазоном вектора и интервалом опроса. Но оно имеет свой недостаток: так как функция учитывает только два образца, ее можно использовать только на графиках с полным увеличением¹. На рис. 16.1 показан результат сравнения `rate` и `irate` с 5-минутным интервалом.

Из-за отсутствия усреднения в `irate` график может выглядеть более изменчивым², и его труднее читать. Не рекомендуется использовать `irate` в оповещениях, поскольку она чувствительна к кратким всплескам и провалам; вместо этого используйте `rate`.

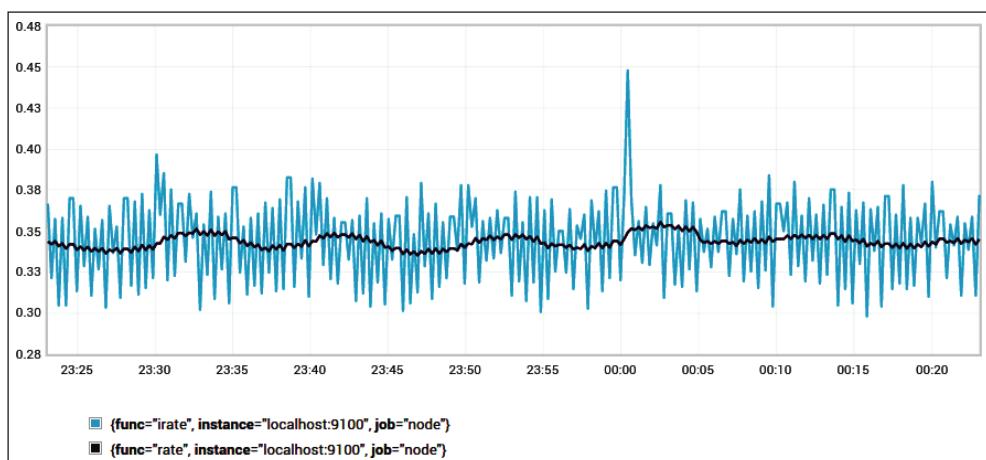


Рис. 16.1 ❖ Графики потребления процессорного времени экспортёром Node Exporter, полученные с помощью `rate` и `irate`

¹ Если шаг в `query_range` больше интервала опроса, то при использовании `irate` вы неизбежно будете пропускать данные.

² Имя `irate` – это сокращение от *instant rate* (мгновенная скорость), тем не менее Брайану кажется забавной игра слов, получившаяся с именем `irate` (можно перевести как *бешеная*).

resets

Иногда может возникнуть подозрение, что счетчик сбрасывается чаще, чем должен. Функция `resets` сообщает, сколько раз сбрасывался каждый временной ряд в векторе диапазона. Например, выражение

```
resets(process_cpu_seconds_total[1h])
```

сообщит, сколько раз за последний час сбрасывалось значение процессорного времени, потребляемого процессом. Теоретически количество сбросов должно совпадать с количеством перезапуска процесса¹, но если имела место какая-то ошибка, из-за которой происходил сброс, то значение, возвращаемое функцией `resets`, будет выше.

`resets` предназначена для отладки счетчиков, поскольку счетчики могут сбрасываться слишком часто и вызывать искажения в виде больших пиков на графиках. Тем не менее некоторые пользователи применяют ее, когда хотят узнать, сколько раз наблюдалось уменьшение значения метрики.

Изменение датчиков

В отличие от значений счетчиков значения датчиков полезны сами по себе, и к ним можно применять двухместные операторы и агрегаторы. Но иногда может понадобиться проанализировать историю изменения датчика, и в таких случаях вам пригодятся несколько функций, описываемых далее.

Подобно функциям для работы со счетчиками, эти функции тоже принимают вектор диапазона и возвращают мгновенный вектор с одним образцом для каждого временного ряда.

changes

Некоторые датчики могут меняться очень редко. Например, время запуска процесса не меняется в течение всего времени его жизни². Функция `changes` позволяет вам подсчитать, сколько раз менялось значение датчика. Например, выражение

```
changes(process_start_time_seconds[1h])
```

сообщит, сколько раз перезапускался процесс за последний час. Агрегирование этого значения по всем приложениям позволит вам определить, происходит ли сбой приложений с неизменной периодичностью.

¹ Более точную оценку количества перезапусков даст `changes(process_start_time_seconds[1h])`, поскольку отметки времени – это датчики (<https://oreil.ly/EHfIO>).

² Хотя были случаи, например в отчете об ошибке 289 в Prometheus (<https://oreil.ly/HW1K1>), когда ядро облачного провайдера предоставляло неверные метрики.

Из-за фундаментального характера метрик Prometheus может производить опрос недостаточно часто, чтобы увидеть все возможные изменения. Однако, если процесс перезапускается слишком часто, вы все равно обнаружите этот факт либо с помощью `changes`, либо по значению 0 в метрике `up`.

Функцию `changes` можно применять не только к `process_start_time_seconds`, но и к другим датчикам, когда интересен сам факт изменения датчика.

deriv

Часто бывает желательно знать, как быстро меняется датчик; например, как быстро увеличивается отставание записи в журнал, если оно вообще увеличивается. Это позволит не только заметить увеличение нагрузки, но и то, что она еще не начала снижаться.

Можно, конечно, использовать выражение `x - x offset 1h`, но оно использует только два образца и, следовательно, ненадежно, поскольку восприимчиво к отдельным выбросам. Функция `deriv` использует *регрессию методом наименьших квадратов*¹ для оценки наклона каждого временного ряда векторе диапазона. Например,

```
deriv(process_resident_memory_bytes[1h])
```

вычислит на основе значений, полученных за последний час, как быстро изменяется потребление оперативной памяти.

predict_linear

`predict_linear` делает еще один шаг вперед, по сравнению с `deriv`, и предсказывает, какое значение получит датчик в будущем, опираясь на данные в предоставленном диапазоне. Например,

```
predict_linear(node_filesystem_free_bytes{job="node"}[1h], 4 * 3600)
```

предскажет, сколько свободного места останется в каждой файловой системе через четыре часа, исходя из тенденций, наблюдавшихся за прошлый час. Это выражение примерно эквивалентно выражению:

```
deriv(node_filesystem_free_bytes{job="node"}[1h]) * 4 * 3600  
+  
node_filesystem_free_bytes{job="node"}
```

Но `predict_linear` даст несколько более точный прогноз, потому что использует точку пересечения линии регрессии.

`predict_linear` можно использовать для вывода предупреждений о нехватке ресурсов, когда статические пороговые значения, такие как 1 Гбайт, или процентные пороговые значения, такие как 10 % свободного места, могут

¹ Так же известна как *простая линейная регрессия*.

давать ложноположительные или ложноотрицательные срабатывания, в зависимости от размера файловой системы. Пороговое значение 1 Гбайт для файловой системы объемом 1 Тбайт предупредит о нехватке слишком поздно, а для файловой системы объемом 2 Гбайт – слишком рано. `predict_linear` действует более оптимально для всех размеров.

Может потребоваться немного поэкспериментировать, чтобы подобрать хорошие значения для диапазона прогнозирования. Если в данных наблюдается обычная пилообразная закономерность, то желательно увеличить диапазон, чтобы увеличить точность экстраполяции восходящей составляющей.

delta

`delta` действует подобно `increase`, но без обработки сброса счетчика. По мере возможности старайтесь не использовать эту функцию, так как на нее могут чрезмерно влиять одиночные выбросы. Вместо нее для сравнения с более ранним значением лучше использовать `deriv` или `x - x offset 1h`.

idelta

`idelta` извлекает два последних образца в заданном диапазоне и возвращает их разницу. `idelta` предназначена для расширенных вариантов использования. Например, приемы с применением `rate` и `i_rate` нравятся не всем, поэтому не желающие захламлять PromQL различными вариантами функции `rate` могут использовать `idelta` и правила записи.

holt_winters

Функция `holt_winters`¹ реализует *двойное экспоненциальное сглаживание Холта–Винтерса (Holt-Winters)*. Датчики иногда могут давать излишне пилообразные и трудно читаемые графики, к которым часто полезно применить некоторое сглаживание. В самом простом случае можно использовать `avg_over_time`, но иногда требуется что-то более сложное.

Эта функция работает с образцами из временного ряда, отслеживает сглаженное значение на данный момент и оценивает тенденции изменения данных. Каждый новый образец учитывается на основе *фактора сглаживания*, который определяет важность старых данных по сравнению с новыми, и *фактора тренда*, определяющего важность тренда. Например,

```
holt_winters(process_resident_memory_bytes[1h], 0.1, 0.5)
```

¹ Есть мнение, что эта функция получила не совсем правильное имя; отчет об ошибке 2458 в Prometheus (<https://oreil.ly/WTR0v>).

выполнит сглаживание графика потребления памяти с коэффициентом сглаживания 0.1 и коэффициентом тренда 0.5. Оба коэффициента должны иметь значение между 0 и 1.

Агрегирование по времени

Агрегаторы, такие как `avg`, работают с образцами в мгновенном векторе. Существует также набор функций, таких как `avg_over_time`, применяющих ту же логику к значениям временного ряда в векторе диапазона. Вот эти функции:

- `sum_over_time`;
- `count_over_time`;
- `avg_over_time`;
- `stddev_over_time`;
- `stdvar_over_time`;
- `min_over_time`;
- `max_over_time`;
- `quantile_over_time`.

Две дополнительные функции не связаны напрямую с агрегированием:

- `present_over_time` действует подобно агрегатору `group` и возвращает значение 1 для любой серии, соответствующей селектору диапазона;
- `last_over_time` возвращает последнее значение из любой серии, соответствующей селектору диапазона.

Например, чтобы увидеть пиковое потребление памяти процессом, наблюдавшееся в Prometheus, можно использовать выражение

```
max_over_time(process_resident_memory_bytes[1h])
```

и даже сделать еще один шаг и вычислить то же самое значение для всего приложения:

```
max without(instance)(max_over_time(process_resident_memory_bytes[1h]))
```

Эти функции работают только со значениями образцов; они не выполняют никакого взвешивания, основанного на промежутке времени между образцами или какой-либо другой логике, связанной с отметками времени. Это означает, что если, например, изменить интервал опроса, то в результатах таких функций, как `avg_over_time` и `quantile_over_time`, будет наблюдаться смещение в сторону периода времени с более частыми циклами опроса. Точно так же если в течение определенного времени имели место неудачные попытки извлечения данных, то этот период будет менее представлен в результатах.

Эти функции используются с датчиками¹. Если вы решите получить `avg_over_time` для `rate`, то у вас ничего не получится, потому что `rate` возвращает

¹ Хотя `count_over_time` и `present_over_time` игнорируют значения, они могут пригодиться для отладки метрик любого типа.

мгновенные значения, а не векторы диапазонов. Однако `rate` уже вычисляет среднее значение во времени, поэтому вы можете просто увеличить интервал, передаваемый в вызов `rate`. Например, вместо вызова

```
avg_over_time(rate(x_total[5m])[1h])
```

который генерирует ошибку, можно выполнить:

```
rate(x_total[1h])
```

О том, как мгновенные векторы, возвращаемые функциями, передать другим функциям, принимающим векторы диапазонов, мы поговорим в следующей главе, посвященной правилам записи.

Глава 17

Правила записи

HTTP API не единственный способ доступа к возможностям PromQL. Для регулярного выполнения запросов PromQL и получения их результатов можно использовать так называемые *правила записи* (recording rules). С их помощью можно повысить эффективность дашбордов, организовать передачу агрегированных результатов куда-то еще и составлять функции для обработки векторов диапазонов. В других системах мониторинга эквивалентные возможности могут называться постоянными или непрерывными запросами. Правила оповещения (описанные в главе 18) тоже являются разновидностью правил записи. В этой главе мы покажем, как и когда использовать правила записи.

Использование правил записи

Правила записи определяются в отдельных файлах – *файлах правил*, – ссылки на которые указываются в *prometheus.yml*. Файлы правил, как и *prometheus.yml*, тоже имеют формат YAML. Ссылки на файлы правил указываются в разделе `rule_files` в файле *prometheus.yml*. В примере 17.1 показана конфигурация опроса двух целей, загружающая файл правил с именем *rules.yml*.

Пример 17.1 ♦ *prometheus.yml* с конфигурацией опроса двух целей и ссылкой на файл правил

```
global:
  scrape_interval: 10s
  evaluation_interval: 10s
rule_files:
  - rules.yml
scrape_configs:
  - job_name: prometheus
    static_configs:
      - targets:
          - localhost:9090
  - job_name: node
```

```
static_configs:
- targets:
  - localhost:9100
```

По аналогии с полем `files` в списке `file_sd_configs`, описанном в разделе «Обнаружение служб на основе файла» главы 8, `rule_files` принимает список путей и позволяет использовать подстановочные знаки в именах файлов. В отличие от обнаружения служб на основе файлов, изменения в `rule_files` и в файлах правил не подхватываются автоматически. Поэтому, чтобы изменения вступили в силу, необходимо перезапустить Prometheus или заставить его перезагрузить конфигурацию.

Чтобы вынудить сервер Prometheus перезагрузить конфигурацию, можно отправить ему сигнал `SIGHUP`, например так:

```
kill -HUP <pid>
```

где `pid` – идентификатор процесса Prometheus. Также можете отправить HTTP-запрос POST в конечную точку `/-/reload` Prometheus, но, чтобы иметь такую возможность, необходимо явно передать флаг `--web.enable-lifecycle` при запуске сервера. Если перезагрузить конфигурацию не получилось, то Prometheus зафиксирует ошибку в журнале, и вы увидите, что метрика `prometheus_config_reload_successful` получила значение 0.

Своевременно обнаружить ошибки в `prometheus.yml` можно с помощью команды `rgomttool check config`. Она также проверит все файлы правил, на которые ссылается `prometheus.yml`. Такую проверку можно выполнять перед отправкой в репозиторий или в модульных тестах перед развертыванием файла конфигурации. Проверить синтаксис отдельных файлов правил можно с помощью `rgomttool check rules`.

Сами файлы правил состоят из нуля¹ или более групп правил. В примере 17.2 показано содержимое файла `rules.yml` с одной группой и двумя правилами.

Пример 17.2 ◊ `rules.yml` с одной группой, содержащей два правила

```
groups:
- name: example
rules:
- record: job:process_cpu_seconds:rate5m
  expr: sum without(instance)(rate(process_cpu_seconds_total[5m]))
- record: job:process_open_fds:max
  expr: max without(instance)(process_open_fds)
```

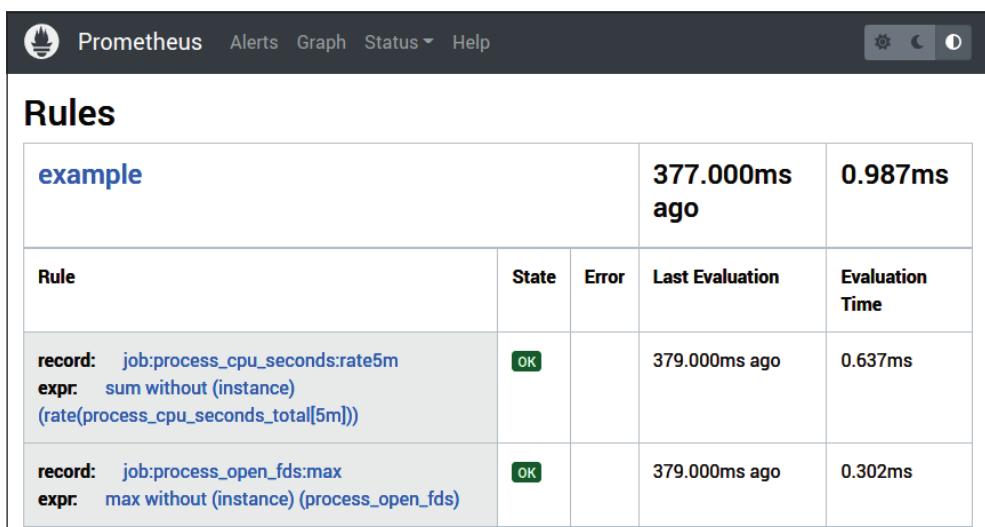
Обратите внимание, что группа имеет имя. Имя должно быть уникальным в пределах файла правил и отображается в пользовательском интерфейсе и метриках Prometheus. `expr` – это выражение PromQL, результат которого должен быть возвращен в виде метрики с именем, определяемым параметром `record`.

¹ Файлы, содержащие нуль групп или нуль правил, технически допустимы, но бесполезны.

Для группы можно указать интервал выполнения `evaluation_interval`, но, как и в случае с параметром `scrape_interval`, старайтесь использовать во всех группах одинаковый интервал для единобразия. В поле `labels` можно указать набор меток, которые будут добавлены в результаты, но эта возможность редко используется в правилах записи¹.

Правила в группе обрабатываются по очереди, и результаты обработки первого правила записываются в базу данных временных рядов перед запуском второго правила. Однако группы правил могут выполняться в разное время, так же как разные цели могут опрашиваться в разное время. Это сделано намеренно, чтобы более равномерно распределить нагрузку на ваш Prometheus.

После загрузки и запуска правил их можно увидеть на странице состояния правил `http://localhost:9090/rules`, как показано на рис. 17.1.



The screenshot shows the Prometheus interface with the 'Rules' section selected. The 'example' rule is listed with its last evaluation time as '377.000ms ago' and evaluation time as '0.987ms'. Below it, two specific metrics are listed:

Rule	State	Error	Last Evaluation	Evaluation Time
<code>record: job:process_cpu_seconds:rate5m expr: sum without (instance) (rate(process_cpu_seconds_total[5m]))</code>	OK		379.000ms ago	0.637ms
<code>record: job:process_open_fds:max expr: max without (instance) (process_open_fds)</code>	OK		379.000ms ago	0.302ms

Рис. 17.1 ❖ Страница состояния правил записи в Prometheus

Помимо списка правил, на странице также отображается время, потребованное для выполнения каждой группы и каждого правила. С помощью этой информации можно найти дорогостоящие правила и при необходимости пересмотреть их реализацию. Метрика `prometheus_rule_group_last_duration_seconds` также сообщает вам, сколько времени потребовалось для выполнения каждой группы в последнем цикле. С ее помощью можно определить, произошли ли изменения в скорости выполнения ваших правил. Метрики с продолжительностями отдельных правил отсутствуют, потому что это может вызвать проблемы с кардинальностью. В нашем примере для выполнения правил требуется меньше миллисекунды, что намного меньше интервала опроса, поэтому нет повода беспокоиться.

¹ Однако параметр `labels` используется практически во всех правилах оповещения.

i В Prometheus не предусмотрен API для загрузки или изменения правил. Как и в случае с конфигурацией Prometheus в целом, файлы предназначены для использования в качестве основы, на которой вы могли бы построить такую систему, если хотите.

Когда использовать правила записи

Правила записи можно использовать в нескольких случаях. Чаще всего правила записи используются для агрегирования метрик, чтобы повысить эффективность запросов. Это особенно востребовано в дашбордах, механизме федерации и перед сохранением метрик в хранилище. Правила записи также можно использовать для составления функций, обрабатывающих векторы диапазонов, и для реализации API метрик, доступного другим командам.

Уменьшение кардинальности

Имея выражение

```
sum without(instance)(rate(process_cpu_seconds_total{job="node"})[5m])
```

в дашборде, можно обнаружить, что при небольшом количестве целей Prometheus отвечает достаточно быстро. Но с увеличением числа целей до сотен и тысяч время отклика конечной точки `query_range` заметно увеличивается.

Вместо того чтобы требовать от PromQL получать и обрабатывать тысячи временных рядов для каждого графика в дашборде, можно предварительно вычислить необходимые значения с помощью группы правил, например¹:

```
groups:
- name: node
  rules:
    - record: job:process_cpu_seconds:rate5m
      expr: >
        sum without(instance)(
          rate(process_cpu_seconds_total{job="node"})[5m]
        )
```

результаты которой будут выводиться в метрику `job:process_cpu_seconds:rate5m`.

Теперь в дашборде вам достаточно получить только один временной ряд. Тот же прием можно применить к инструментированным меткам, чтобы уменьшить количество обрабатываемых временных рядов, если количество имеющихся экземпляров слишком велико. По сути, это обмен текущих затрат ресурсов на гораздо более низкую задержку и ресурсоемкость ваших запросов. Из-за этого компромисса обычно нецелесообразно определять правила, использующие длинные векторы диапазонов, потому что такие запросы, как

¹ Символ `>` – один из способов оформления многострочных строк в YAML.

правило, обходятся очень дорого, а их регулярное выполнение может привести к проблемам с производительностью.

Все правила, соответствующие одному заданию, старайтесь помещать в одну группу. Благодаря этому они будут получать одинаковые отметки времени, а вы сможете избежать искажений при выполнении дальнейших математических операций. Все правила записи в группе получают одно и то же время выполнения запроса, и все выходные образцы тоже получат эту же отметку времени.

Подобные правила агрегирования могут пригодиться не только для ускорения работы дашбордов. При использовании механизма федерации, как обсуждается в разделе «Выход на глобальный уровень с помощью механизма федерации» главы 21, вам всегда будет нужно получать агрегированные метрики, так как иначе придется обрабатывать большие массивы метрик на уровне экземпляра. С точки зрения производительности для Prometheus, использующего механизм федерации, выгоднее самому опрашивать цели¹.

Аналогичная логика применима, когда требуется сохранять некоторые метрики в долговременное хранилище. При планировании емкости на основе данных за месяцы или годы детали отдельных экземпляров не имеют значения. Сохраняя в основном агрегированные показатели, можно сэкономить много ресурсов за счет потери небольшого объема полезной информации².

Вы часто будете создавать правила агрегирования, основанные на одной и той же метрике, но с разными наборами меток. Вместо вычисления каждого агрегатного значения отдельно можно воспользоваться правилами записи, когда одно правило использует результат вычисления другого. Например:

```
groups:
  - name: node
    rules:
      - record: job_device:node_disk_read_bytes:rate5m
        expr: >
          sum without(instance)(
            rate(node_disk_read_bytes_total{job="node"}[5m])
          )
      - record: job:node_disk_read_bytes:rate5m
        expr: >
          sum without(device)(
            job_device:node_disk_read_bytes:rate5m{job="node"}
          )
```

Для правильной работы подобные наборы правил должны быть упорядочены в пределах одной группы³. Обычно лучше явно указать в селекторах

¹ С точки зрения производительности множество мелких опросов, равномерно распределенных во времени, лучше одного массивного опроса.

² Это можно сделать с помощью объединения (механизма федерации), удаленного изменения меток или удаления ненужных временных рядов.

³ До Prometheus 2.0 такое решение было невозможно из-за отсутствия понятия «группа правил» и гарантий, что одно правило всегда будет выполняться только после завершения другого.

задание, к которому применяются ваши правила, чтобы группы не наступали друг другу на пятки.

Составление функций для обработки векторов диапазонов

Как упоминалось в разделе «Агрегирование по времени» главы 16, к результатам функций, возвращающих мгновенные векторы, нельзя применять функции, ожидающие получить векторы диапазонов. Например, попытка вызова `max_over_time(sum without(instance)(rate(x_total[5m]))[1h])` вызовет синтаксическую ошибку. В PromQL есть поддержка подзапросов для таких случаев, однако тот же результат можно получить с помощью правил записи:

```
groups:
- name: j_job_rules
  rules:
    - record: job:x:rate5m
      expr: >
        sum without(instance)(
          rate(x_total{job="j"}[5m])
        )
    - record: job:x:max_over_time1h_rate5m
      expr: max_over_time(job:x:rate5m{job="j"}[1h])
```

Этот подход можно использовать с любой функцией, принимающей вектор диапазона, включая функции `_over_time`, а также `predict_linear`, `deriv` и `holt_winters`.

Однако этот метод не следует использовать с функциями `rate`, `irate` и `increase`, так как выражение `rate(sum(x_total)[5m])` будет давать значительные пики каждый раз, когда один из счетчиков сбрасывается или исчезает.

! Всегда сначала вычисляйте `rate`, а затем `sum`, и никогда в обратном порядке!

Необязательно вызывать внешнюю функцию внутри правила записи. В предыдущем примере может оказаться предпочтительнее вызывать `max_over_time` лишь по мере необходимости. Например, основным предназначением этого конкретного примера может быть планирование пропускной способности, при котором должен учитываться пиковый, а не средний трафик. Поскольку планирование пропускной способности часто выполняется раз в месяц или раз в квартал, нет особого смысла вызывать `max_over_time` каждую минуту, намного эффективнее запускать запрос, только когда это действительно нужно. Вызовы функций, обрабатывающие более длительные диапазоны времени, также могут стоить очень дорого из-за большого объема обрабатываемых данных. Будьте осторожны при работе с диапазонами длиннее одного часа и особенно включающими множество временных рядов.

Правила для API

Обычно ваши серверы Prometheus будут использоваться только вами и вашей командой. Но иногда другие команды могут пожелать получить метрики из вашего сервера Prometheus. Если эти метрики имеют информационный характер или зависит от метрик, которые вряд ли изменятся, то в этом нет ничего необычного, потому что, если вы что-то измените в них, это не приведет к концу света. Но если метрики используются для автоматизации систем или процессов, неподконтрольных вам, то предпочтительнее создать специальные метрики для других команд, чтобы они могли использовать их как разновидность общедоступного API. Если впоследствии вам потребуется изменить метки или правила внутри Prometheus, вы можете сделать это, гарантируя сохранность семантики метрик, от которых зависит другая команда.

Именование таких метрик, как правило, не подчиняется обычным соглашениям об именовании, и в имя метрики или в метку часто включается имя команды, потребляющей ее.

Правила записи довольно редко применяются для этой цели. Если другая команда использует ваш Prometheus до такой степени, что это возлагает на вас нетривиальное бремя обслуживания, вы можете попросить их запустить свой сервер Prometheus для получения необходимых им метрик.

Как не следует использовать правила

В своей практике мы подметили несколько распространенных антипаттернов использования правил записи, о которых хотели бы рассказать, чтобы вы могли их избежать.

Первый антипаттерн: правила, которые сводят на нет преимущества меток. Например:

```
- record: job_device:node_disk_read_bytes_sda:rate5m
expr: >
  sum without(instance)(
    rate(node_disk_read_bytes_total{job="node",device="sda"}[5m])
  )
- record: job_device:node_disk_read_bytes_sdb:rate5m
expr: >
  sum without(instance)(
    rate(node_disk_read_bytes_total{job="node",device="sdb"}[5m])
  )
```

Такое решение может вынудить вас создать отдельное правило для каждого возможного значения метки `device`, и их результаты будет очень сложно агрегировать. Это сводит на нет главное предназначение меток, одной из самых мощных особенностей Prometheus. Избегайте перемещения значений меток в имена метрик, а если вам нужно ограничить возвращаемые временные ряды на основе значения метки, то используйте сопоставитель в запросе. Точно так же не перемещайте метку `job` в имя метрики.

Второй антипаттерн: предварительное агрегирование всех метрик, экспортируемых приложением. Конечно, агрегирование помогает уменьшить кардинальность и повысить производительность, но, применяя агрегирование, важно не переусердствовать. В системе мониторинга, основанной на метриках, часто до 90 % метрик остаются неиспользуемыми¹, поэтому агрегирование по умолчанию всего подряд является пустой тратой ресурсов и требует лишних усилий при добавлении и удалении метрик со временем. Поэтому добавляйте агрегирование по мере необходимости. Эти остальные 90 % метрик по-прежнему будут доступны, когда вам придется отлаживать какую-то странную проблему в недрах вашей системы, а за отказ от их агрегирования вам придется заплатить лишь небольшим увеличением времени выполнения запросов.

Основная цель правил записи – уменьшить кардинальность, поэтому часто нет особого смысла в правилах, сохраняющих метку `instance`. Запрос десяти временных рядов ненамного дороже запроса одного большого временного ряда. Если ваши цели поставляют метрики с высокой кардинальностью, то правила записи, сохраняющие метку `instance`, могут иметь смысл, но при этом вы должны подумать о том, какие инструментированные метки можно было бы удалить для уменьшения кардинальности.

С такими правилами, как

```
- record: job:x:max_over_time1h_rate5m
  expr: max_over_time(job:x:rate5m{job="j"}[1h])
```

из предыдущего раздела, может возникнуть соблазн присвоить их параметру `evaluation_interval` значение, равное одному часу, чтобы сэкономить ресурсы. Но имейте в виду, что это не лучшая мысль. Во-первых, входная метрика получена из правила записи, которое уже уменьшило кардинальность, поэтому любая экономия ресурсов, скорее всего, будет ничтожной. Во-вторых, Prometheus гарантирует только, что правило будет выполнено только один раз в час, но не гарантирует, в какой момент в течение часа это произойдет. Поскольку результаты обычно нужны в начале часа, в сочетании с обработкой устаревания этот прием не сработает. В-третьих, чтобы избежать ненужных сложностей и для единообразия желательно использовать один и тот же интервал во всех ваших серверах Prometheus.

Последний антипаттерн, которого мы советуем избегать, – использование правил записи для исправления плохих имен метрик и меток. Этот антипаттерн приводит к потере исходных отметок времени и затрудняет определение источника и назначения метрики. Исправлять имена метрик лучше всего в источнике, а если это невозможно по техническим или политическим причинам, то следует подумать о возможности применения `metric_relabel_configs`, как описано в разделе «`metric_relabel_configs`» главы 8, если улучшение названий метрик стоит того, чтобы пойти наперекор ожиданиям других увидеть первоначальные названия.

К сожалению, всегда будут встречаться системы, экспортирующие метрики, которые противоречат соглашениям, принятым в Prometheus, и у вас не будет другого выбора, кроме как исправить их при возможности.

¹ Брайан получил эту цифру на примере нескольких систем мониторинга.

Именование правил записи

Использование хорошего соглашения для именования правил записи позволит не только сразу сказать, что означает имя метрики, создаваемой данным правилом, но также упростит совместное использование правил с другими благодаря общему словарю.

Как упоминалось в разделе «Какие имена выбирать для метрик?» главы 3, двоеточие считается допустимым символом в именах метрик, но его следует избегать при инструментировании, потому что пользователь может воспользоваться ими, чтобы добавить свою структуру в правила записи. Предлагаемое здесь соглашение уравновешивает точность и лаконичность и основано на многолетнем опыте.

Суть этого соглашения состоит в следующем: имена метрик должны начинаться с участвующих в них меток, затем должно следовать имя метрики и операции, выполняемые с метрикой. Эти три раздела разделены двоеточиями, поэтому в имени метрики всегда будет либо ноль, либо два двоеточия. Например, увидев имя

```
job_device:node_disk_read_bytes:rate5m
```

мы можем сказать, что правило возвращает метрику, имеющую метки `job` и `device`, основанную на метрике `node_disk_read_bytes`, и что эта метрика – счетчик, к которому была применена функция `rate(node_disk_read_bytes_total[5m])`. Эти части определяют *уровень, метрику и операции*.

Уровень

Уровень сообщает уровень агрегирования метрики по имеющимся меткам. Он всегда будет включать инструментированные метки (если они еще не были агрегированы), метку `job` и любые другие целевые метки, имеющие отношение к метрике. Выбор включаемых целевых меток зависит от контекста. Если все ваши цели экспортируют метку `env`, которая не влияет на правила, то нет необходимости включать ее в имена метрик. Но если задание разбито на части с меткой `shard`, то вам, вероятно, следует включить ее.

Метрика

Метрика – это просто имя метрики или временного ряда. Вы можете удалить суффикс `_total` из имен счетчиков для большей краткости, но в остальных случаях желательно использовать точные имена метрик. Сохраняя имена метрик, вы потом легко сможете отыскать в коде эту метрику, и наоборот, просматривая код, вы сможете быстро определить, используется ли она в агрегировании. Для отношений желательно использовать фразы вида `foo_reg_bar`, но есть специальное правило для работы с отношениями `_sum` и `_count`.

Операции

Операции – это список функций и агрегаторов, применяемых к метрике, функции, применяемые последними, должны следовать первыми. Если

к метрике применяются две операции `sum` или `max`, то указывайте только одну, так как сумма суммы остается суммой. Поскольку `sum` является функцией агрегирования по умолчанию, обычно не нужно указывать ее. Но если никакие другие операции не применяются, то `sum` является хорошим значением по умолчанию. В зависимости от того, какие операции вы планируете применять на других уровнях, может иметь смысл добавить `min` или `max` к имени базовой метрики. Для обозначения операции деления используйте `ratio`.

Рассмотрим простой пример. Если у вас есть счетчик `foo_total` с инструментированной меткой `bar`, то агрегирование по метке `instance` выглядело бы так:

```
- record: job_bar:foo:rate5m
expr: sum without(instance)(rate(foo_total{job="j"}[5m]))
```

Последующее агрегирование по метке `bar` будет выглядеть так:

```
- record: job:foo:rate5m
expr: sum without(bar)(job_bar:foo:rate5m{job="j"})
```

В этом примере уже видны некоторые преимущества описываемого подхода. При первом взгляде уже видно, что обработка меток здесь соответствует ожиданиям, поскольку во входном временном ряду в качестве уровня указано `job_bar`, затем `bar` была удалена с помощью оператора `without`, и в результате в качестве уровня указано имя метки `job`. В более сложных правилах и иерархиях это может пригодиться для выявления ошибок. Например, правило

```
- record: job:foo_per_bar:ratio_rate5m
expr: >
(
    job:foo:rate5m{job="j"}
/
    job:bar:rate10m{job="j"}
)
```

выглядит так, будто следует соглашению об именовании для отношений, но здесь есть несоответствие между `rate5m` и `rate10m`, которое вы должны заметить и понять, что это выражение и результатирующее правило записи не имеют смысла. Правильное соотношение может выглядеть так:

```
- record: job_mountpoint:node_filesystem_avail_bytes_per_
        node_filesystem_size_bytes:ratio
expr: >
(
    job_mountpoint:node_filesystem_avail_bytes:sum{job="node"}
/
    job_mountpoint:node_filesystem_size_bytes:sum{job="node"}
)
```

Здесь видно, что числитель и знаменатель имеют одинаковый уровень и операции, которые переносятся в имя выходной метрики¹. Здесь операция `sum` удалена, так как она ни о чем не говорит. Этого бы не произошло, если бы во входных метриках была операция `rate5m`.

Предыдущая нотация для определения средних значений размеров событий выглядит немного громоздкой, поэтому здесь я сохранил имя метрики, а в качестве операции использовал функцию `mean5m`, которая основана на `rate5m` и, соответственно, вычисляет среднее значение за 5 мин:

```
- record: job_instance:go_gc_duration_seconds:mean5m
expr: >
(
    job_instance:go_gc_duration_seconds_sum:rate5m{job="prometheus"}
/
    job_instance:go_gc_duration_seconds_count:rate5m{job="prometheus"}
)
```

Если далее вы увидите правило

```
- record: job:go_gc_duration_seconds:mean5m
expr:
  avg without(instance)(
    job_instance:go_gc_duration_seconds:mean5m{job="prometheus"})
)
```

то сразу сможете сказать, что оно пытается получить среднее значение от среднего значения, что не имеет смысла. Правильным будет агрегировать:

```
- record: job:go_gc_duration_seconds:mean5m
expr:
(
  sum without(instance)(
    job_instance:go_gc_duration_seconds_sum:rate5m{job="prometheus"})
)
/
  sum without(instance)(
    job_instance:go_gc_duration_seconds_count:rate5m{job="prometheus"})
)
```

Для агрегирования необходимо суммировать и выполнять деление для усреднения только на последнем шаге вычислений.

Предыдущие случаи просты, как и имена метрик в целом, но если вы сойдетесь с проторенной дорожки, то именование правил записи может быстро превратиться из науки в искусство. Вы должны стремиться ясно и четко определять семантику и метки в именах своих правил, а также однозначно указывать связь правил записи с оригинальными метриками.

¹ Вы могли бы удалить суффикс `_bytes`, который аннулируется, но это может затруднить поиск оригинальных метрик в исходном коде.

Кроме очень редких исключений (см. раздел «Правила для API» выше), имена правил должны четко идентифицировать метрики, чтобы можно было понять, что это такое. Имена метрик не следует использовать как способ хранения аннотаций для политики.

Например, не поддавайтесь искушению добавить `:federate` или `:longterm` или другой подобный суффикс к именам метрик, чтобы сообщить, что такая-то и такая-то метрика передаются в другую систему. Это приводит к раздуванию имен метрик и вызывает проблемы при изменении политики. Лучше определяйте и внедряйте свои политики с помощью сопоставителей при извлечении данных, например извлекая все имена метрик, соответствующие `job:.*`, вместо того чтобы пытаться показать, какие именно метрики будут извлекаться, а какие нет. К тому времени, когда метрика проходит через правило записи, она, скорее всего, будет достаточно агрегирована и иметь незначительную кардинальность, поэтому не стоит тратить время на дополнительную оптимизацию для экономии ресурсов ниже по течению.

Теперь, показав, как использовать правила записи, в следующей главе мы продемонстрируем правила оповещения. Правила оповещения также объединяются в группы и имеют аналогичный синтаксис.

Часть V

ОПОВЕЩЕНИЕ

Если вы хотите, чтобы ваша система мониторинга разбудила вас в 3 часа ночи¹, то эти главы для вас.

Опираясь на предыдущую главу, в главе 18 мы расскажем о правилах оповещения в Prometheus, которые предлагают формировать уведомления в самых разных ситуациях – не только при пересечении метрикой некоторого порогового значения.

Когда Prometheus принимает решение о запуске оповещений, Alertmanager преобразует их в уведомления, пытаясь сгруппировать и ограничить уведомления, чтобы увеличить ценность каждого из них, как описано в главе 19.

¹ Конечно же, только если произойдет какое-то чрезвычайное событие.

Глава 18

Оповещение

Вернемся к разделу «Что такое мониторинг?» главы 1. В нем мы заявили, что оповещение – это один из компонентов мониторинга, позволяющий уведомить человека о возникновении проблемы. Prometheus позволяет определять условия в форме выражений PromQL, которые постоянно вычисляются, и любые получившиеся в результате временные ряды становятся уведомлениями. В этой главе мы покажем, как настроить *оповещение* в Prometheus.

Как показано в примере в разделе «Оповещение» главы 2, Prometheus не несет ответственности за отправку уведомлений, таких как электронные письма, сообщения в чат или короткие сообщения SMS. Этую задачу решает диспетчер уведомлений *Alertmanager*.

Prometheus – это место, где находится логика, которая определяет, требует ли оповещения та или иная ситуация. Как только логика в Prometheus решит, что требуется сгенерировать предупреждение, оно передается диспетчеру уведомлений Alertmanager, который может принимать предупреждения от множества серверов Prometheus. Далее Alertmanager группирует оповещения и отправляет вам конкретные уведомления (рис. 18.1).



Рис. 18.1 ♦ Архитектура Prometheus и Alertmanager

Архитектура, изложенная на рис. 18.1, обеспечивает не только гибкость, но и позволяет отправить только одно уведомление на основе оповещений, полученных от нескольких серверов Prometheus. Например, в случае появления проблемы с передачей служебных данных во всех центрах обработки данных можно настроить группировку предупреждений и создать только

одно уведомление, чтобы не отвлекать операторов множеством уведомлений от каждого имеющегося у вас центра обработки данных.

Правила оповещения

Правила оповещения аналогичны правилам записи, рассмотренным в главе 17. Правила оповещения точно так же объединяются в группы правил и могут смешиваться и сочетаться по вашему усмотрению. Например, нередко все правила оповещения, касающиеся конкретного задания, помещаются в одну группу¹:

```
groups:  
- name: node_rules  
rules:  
- record: job:up:avg  
expr: avg without(instance)(up{job="node"})  
- alert: ManyInstancesDown  
expr: job:up:avg{job="node"} < 0.5
```

Это правило определяет оповещение с именем `ManyInstancesDown`, которое активируется, когда выясняется, что более половины экспортёров Node Exporter не работают. О том, что это правило оповещения, свидетельствует поле `alert`.

В этом примере мы использовали `without`, а не `by`, чтобы сохранить любые другие метки, имеющиеся во временных рядах, и передать их диспетчеру уведомлений Alertmanager. Знание деталей, касающихся задания, окружения и кластера, может очень пригодиться тому, кто получит уведомление.

Обычно в правилах записи следует избегать фильтрации в выражениях, потому что появление и исчезновение временных рядов может вызвать дополнительные сложности. Но в правилах оповещения фильтрация необходима. Если выражение, вычисляемое правилом оповещения, возвращает пустой мгновенный вектор, то никакие оповещения не будут сгенерированы, но если вектор будет содержать какие-либо образцы, то каждый из них станет оповещением.

Вследствие этого единственное правило оповещения, такое как

```
- alert: InstanceDown  
expr: up{job="node"} == 0
```

автоматически применяется к каждому экземпляру в задании `node`, возвращаемому механизмом обнаружения служб, и если у вас обнаружится сотня отключившихся экземпляров, то правило генерирует сотню оповещений. Если к следующему циклу опроса некоторые из этих экземпляров восстано-

¹ Если группа окажется слишком большой, чтобы все ее правила можно было обработать за один интервал, то вам может понадобиться разделить ее, если сократить ее не представляется возможным.

вят работоспособность, то соответствующие им оповещения будут считаться *устраненными*.

Оповещения идентифицируются между циклами вычисления по их меткам, за исключением метки `__name__` с именем метрики, но включая метку `alertname` с именем оповещения.

Помимо отправки оповещений диспетчеру Alertmanager, правила оповещения также будут заполнять две метрики: ALERTS и ALERTS_FOR_STATE. В дополнение к меткам оповещения в ALERTS также добавляется метка `alert-state`. Она будет иметь значение `firing` для активных оповещений и `pending` для ожидающих, как описано в разделе «`for`» ниже. Образцы устранимых оповещений не добавляются в ALERTS. Метрику ALERTS можно использовать в правилах оповещения подобно любой другой метрике, но мы рекомендуем проявлять сдержанность, потому что манипулирование этой метрикой может указывать на чрезмерное усложнение конфигурации.

Значение ALERT_FOR_STATE – это отметка времени начала оповещение. Эта метрика используется внутри Prometheus для восстановления состояния предупреждений после перезапуска.



Правильная обработка устранимых оповещений в ALERTS зависит от того, запускается ли оповещение из одного и того же правила. Если в группе правил имеется несколько оповещений с одним и тем же именем и данное оповещение от нескольких правил, то иногда может проявляться странное поведение ALERTS¹.

Имейте в виду, что Alertmanager не поддерживает маршрутизацию на основе времени. Поэтому, чтобы ограничить рассылку уведомлений, соответствующих оповещениям, определенным временем суток, можно использовать функции для работы с датой и временем, описанные в разделе «Функции minute, hour, day_of_week, day_of_month, day_of_year, days_in_month, month и year» главы 16. Например:

```
- alert: ManyInstancesDown
expr: >
(
    avg without(instance)(up{job="node"}) < 0.5
    and on()
        hour() >= 9 < 17
)
```

Это оповещение будет активироваться только с 9:00 до 17:00 UTC. Для объединения нескольких условий оповещения обычно используется оператор `and`, как обсуждалось в разделе «Оператор and» главы 15. Здесь мы использовали `on()`, так как два операнда `and` не имеют общих меток, что случается довольно редко.

Для пакетных заданий обычно бывает желательно оповестить о неудачной попытке выполнить их:

¹ Это также относится к правилам записи, но наличие нескольких правил записи с одинаковым именем метрики в группе – большая редкость.

```
- alert: BatchJobNoRecentSuccess
expr: >
  time() - my_batch_job_last_success_time_seconds{job="batch"} > 86400*2
```

Как обсуждалось во врезке «Идемпотентность пакетных заданий» в главе 3, идемпотентные пакетные задания позволяют не беспокоиться о единичных сбоях и не отправлять уведомлений о них.

for

Мониторинг на основе метрик предполагает возможность множества состояний гонки: из-за потери сетевого пакета опрос может завершиться по тайм-ауту, вычисление правила может немного задержаться из-за особенностей планирования, и в контролируемых вами системах может возникнуть кратковременный всплеск активности.

Едва ли вам понравится, если вас будут будить среди ночи из-за каждого артефакта или каждой странности в ваших системах; вы наверняка предпочли бы сохранить свою энергию для решения реальных проблем, затрагивающих пользователей. Поэтому активация оповещений на основе результатов проверки только одного правила редко бывает хорошей идеей. Вот тут-то и вступает в игру поле `for` в определениях правил оповещения:

```
groups:
- name: node_rules
rules:
- record: job:up:avg
  expr: avg without(instance)(up{job="node"})
- alert: ManyInstancesDown
  expr: avg without(instance)(up{job="node"}) < 0.5
  for: 5m
```

Поле `for` говорит, что данное оповещение должно возвращаться как минимум в течение указанного времени, прежде чем оно будет активировано. Пока условие `for` не выполнено, оповещение считается ожидающим (`pending`). Ожидающее, но не активированное оповещение не передается диспетчеру уведомлений `Alertmanager`. Список текущих ожидающих и активированных оповещений доступен по адресу <http://localhost:9090/alerts>, как показано на рис. 18.2.

В Prometheus не поддерживаются понятия гистерезиса и «дребезга» для оповещений. Вы должны сами выбрать пороги оповещений так, чтобы имело смысл вызвать человека, даже если проблема впоследствии исчезнет.

Обычно мы рекомендуем использовать в `for` интервал не менее 5 мин. Это избавит от большинства ложных срабатываний. Кто-то из вас может возразить, что это помешает вам немедленно заняться проблемой, но имейте в виду, что вам, скорее всего, потребуется 5 мин, чтобы проснуться, загрузить свой ноутбук, войти в систему, подключиться к корпоративной сети и начать отладку. И даже если вы сидите перед своим компьютером, готовый к работе,

то, как показывает наш опыт, в отложенной системе оповещения обычно свидетельствуют о нетривиальных проблемах, и вам потребуется не менее 20–30 мин, чтобы просто получить представление о происходящем.

The screenshot shows the Prometheus interface with the 'Alerts' tab selected. At the top, there are three filter buttons: 'Inactive (0)', 'Pending (0)', and 'Firing (1)'. Below them is a search bar labeled 'Filter by name or labels' and a checkbox for 'Show annotations'. A red header bar highlights the first alert: 'InstanceDown (2 active)'. The alert configuration is shown as follows:

```

name: InstanceDown
expr: up == 0
for: 5m
  
```

Below the configuration, a table lists the active alerts:

Labels	State	Active Since	Value
alername=InstanceDown, instance=localhost:9100, job=node	PENDING	2022-11-16T09:00:24.499989908Z	0
alername=InstanceDown, instance=otherhost:9100, job=node	FIRING	2022-11-16T08:55:34.499989908Z	0

Рис. 18.2 ❖ Страница с информацией об активированных и ожидающих оповещениях

Желание сразу же приступить к решению каждой проблемы похвально, но большое количество оповещений притупит внимание ваше и вашей команды и значительно снизит вашу эффективность. Если у вас есть оповещение, требующее, чтобы человек выполнил действие менее чем за 5 мин, то вам следует работать над автоматизацией этого действия, потому что подобные ситуации обходятся очень дорого, даже если вы уверенно сможете реагировать на них менее чем за 5 мин.

Также могут иметься оповещения, менее важные или более надоедливые, с которыми предпочтительнее использовать более длительный интервал в поле `for`. Как и в случае с другими длительностями и интервалами, старайтесь не усложнять. Например, как показывает наша практика, для любых оповещений можно ограничиться интервалами 5м, 10м, 30м и 1h, и нет особого смысла заниматься микрооптимизацией, добавляя интервал 12м или 20м.

Так как `for` требует, чтобы правило оповещения возвращало одни и те же временные ряды в течение определенного периода времени, состояние `for` может быть сброшено, если в одном из результатов проверки правила не окажется заданного временного ряда. Например, если вы используете метрику-датчик, извлекаемую непосредственно из цели, то в случае сбоя одного из циклов опроса состояние `for` будет сброшено:

```

- alert: FDsNearLimit
  expr: >
    process_open_fds > process_max_fds * .8
  for: 5m
  
```

Для защиты от подобных ошибок можно использовать функции `_over_time`, описанные в разделе «Агрегирование по времени» главы 16, такие как `avg_over_time`, `last_over_time` и `max_over_time`:

```
- alert: FDsNearLimit
expr:
(
    max_over_time(process_open_fds[5m])
    >
    max_over_time(process_max_fds[5m]) * 0.9
)
for: 5m
```

Метрика `up` уникальна: она присутствует всегда, даже если опрос завершился неудачей, что избавляет от необходимости использовать функцию `_over_time`. Соответственно, если при использовании Blackbox Exporter, как описано в разделе «Blackbox» главы 10, требуется отделить неудачные попытки опроса от неудачных попыток зондирования¹, то можно использовать такое правило:

```
- alert: ProbeFailing
expr: up{job="blackbox"} == 0 or probe_success{job="blackbox"} == 0
for: 5m
```

Метки оповещений

Как и в случае с правилами записи, в правилах оповещения можно указывать метки. В правилах записи метки определяются редко, но это стандартная практика при определении правил оповещения.

При маршрутизации оповещений в Alertmanager, как описано в разделе «Дерево маршрутизации» главы 19, необязательно упоминать имя каждого отдельного оповещения в файле конфигурации Alertmanager. Вместо этого для обозначения намерений можно использовать метки.

Многие обычно определяют метку `severity`, определяющую уровень серьезности и способ доставки уведомления: в виде сообщения SMS, что позволяет уведомить ответственного работника даже ночью, или записи в журнале проблем, если ошибка не требует срочной обработки.

Например, выход из строя одной машины не должен быть чрезвычайной ситуацией, но выход из строя половины машин требует срочного вмешательства:

```
- alert: InstanceDown
expr: up{job="node"} == 0
for: 1h
```

¹ Экспортёр Blackbox Exporter должен возвращать ответ до истечения тайм-аута, но иногда проблемы могут возникнуть из-за медленной работы сети или из-за сбоя в Blackbox Exporter.

```

labels:
  severity: ticket
- alert: ManyInstancesDown
expr: job:up:avg{job="node"} < 0.5
for: 5m
labels:
  severity: page

```

Метка `severity` в этом примере не имеет особого семантического значения; это самая обычная метка, добавленная в оповещение, которую можно использовать при настройке Alertmanager. Добавляя оповещения в Prometheus, старайтесь настроить их так, чтобы для правильной маршрутизации достаточно было одной метки `severity` и только в редких случаях требовалось бы настраивать конфигурацию Alertmanager.

Если есть необходимость отправлять оповещения разным командам, то в дополнение к метке `severity` можно определить метку `team` или `service`. Если Prometheus настроен на отправку оповещений только одной команде, то можно использовать внешние метки (как обсуждается в разделе «Внешние метки» ниже). В правилах оповещения не должно быть необходимости упоминать такие метки, как `env` или `region`; они уже должны иметься в оповещении, являясь целевыми метками, получаемыми в процессе выполнения выражения, или будут добавлены впоследствии с помощью `external_labels`.

Поскольку метки определяют идентичность оповещения, важно, чтобы они не менялись от случая к случаю, иначе такие оповещения не только не будут удовлетворять требованиям поля `for`, но также будут захламлять базу данных временных рядов в Prometheus и Alertmanager.

Prometheus не позволяет определить несколько порогов для оповещения, но вы можете определить несколько оповещений с разными порогами и метками:

```

- alert: FDsNearLimit
expr: >
  process_open_fds > process_max_fds * .95
for: 5m
labels:
  severity: page
- alert: FDsNearLimit
expr: >
  process_open_fds > process_max_fds * .8
for: 5m
labels:
  severity: ticket

```

Обратите внимание, что, если при превышении 95%-го порога занятых файловых дескрипторов сработают оба этих оповещения, попытка активировать только одно из них чревата опасностью, потому что если значение будет колебаться около 95 %, то может не активироваться ни одно из оповещений. Кроме того, оповещение должно активироваться в ситуации, когда вы уже решили, что необходимо потребовать от человека рассмотреть проблему. Если

вы чувствуете, что такие двойные оповещения нежелательны, то попробуйте настроить сами оповещения и прежде всего подумайте, так ли они нужны, а не пытайтесь затолкать джина обратно в бутылку, когда оповещение уже срабатывает.

У каждого оповещения должен быть хозяин

Брайан намеренно не включил в примеры уровни серьезности *email* и *chat*. Давайте позволим ему рассказать свою историю, объясняющую причину такого решения.

Однажды я входил в состав команды, которая вынуждена была создавать новый список рассылки каждые несколько месяцев. У нас имелся список рассылки оповещений по электронной почте, но эти оповещения не всегда привлекали должное внимание, потому что их было слишком много, и порой было неясно, кто конкретно должен реагировать на то или иное оповещение, поэтому чаще всего на них никто не реагировал. Были некоторые оповещения, считавшиеся важными, но не настолько важными, чтобы вызвать дежурного инженера. Поэтому эти оповещения просто рассыпались членам основной команды в надежде, что кто-нибудь их прочитает и предпримет какие-то действия. То же произошло с новым списком рассылки для команды, куда тоже начали регулярно поступать автоматические оповещения. В какой-то момент все стало настолько плохо, что был создан новый список рассылки, и история повторилась. К этому моменту у команды было три списка рассылки оповещений по электронной почте.

Учитывая этот опыт и опыт других, я категорически не одобряю рассылку оповещений по электронной почте и оповещений, отправляемых группе лиц¹. Я сторонник отправки уведомлений в систему регистрации инцидентов, где они будут закреплены за конкретными людьми, чья работа заключается в решении проблем. Я также видел, как обрабатывается ежедневная электронная рассылка членам оперативной группы, в которой перечислены все текущие оповещения.

После сбоя все виноваты в том, что не просматривают оповещения по электронной почте², но никто не несет ответственности. Поэтому очень важно, чтобы уведомления посыпались конкретным ответственным лицам, а не просто всем перечисленным в списке рассылки по электронной почте.

То же относится к оповещениям, посыпаемым в системы обмена мгновенными сообщениями, такие как IRC, Slack и Telegram. Рассылка важных уведомлений через систему обмена сообщениями привлекает внимание, и такие уведомления посыпаются нечасто. Но рассылка маловажных уведомлений влечет те же проблемы, что и рассылка оповещений по электронной почте, и даже хуже, потому что сильно отвлекает. Вы не можете фильтровать сообщения, например SMS, помещая их в папку со спамом, как это часто делается с электронной почтой.

¹ Я также категорически против любой формы рассылки сообщений, не написанных рукой человека, в том числе оповещений, запросов на включение и отчетов об ошибках/проблемах.

² Среди тысяч маловажных оповещений, которые все игнорировали, неизменно находилось одно, предвещавшее сбой. Но, чтобы обнаружить это электронное письмо, пришлось бы исследовать тысячи уведомлений, не связанных с проблемой.

Аннотации и шаблоны

Метки идентифицируют оповещения, поэтому их нельзя использовать для передачи дополнительных сведений, таких как текущее значение, которое может меняться между циклами опроса. Вместо меток для этой цели можно использовать *аннотации оповещений*, которые подобны меткам и могут использоваться в оповещениях. Однако аннотации не участвуют в идентификации оповещений, поэтому их нельзя использовать для группировки и маршрутизации в Alertmanager.

Поле `annotations` позволяет передать дополнительную информацию об оповещении, например краткое описание проблемы. Кроме того, в значении поля `annotations` можно использовать шаблоны Go (<https://oreil.ly/x0tjn>). Это позволяет отформатировать результат запроса и даже выполнить дополнительные запросы PromQL, чтобы добавить дополнительную информацию.

Prometheus не передает значения оповещений в Alertmanager. Поскольку Prometheus позволяет использовать в правилах предупреждений все возможности PromQL, нет никакой гарантии, что значение будет полезным или даже значимым для пользователя. Метки определяют оповещение, а не значение, и оповещения могут быть чем-то большим, чем просто пороговое значение для некоторого временного ряда.

В аннотациях, например, в отличие от меток, можно показать количество работающих экземпляров в процентах. В системе шаблонов Go непросто выполнять математические операции, но необходимое значение можно подготовить в выражении предупреждения¹:

```
groups:
  - name: node_rules
    rules:
      - alert: ManyInstancesDown
        for: 5m
        expr: avg without(instance)(up{job="node"}) * 100 < 50
        labels:
          severity: page
        annotations:
          summary: 'Only {{printf "%2f" $value}}% of instances are up.'
```

Здесь `$value` представляет значение оповещения. Оно передается в вызов функции `printf`² для форматирования. Фигурные скобки обозначают шаблонные выражения.

Помимо `$value`, имеется также `$labels` с метками оповещения. Например, `$labels.job` вернет значение метки `job`.

Выполнять запросы в шаблонах аннотаций можно с помощью функции `query`. После этого нередко бывает нужно выполнить обход результатов запроса; это можно реализовать с помощью функции `range` – аналога цикла `for`:

¹ В особенно сложных случаях можно воспользоваться оператором `and` со значением для шаблона слева стороны и выражением оповещения справа.

² На самом деле это функция `sprintf`, потому что возвращает результат, а не выводит его. Она позволяет создавать запросы для передачи функции `query`.

```
- alert: ManyInstancesDown
  for: 5m
  expr: avg without(instance)(up{job="node"}) < 0.5
  labels:
    severity: page
  annotations:
    summary: 'More than half of instances are down.'
    description: >
      Down instances: {{ range query "up{job=\"node\"} == 0" }}{{ .Labels.instance }}{{ end }}
```

Значение элемента в теле цикла представлено точкой (.). Соответственно, .Labels – это метки текущего образца из мгновенного вектора, а .Labels.instance – это метка instance образца. .Value представляет значение образца.

 Правила оповещений оцениваются независимо в каждом цикле опроса. Поэтому, если правило содержит тяжелый шаблон, создающий сотни оповещений, оно может вызывать проблемы с производительностью.

В аннотациях можно также передавать статические значения, например ссылки на полезные дашборды или документацию:

```
- alert: InstanceDown
  for: 5m
  expr: up{job="prometheus"} == 0
  labels:
    severity: page
  annotations:
    summary: 'Instance {{$labels.instance}} of {{$labels.job}} is down.'
    dashboard: http://some.grafana:3000/dashboard/db/prometheus
```

В зрелой системе передача в оповещении всей возможной отладочной информации будет не только тормозить и сбивать с толку дежурного оператора, но и, вероятно, окажется не особенно полезна, кроме самых простых случаев. Аннотации оповещений следует рассматривать в первую очередь как указатель, показывающий правильное направление для начала отладки. В дашборде можно получить намного более подробную и актуальную информацию, чем в нескольких строках оповещения.

Шаблоны уведомлений (см. раздел «Шаблоны уведомлений» главы 19) – еще один уровень шаблонов, поддерживаемый в Alertmanager. Шаблоны уведомлений можно рассматривать как электронные письма с несколькими полями, которые необходимо заполнить, а шаблоны оповещений в Prometheus – как значения для этих полей.

Например, вы можете решить сопроводить каждое из своих оповещений планом действий¹ для включения в уведомление и, вероятно, создадите вики-страницы с описанием оповещений. Вы можете добавить аннотацию wiki в каждое оповещение, но, поймав себя на том, что каждый раз вы добавляете

¹ План действий – это документ, описывающий шаги, которые необходимо предпринять в ответ на определенный тип инцидента.

одну и ту же аннотацию в каждое правило оповещения, подумайте о возможности использовать для этого шаблоны уведомлений в Alertmanager. Alertmanager уже знает имя оповещения, поэтому он с легкостью сконструирует ссылку `wiki.yourcompany/Alertname` и избавит вас от необходимости повторять правила оповещений. Как и повсюду в управлении конфигурацией и мониторингом, следование непротиворечивым соглашениям упрощает жизнь.

- i** В поле `labels` в правилах оповещения тоже можно использовать шаблоны, как и в поле `annotations`, но эта возможность может пригодиться только в особенно сложных ситуациях. На практике же для меток почти всегда используются простые статические значения. Если вы решите определять метки с помощью шаблонов, то не забывайте, что значения меток не должны меняться между циклами опроса.

Что такое хорошее оповещение?

При мониторинге в стиле Nagios принято предупреждать о потенциальных проблемах, таких как высокая средняя нагрузка, большое потребление процессорного времени или сбой запуска процесса. Все это потенциальные причины проблем, но они не всегда указывают на проблему, требующую срочного вмешательства человека.

По мере того как системы становятся все более сложными и динамичными, создавать оповещения для каждой возможной проблемы становится все труднее. Даже если вам удастся это сделать, количество ложных срабатываний будет настолько велико, что в конечном итоге вы и ваша команда пропустите настоящие проблемы, спрятанные в потоке маловажных уведомлений.

Лучшее решение – предупреждать о *симптомах*. Вашим пользователям все равно, насколько высока средняя нагрузка; их больше заботит, чтобы видеосервисы с котиками загружались достаточно быстро. Имея оповещения для таких метрик, как задержки и сбои, с которыми столкнулись пользователи¹, вы сможете выявлять действительно важные проблемы, а не просто какие-то признаки, которые могут указывать на проблему.

Например,очные задания могут вызвать всплеск потребления процессорного времени, но в это время количество пользователей невелико, и у вас, вероятно, не будет проблем с их обслуживанием. С другой стороны, создать оповещение для периодической потери пакетов может быть сложно, но эти потери довольно четко отражаются на метриках задержки. Если у вас есть соглашения об уровне обслуживания (Service-Level Agreements, SLA) с вашими пользователями, то в них вы найдете метрики для создания оповещений и хорошие начальные пороговые значения. Также желательно иметь оповещения для обнаружения проблем с потреблением ресурсов, таких как исчерпание дисковой квоты, а также для проверки работы системы мониторинга².

¹ Пользователи не обязательно должны быть клиентами вашей компании. Это могут быть сотрудники компании, если, к примеру, вы управляете какой-то внутренней службой.

² Мы подробно продемонстрируем это в разделе «Мета- и перекрестный мониторинг» главы 21.

В идеале каждое сообщение дежурному оператору и каждое уведомление, регистрируемое в системе инцидентов, должны требовать вмешательства человека. Если вмешательства человека не требуется, то такое оповещение – главный кандидат для автоматизации. Поскольку на исправление нетрииальной проблемы может уйти несколько часов, вы должны стремиться уведомлять не более, чем о двух инцидентах в день. Это не относится к несрочным оповещениям, поступающим в систему регистрации инцидентов, но все же нежелательно создавать слишком много записей.

Если обнаружится, что на инциденты вы отвечаете словами «проблема не наблюдается», такие оповещения вообще не должны появляться. Для таких оповещений подумайте об увеличении порога срабатывания, чтобы сделать их менее чувствительными, или, может быть, вообще удалите их.

Для дополнительного обсуждения подходов к оповещению и к управлению системами мы рекомендуем прочитать статью Роба Эващук (Rob Ewaschuk) «My Philsophy on Alerting» (<https://oreil.ly/WYPVf>). Роб также написал главу 6 книги «Site Reliability Engineering»¹ (Betsy Beyer et al, eds., O'Reilly), в которой содержатся более общие советы по управлению системами.

Настройка диспетчеров уведомлений в Prometheus

Настройка списка диспетчеров уведомлений Alertmanager в Prometheus производится с использованием той же конфигурации обнаружения служб, которая описана в главе 8. Например, для настройки одного локального Alertmanager может иметься файл *prometheus.yml* с таким содержимым:

```
global:
  scrape_interval: 10s
  evaluation_interval: 10s
alerting:
  alertmanagers: ①
    - static_configs:
      - targets: ['localhost:9093']
rule_files:
  - rules.yml
scrape_configs:
  - job_name: node
    static_configs:
      - targets:
        - localhost:9100
  - job_name: prometheus
    static_configs:
```

¹ Мерфи Нейл Ричард, Бейер Бетси. Site Reliability Engineering. Надежность и безотказность как в Google. СПб.: Питер, 2019. ISBN: 978-5-4461-0976-0. – Прим. перев.

```
- targets:
  - localhost:9090
```

❶ Этот раздел содержит настройки обнаружения диспетчеров Alertmanager.

Поле `alertmanagers` во многом похоже на раздел с настройками опроса, но здесь нет `job_name`, и инструкции по изменению меток, влияющие на их имена, не имеют значения, потому что диспетчеры уведомлений не поддерживают понятия целевых меток, соответственно, действовать будут только инструкции `drop` и `keep`.

У вас может быть настроено несколько диспетчеров Alertmanager, о чем подробно рассказывается в разделе «Кластеризация диспетчеров уведомлений» главы 21. В таком случае Prometheus будет посыпать все оповещения всем настроенным диспетчерам.

Поле `alerting` также содержит параметр `alert_relabel_configs`, определяющий порядок изменения меток, как описано в разделе «Изменение меток» главы 8, но применяется к меткам оповещений. Вы можете реорганизовать метки оповещений или даже удалять оповещения. Например, вы можете создавать информационные оповещения, не покидающие пределы Prometheus:

```
alerting:
  alertmanagers:
    - static_configs:
      - targets: ['localhost:9093']
  alert_relabel_configs:
    - source_labels: [severity]
      regex: info
      action: drop
```

Здесь же можно добавить метки `env` и `region` ко всем оповещениям и избавить себя от хлопот определять их в другом месте, но есть лучший способ сделать это – `external_labels`.

Внешние метки

Внешние метки – это метки, применяемые по умолчанию, когда Prometheus общается с другими системами, такими как Alertmanager, использует федерацию, осуществляет удаленное чтение и запись¹, но не с HTTP API. Внешние метки идентифицируют сервер Prometheus, и каждый сервер Prometheus в вашей организации должен иметь уникальные внешние метки. `external_labels` является частью раздела `global` в `prometheus.yml`:

```
global:
  scrape_interval: 10s
  evaluation_interval: 10s
  external_labels:
```

¹ Как описано в разделах «Выход на глобальный уровень с помощью механизма федерации» и «Долгосрочное хранение» главы 21.

```
region: eu-west-1
env: prod
team: frontend
alerting:
  alertmanagers:
    - static_configs:
      - targets: ['localhost:9093']
```

Удобнее всего определять в `external_labels` такие метки, как `region`, которые применяются ко всем опрашиваемым целям, требуют помнить о них при написании запросов PromQL или добавлять их в каждое правило оповещения в Prometheus. Такой подход экономит время и силы, а также упрощает совместное использование правил записи и оповещения на разных серверах Prometheus, поскольку они не привязаны к конкретному окружению или к одной организации. Если потенциальная внешняя метка меняется внутри Prometheus, то, скорее всего, она должна быть целевой меткой.

Поскольку внешние метки применяются после оценки правил оповещения¹, они недоступны в шаблонах оповещений, потому что оповещения не должны зависеть от того, на каком из серверов Prometheus они оцениваются. Диспетчер оповещений Alertmanager будет иметь доступ к внешним меткам в своих шаблонах уведомлений, как и к любым другим меткам, и шаблоны – самое подходящее место для работы с ними.

Внешние метки используются только по умолчанию; если один из временных рядов уже имеет метку с таким именем, то одноименная внешняя метка не будет применяться. Поэтому мы рекомендуем не создавать целевые метки с именами, совпадающими с внешними метками.

Теперь, узнав, как настроить оповещения в Prometheus, перейдем к следующей главе, где рассказывается, как настроить диспетчер уведомлений, который преобразует оповещения в уведомления.

¹ Применение действий в `alert_relabel_configs` происходит после действий в `external_labels`.

Глава 19

Alertmanager

В главе 18 вы видели, как определить в Prometheus правила отправки оповещений в Alertmanager. Alertmanager должен принимать все оповещения от всех серверов Prometheus и преобразовывать их в уведомления, такие как электронные письма, сообщения в чате или мгновенные сообщения. В главе 2 мы дали краткое введение в Alertmanager, и теперь в этой главе мы подробнее расскажем о всех его возможностях.

Конвейер уведомлений

Alertmanager не просто преобразует оповещения в уведомления. В идеальном мире для каждого производственного инцидента посыпается ровно одно уведомление, и Alertmanager пытается обеспечить это, предоставляя управляемый конвейер для обработки оповещений и превращения их в уведомления. Как мы уже знаем, в основе Prometheus лежат метки, но они также являются ключевым компонентом Alertmanager, обеспечивая описываемые далее возможности.

Подавление

Иногда, даже при использовании оповещений на основе симптомов, может потребоваться запретить отправку уведомлений, если активируется другое, более серьезное оповещение. Например, можно запретить отправку оповещения о неработоспособности службы, если центр обработки данных, в котором она находится, оказался недоступным. В таких случаях применяется *подавление*.

Выключение

Если вы уже знаете о проблеме или останавливаете службу для профилактического обслуживания, то нет смысла сообщать об этом дежурному оператору. Функция *выключения* позволяет игнорировать определенные оповещения в течение некоторого времени и настраивается через веб-интерфейс Alertmanager.

Маршрутизация

Предполагается, что в каждой организации будет запускаться один диспетчер уведомлений Alertmanager, но иногда разные команды могут по желать, чтобы их уведомления доставлялись в разные места; и даже внутри команды может потребоваться, чтобы оповещения для рабочего окружения и окружения разработки обрабатывались по-разному. Alertmanager позволяет настроить это с помощью дерева маршрутизации.

Группировка

Далее после выбора маршрута для оповещений вашей команды было бы излишне отправлять отдельное уведомление для каждой машины в стойке¹, которая вышла из строя. Чтобы избежать этого, в Alertmanager предусмотрена возможность группировки оповещений и отправки только одного уведомления на стойку, на центр обработки данных или даже одного глобального уведомления о недоступности машин.

Регулирование частоты и повторение

Представьте, что у вас есть группа оповещений, активируемых, когда стойка с машинами выходит из строя, и уже после отправки соответствующего уведомления приходит оповещение о недоступности одной из машин в этой стойке. Если бы Alertmanager отправлял новое уведомление каждый раз, когда получает новое оповещение из группы, это сделало бы группировку бессмысленной. Вместо этого Alertmanager будет ограничивать частоту отправки уведомлений для данной группы, чтобы избавить вас от лишнего шума.

В идеальном мире все уведомления обрабатываются быстро и без задержек, но в реальности дежурный оператор или другая система могут упустить проблему из виду. Чтобы этого не случилось, Alertmanager будет повторять отправку уведомлений.

Уведомление

После преодоления этапов подавления, отключения, маршрутизации, группировки и ограничения оповещения наконец попадают на этап отправки уведомлений через *приемник*. Для уведомлений поддерживаются шаблоны, что позволяет настраивать их содержимое и выделять важные детали.

Конфигурационный файл

Как и все другие настройки, которые вы видели в этой книге, настройки Alertmanager сохраняются в файле YAML, обычно с именем *alertmanager.yml*. Обновленную конфигурацию можно перезагрузить во время выполнения,

¹ В центрах обработки данных машины обычно размещаются в вертикальных стойках, причем каждая стойка часто имеет свою схему питания и сетевой коммутатор. Поэтому из-за проблем с питанием или коммутатором недоступными становятся сразу все машины в стойке.

отправив сигнал `SIGHUP` или HTTP-запрос `POST` в конечную точку `/-/reload`. Для проверки ошибок в конфигурационном файле `alertmanager.yml` можно использовать команду `amtool check-config`¹.

Например, вот минимальная конфигурация, которая отправляет все уведомления по электронной почте с использованием локального SMTP-сервера:

```
global:
  smtp_smarthost: 'localhost:25'
  smtp_from: 'yourprometheus@example.org' ①

route:
  receiver: example-email

receivers:
  - name: example-email
    email_configs:
      - to: 'youraddress@example.org' ②
```

① Адрес электронной почты для поля `От` (`From`).

② Адрес электронной почты, куда будут отправляться электронные письма.

У вас всегда должен быть определен хотя бы один маршрут и один получатель. Для разных типов приемников имеется множество самых разных глобальных настроек, но почти всегда их оставляют со значениями по умолчанию. Далее мы рассмотрим некоторые другие части конфигурационного файла. Полный файл `alertmanager.yml`, включающий все примеры из этой главы, можно найти на GitHub (<https://oreil.ly/hQduB>).

Дерево маршрутизации

Поле `route` задает маршрут по умолчанию (он же *резервный* маршрут). Маршруты образуют дерево, поэтому вслед за маршрутом по умолчанию обычно определяется еще несколько маршрутов. Например:

```
route:
  receiver: fallback-pager
  routes:
    - matchers:
        - severity = page
        receiver: team-pager
    - matchers:
        - severity = ticket
        receiver: team-ticket
```

Проверка поступившего оповещения начинается с маршрута по умолчанию, и производится попытка сопоставления с первым *дочерним маршрутом*,

¹ `amtool` также можно использовать для запроса оповещений и управления их отключением.

который определен в (возможно, пустом) поле `routes`. Если оповещение имеет метку, точно соответствующую `severity="page"`, то ему назначается этот маршрут, и на этом обход дерева прекращается, поскольку у этого маршрута нет дочерних маршрутов.

Если оповещение не имеет метки `severity="page"`, то проверяется соответствие следующему дочернему маршруту, в данном случае с меткой `severity="ticket"`. Если оповещение соответствует этому маршруту, то обход дерева также прекращается. Иначе, если не найдено совпадение ни с одним дочерним маршрутом, процедура сопоставления возвращается вверх по дереву и выбирается маршрут по умолчанию. Это называется *обходом дерева в обратном порядке*, когда дочерние элементы проверяются до их родителя, и результатом становится первое совпадение.

Помимо оператора `=` в выражениях сопоставления, можно использовать другие операторы, такие как `!=`, `=~` и `!~`. Оператор `=~` требует совпадения значения данной метки с регулярным выражением, а `!~`, напротив, требует несовпадения. Как и во многих¹ других местах в Prometheus, регулярные выражения привязываются к началу и концу строки. Дополнительные сведения о регулярных выражениях ищите в разделе «Регулярные выражения» главы 8.

Если разные команды передают в метке `severity` разные значения, такие как `ticket`, `issue` и `email` для передачи оповещения по одному и тому же маршруту, то для сопоставления можно использовать оператор `=~`:

```
route:
  receiver: fallback-pager
  routes:
    - matchers:
        - severity = page
      receiver: team-pager
    - matchers:
        - severity =~ "(ticket|issue|email)"
      receiver: team-ticket
```

В определении одного маршрута можно задать несколько сопоставителей; для совпадения с таким маршрутом оповещения должны удовлетворять всем условиям, перечисленным в сопоставителях.

 Все оповещения должны соответствовать какому-то маршруту, при этом маршрут верхнего уровня проверяется последним. Он действует как маршрут по умолчанию, и ему должны соответствовать все оповещения, поэтому использование сопоставителей `matchers` в маршруте по умолчанию является ошибкой.

Диспетчер Alertmanager редко используется одной командой, но разные команды могут потребовать отправлять их оповещения по-разному. Для таких случаев в вашей организации должна иметься стандартная метка, такая

¹ Функция `reReplaceAll` в шаблонах предупреждений и уведомлений (<https://oreil.ly/heUJc>) не привязывает регулярное выражение к началу и концу строки, так как это противоречит ее назначению.

как `team` или `service`, определяющая принадлежность оповещений. Эта метка не всегда исходит из `external_labels`, как обсуждалось в разделе «Внешние метки» главы 18. Используя метку, такую как `team`, можно определить маршруты с индивидуальными конфигурациями для каждой команды:

```
route:
  receiver: fallback-pager
  routes:
    # Команда пользовательского интерфейса.
    - matchers:
        - team = frontend
      receiver: frontend-pager
    routes:
      - matchers:
          - severity = page
        receiver: frontend-pager
      - matchers:
          - severity = ticket
        receiver: frontend-ticket
    # Команда серверной части.
    - matchers:
        - team = backend
      receiver: backend-pager
    routes:
      - matchers:
          - severity = page
          - env = dev
        receiver: backend-ticket
      - matchers:
          - severity = page
        receiver: backend-pager
      - matchers:
          - severity = ticket
        receiver: backend-ticket
```

Команда пользовательского интерфейса имеет простую конфигурацию: срочные оповещения отправляются на пейджер, а заявки – в систему регистрации инцидентов, а все оповещения с неожиданными значениями метки `severity` отправляются на пейджер.

Команда серверной части действует более избирательно. Любые срочные оповещения из окружения разработки отправляются в приемник `backend-ticket`, т. е. их статус понижается до рядовых проблем, не требующих срочного вмешательства¹. Таким способом можно по-разному маршрутизировать оповещения из разных окружений и избавить себя от необходимости настраивать правила оповещения для каждого окружения. Такой подход позволяет вариировать только метки `external_labels`.

¹ Именование приемника – это просто соглашение, но если под именем `backend-ticket` скрывается нечто иное, отличное от системы регистрации проблем, то такая конфигурация может ввести в заблуждение.

i Иногда разобраться в существующем дереве маршрутизации может быть довольно сложно, особенно если оно не соответствует стандартной структуре. На веб-сайте Prometheus есть визуальный редактор деревьев маршрутизации (<https://oreil.ly/KtvK>), который может показать дерево в графическом виде и как оповещения будут следовать по нему.

Такая конфигурация имеет свойство расширяться с увеличением числа команд, поэтому можно написать утилиту для объединения фрагментов дерева маршрутизации из файлов меньшего размера. YAML – это стандартный формат, и для него доступно множество готовых средств парсинга, так что создание такой утилиты – несложная задача.

Еще один параметр, о котором мы должны упомянуть в контексте маршрутизации – `continue`. Обычно выбирается первый совпадший маршрут, но если указан параметр `continue: true`, то найденное совпадение не остановит процесс поиска. Совпадший маршрут с параметром `continue` будет запомнен, и процесс проверки последующих маршрутов продолжится. Таким способом можно организовать отправку оповещения по нескольким маршрутам. Параметр `continue` в основном используется для регистрации всех оповещений в некоторой другой системе:

```
route:
  receiver: fallback-pager
  routes:
    # Зарегистрировать все оповещения.
    - receiver: log-alerts
      continue: true
    # Команда пользовательского интерфейса.
    - matchers:
      - team = frontend
    receiver: frontend-pager
```

После выбора маршрута к оповещению применяются правила группировки, ограничения частоты и повторения, а также приемник, соответствующие этому маршруту. Все настройки дочерних маршрутов наследуются по умолчанию от родительского маршрута, за исключением `continue`.

Группировка

После определения маршрута Alertmanager помещает все оповещения, соответствующие конкретному маршруту, в одну группу, т. е. объединяет их в одно большое уведомление. В некоторых случаях это может быть нормально, но иногда желательно, чтобы уведомления имели меньший размер.

Поле `group_by` позволяет указать список меток для группировки оповещений. Группировка в этом случае осуществляется так же, как при использовании оператора `by` в функциях агрегирования (обсуждается в разделе «`by`» главы 14). Обычно требуется разделить оповещения по одному или нескольким параметрам: имени оповещения, окружению и/или местоположению.

Проблема в промышленном окружении вряд ли будет связана с проблемой в окружении разработки, а также с проблемами в других центрах обработки

данных. При оповещении о симптомах, а не о причинах, как рекомендуется в разделе «Что такое хорошее оповещение?» главы 18, вполне вероятно, что разные оповещения будут связаны с разными инцидентами¹.

На практике конфигурация с группировкой может выглядеть примерно так:

```
route:
  receiver: fallback-pager
  group_by: [team]
  routes:
    # Команда пользовательского интерфейса.
    - matchers:
        - team = frontend
      group_by: [region, env]
      receiver: frontend-pager
      routes:
        - matchers:
            - severity = page
          receiver: frontend-pager
        - matchers:
            - severity = ticket
          group_by: [region, env, alertname]
          receiver: frontend-ticket
```

Здесь оповещения, для которых выбирается маршрут по умолчанию, группируются по метке `team`, соответственно, оповещения для команд, не определяющих свой маршрут, можно рассылать отдельно. Команда пользовательского интерфейса решила сгруппировать оповещения по меткам `region` и `env`. Эта настройка `group_by` будет унаследована дочерними маршрутами, поэтому все их записи в системе регистрации инцидентов и срочные оповещения тоже будут группироваться по регионам и окружениям.

Группировать по метке `instance` обычно не рекомендуется, так как может возникнуть большое количество уведомлений в случае появления проблемы, затрагивающей все приложение. Однако для отправки оповещений об отключении компьютеров, чтобы создать заявки на физическую проверку их человеком, группировка по `instance` может иметь смысл.

i Группировку оповещений в Alertmanager можно запретить, определив параметр `group_by` со значением `[...]2`. Однако группировка – это благо, потому что уменьшает количество рассылаемых уведомлений и позволяет более целенаправленно реагировать на инциденты. Пропустить важное уведомление сложнее, если сообщений несколько, а не несколько сотен³.

Если вы решите отключить группировку, потому что в вашей организации уже есть некоторый механизм, играющий роль Alertmanager, то вам, возможно, лучше отказаться от Alertmanager и работать с оповещениями, отправляемыми из Prometheus.

¹ С другой стороны, согласно методу RED высокая частота отказов и высокая задержка могут возникать одновременно. На практике один симптом обычно проявляется намного раньше другого, и у вас остается достаточно времени, чтобы решить проблему.

² Это список YAML со строкой из трех точек.

³ Сотня сообщений – это довольно много.

Регулирование частоты и повторение

При отправке сгруппированных уведомлений иногда желательно ограничить отправку новых уведомлений при смене набора условий, приведших к оповещению, поскольку это слишком отвлекает внимание. С другой стороны, не хотелось бы начать получать дополнительные оповещения спустя много часов после свершившегося.

Регулирование частоты отправки уведомлений в группе в Alertmanager осуществляется с помощью параметров `group_wait` и `group_interval`.

Если есть группа, в которую не отправлялось никаких оповещений, а затем возникают условия, приводящие к отправке нового набора оповещений, то вполне вероятно, что все эти новые оповещения будут посыпаться неодновременно. Например, поскольку операции опроса распределяются по интервалу опроса, в случае сбоя стойки вы обнаружите, что некоторые машины вышли из строя на один интервал раньше других. Было бы хорошо иметь возможность немного отложить отправку первого уведомления для группы, чтобы посмотреть, не появятся ли дополнительные оповещения. Именно такое поведение позволяет организовать параметр `group_wait`. По умолчанию Alertmanager ждет 30 с перед отправкой первого уведомления. Вам может показаться, что это задержит реакцию на инциденты, но имейте в виду, что если 30 с имеют значение, то такие проблемы должны решаться автоматически и без участия человека.

После отправки в группу первого уведомления могут начать появляться дополнительные оповещения. Когда Alertmanager должен отправить следующее уведомление, принадлежащее этой же группе, но включающее новые оповещения? Это время определяется параметром `group_interval`, который по умолчанию равен 5 мин. Через указанный интервал после отправки первого уведомления будет отправлено новое уведомление, если появятся новые оповещения, попадающие в эту же группу. Если новых оповещений не появится, то вы не получите дополнительных уведомлений.

Как только оповещения для группы перестанут появляться и пройдет определенное время, состояние сбрасывается, и снова применяется `group_wait`. Регулирование частоты отправки уведомлений для каждой группы осуществляется независимо, поэтому, если вы выполняете группировку по регионам, то оповещения, активируемые для одного региона, не заставят новые оповещения в другом регионе ждать `group_interval`, только `group_wait`.

Рассмотрим пример появления четырех оповещений в разное время:

```
t= 0 Alert firing {x="foo"}
t= 25 Alert firing {x="bar"}
t= 30 Notification for {x="foo"} and {x="bar"}
t=120 Alert firing {x="baz"}
t=330 Notification for {x="foo"}, {x="bar"} and {x="baz"}
t=400 Alert resolved {x="foo"}
t=700 Alert firing {x="quu"}
t=930 Notification for {x="bar"}, {x="baz"}, {x="quu"}
```

После получения первого оповещения начинается обратный отсчет `group_wait`, и до истечения этого срока поступает второе оповещение. Оба

этих оповещения, `foo` и `bar`, будут отправлены в уведомлении через 30 с после получения первого оповещения. В момент отправки уведомления запускается таймер `group_interval`. До его срабатывания появляется новое оповещение `baz`, поэтому через 300 с (один групповой интервал) после отправки первого уведомления посыпается второе уведомление, содержащее все три оповещения, активных в данный момент. В следующем интервале одно оповещение было устранено, но новых оповещений не поступало, поэтому в момент времени $t=630$ не посыпается никаких уведомлений. Затем поступает четвертое оповещение `qui`, и через очередной групповой интервал отправляется третье уведомление, содержащее все три оповещения, активных в данный момент.

-  Если оповещение появляется, устраниется и снова появляется в течение группового интервала, то оно обрабатывается, как если бы никогда не устранилось. Точно так же, если оповещение устраниется, появляется и снова устраниется в течение группового интервала, то это равносильно тому, как если бы оповещение никогда не появлялось в этом интервале. Это не то, о чем стоит беспокоиться на практике.

Ни люди, ни машины не могут быть абсолютно надежными; даже если уведомление дошло до дежурного, он может забыть о нем, если случится что-то более срочное. В системе регистрации инцидентов вы можете закрыть инцидент, отметив его как решенный, но вам потребуется снова открыть его, если оповещение продолжает возникать.

Для этого можно воспользоваться параметром `repeat_interval`, который по умолчанию равен 4 часам. Если с момента отправки уведомления с группой оповещений миновал интервал повтора `repeat_interval`, то будет отправлено новое уведомление. При этом отправка уведомления по истечении группового интервала `group_interval` сбросит таймер для интервала `repeat_interval`. Устанавливать значение `repeat_interval` меньше, чем `group_interval`, не имеет смысла.

-  Если уведомления посыпаются слишком часто, то лучше настроить `group_interval`, а не `repeat_interval`, потому что проблема, скорее всего, в том, что оповещения появляются снова до истечения (обычно довольно длительного) интервала `repeat_interval`.

Эти параметры имеют вполне разумные значения по умолчанию, но при необходимости их можно немного изменить. Например, даже сложная проблема, как правило, решается в течение 4 часов, поэтому, если оповещение все еще активно после такого длительного времени, то причина может быть в том, что человек забыл отключить оповещение или был занят решением более срочной задачи, и повторная оправка уведомления вряд ли будет воспринята как спам. Для системы регистрации инцидентов обычно достаточно создавать и отправлять инциденты один раз в день, поэтому параметры `group_interval` и `repeat_interval` можно установить равными одному дню. Alertmanager будет автоматически повторять неудачные попытки отправить уведомления, поэтому нет необходимости уменьшать `repeat_interval` только по этой причине. В зависимости от конкретных особенностей вы можете

увеличить `group_wait` и `group_interval`, чтобы уменьшить количество получаемых уведомлений.

Все эти настройки можно определить отдельно для каждого маршрута, и они будут наследоваться дочерними маршрутами как значения по умолчанию. Вот пример конфигурации с их использованием:

```
route:
  receiver: fallback-pager
  group_by: [team]
  routes:
    # Команда пользовательского интерфейса.
    - matchers:
        - team = frontend
      group_by: [region, env]
      group_interval: 10m
      receiver: frontend-pager
      routes:
        - matchers:
            - severity = page
          receiver: frontend-pager
          group_wait: 1m
        - matchers:
            - severity = ticket
          receiver: frontend-ticket
          group_by: [region, env, alertname]
          group_interval: 1d
          repeat_interval: 1d
```

Приемники

Приемники принимают сгруппированные оповещения и создают уведомления. Приемник содержит **уведомители**, фактически посылающие уведомления. Начиная с версии Alertmanager 0.24.0, поддерживаются уведомители, осуществляющие отправку уведомлений по электронной почте, в PagerDuty, Pushover, Slack, Opsgenie, VictorOps, WeChat, AWS SNS, Telegram и веб-обработчики. Подобно тому, как обнаружение служб на основе файлов считается универсальным механизмом обнаружения служб, веб-обработчики считаются универсальными уведомителями – они позволяют подключаться к системам, которые не поддерживаются по умолчанию.

Структура определения приемников аналогична определению механизма обнаружения служб в конфигурации опроса. Каждый приемник должен иметь уникальное имя и может содержать любое количество уведомителей. В простейшем случае приемник имеет одного уведомителя:

```
receivers:
  - name: fallback-pager
    pagerduty_configs:
      - service_key: XXXXXXXX
```

PagerDuty – один из самых простых в использовании уведомителей, поскольку ему требуется только служебный ключ. Всем уведомителям необходимо сообщить, куда отправлять уведомления. Это может быть имя канала в чате, адрес электронной почты и любые другие идентификаторы, используемые системой. Большинство уведомителей являются коммерческими предложениями «программное обеспечение как услуга» (Software as a Service, SaaS), поэтому вы должны будете использовать их пользовательский интерфейс и документацию, чтобы получить различные ключи, идентификаторы, URL и токены. Мы не будем пытаться дать здесь полные инструкции, потому что уведомители и пользовательские интерфейсы SaaS постоянно меняются.

В одном приемнике также может быть определено несколько уведомителей. Вот пример приемника `frontend-pager`, отправляющий уведомления в службу PagerDuty и в канал Slack¹:

```
receivers:
- name: frontend-pager
  pagerduty_configs:
    - service_key: XXXXXXXX
  slack_configs:
    - api_url: https://hooks.slack.com/services/XXXXXXXXXX
      channel: '#pages'
```

Некоторые уведомители имеют настройки, которые должны быть одинаковыми для всех вариантов их использования, например ключ VictorOps API. Такие настройки можно указать в каждом приемнике, но в Alertmanager есть специальный раздел глобальных параметров. Благодаря этому в случае с VictorOps, вам останется настроить в самом уведомителе лишь ключ маршрутизации:

```
global:
  victorops_api_key: XXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX

receivers:
- name: backend-pager
  victorops_configs:
    - routing_key: a_route_name
```

Поскольку каждое поле, такое как `victorops_configs`, – это список, можно настроить передачу уведомлений сразу нескольким уведомителям одного типа, например отправлять сообщения в несколько чатов Telegram²:

```
receivers:
- name: backend-pager
  opsgenie_configs:
```

¹ PagerDuty тоже имеет интеграцию со Slack, позволяющую пересыпать оповещения непосредственно из Slack. Такая интеграция весьма удобна и может использоваться для передачи уведомлений, поступающих в PagerDuty из других источников, отличных от Alertmanager.

² Этот способ предпочтительнее использования `continue`, потому что менее хрупкое и не требует поддерживать несколько маршрутов.

```

- teams: backendTeam # Элементы в списке перечисляются через запятую.

telegram_configs:
- bot_token: XXX
  chat_id: YYY
- bot_token: XXX
  chat_id: ZZZ

```

Также можно вообще не указывать приемники, и в этом случае никакие уведомления отправляться не будут:

```
receivers:
- name: null
```

Однако по возможности лучше вообще не отправлять оповещения в Alertmanager, чем тратить ресурсы Alertmanager на обработку оповещений, чтобы потом просто отбросить их.

Уведомитель в форме веб-обработчика унаследовал тем, что не уведомляет напрямую существующую систему обмена сообщениями, которая у вас уже может быть установлена, а отправляет всю имеющуюся информацию о группе оповещений в виде HTTP-сообщения в формате JSON и позволяет делать с ним все, что вы посчитаете нужным. Этот механизм можно использовать для журналирования оповещений, выполнения автоматических действий в той или иной форме или для отправки уведомления через систему, которую Alertmanager не поддерживает напрямую. Конечная точка HTTP, которая принимает HTTP-запрос POST от веб-обработчика, называется *приемником веб-обработчика*.



Довольно заманчиво широко использовать веб-обработчики для выполнения произвольного кода, но разумнее стараться минимизировать циклы управления. Например, вместо использования цепочки «экспортер–Prometheus–Alertmanager–веб-обработчик» для перезапуска зависшего процесса лучше организовать решение этой задачи в пределах одной машины с применением супервизора, такого как Supervisord или Монит. Это обеспечит более короткое время отклика и большую надежность из-за меньшего количества движущихся частей.

Уведомители на основе веб-обработчика похожи на другие уведомители; они принимают URL для отправки уведомления. Решив журналировать все оповещения, вы бы могли использовать continue в первом маршруте, ведущем к веб-обработчику:

```

route:
  receiver: fallback-pager
  routes:
    - receiver: log-alerts
      continue: true
      # Настройки остальных маршрутов.

receivers:
- name: log-alerts
  webhook_configs:
    - url: http://localhost:1234/log

```

Для приема таких уведомлений и обработки содержащихся в них оповещений можно использовать сценарий на Python 3, как показано в примере 19.1.

Пример 19.1 ❖ Простой приемник в виде веб-обработчика, написанный на Python 3

```
import json
from http.server import BaseHTTPRequestHandler
from http.server import HTTPServer

class LogHandler(BaseHTTPRequestHandler):
    def do_POST(self):
        self.send_response(200)
        self.end_headers()
        length = int(self.headers['Content-Length'])
        data = json.loads(self.rfile.read(length).decode('utf-8'))
        for alert in data["alerts"]:
            print(alert)

if __name__ == '__main__':
    httpd = HTTPServer(('', 1234), LogHandler)
    httpd.serve_forever()
```

Все HTTP-приемники имеют поле `http_config`, аналогичное конфигурации опроса, как описано в разделе «Как извлекать метрики» главы 8, которое позволяет определить `proxy_url`, параметры HTTP Basic Authentication, настройки TLS и вообще все, что связано с HTTP.

Шаблоны уведомлений

На начальном этапе вполне подойдут конкретные сообщения, посылаемые различными уведомителями, но с течением времени вам наверняка захочется их настроить. Все уведомители, кроме веб-обработчиков¹, поддерживают определение шаблонов с использованием той же системы шаблонов Go (<https://oreil.ly/pY91X>), что используется для поддержки правил оповещений, как описывается в разделе «Аннотации и шаблоны» главы 18. Однако доступные данные и функции немного отличаются, потому что обрабатывается группа, а не одно оповещение.

Например, в уведомлениях Slack практически всегда могут понадобиться метки `region` и `env`:

```
receivers:
- name: frontend-pager
  slack_configs:
    - api_url: https://hooks.slack.com/services/XXXXXXX
      channel: '#pages'
      title: 'Alerts in {{ .GroupLabels.region }} {{ .GroupLabels.env }}!'
```

¹ Приемники на основе веб-обработчиков специально создавались для отправки сообщений в формате JSON, поэтому для них не предусмотрена поддержка шаблонов. На самом деле сообщение в формате JSON – это точно такая же структура данных, которая используется для описания шаблонов уведомлений.

Этот шаблон создаст уведомление, подобное изображенному на рис. 19.1.



Рис. 19.1 ♦ Сообщение в Slack с информацией о регионе и окружении

`GroupLabels` – одно из полей верхнего уровня, доступных в шаблонах, но, кроме него, есть еще целый ряд полей.

`GroupLabels`

`GroupLabels` содержит групповые метки уведомления, т. е. все метки, перечисленные в параметре `group_by` маршрута, откуда пришла эта группа.

`CommonLabels`

`CommonLabels` – это все метки, общие для всех оповещений в уведомлении. В их число всегда входят все метки из `GroupLabels`, а также любые другие общие метки. Это поле удобно использовать для выявления сходств в оповещениях. Например, если оповещения группируются по региону и из строя вышла стойка машин, то оповещения для всех отключенных экземпляров могут иметь общую метку `gack`, доступную в `CommonLabels`. Однако если выйдет из строя хотя бы одна машина в другой стойке, то метка `gack` не будет включена в `CommonLabels`.

`CommonAnnotations`

Поле `CommonAnnotations` похоже на поле `CommonLabels`, но содержит общие аннотации. Оно имеет очень ограниченное использование. Обычно аннотации формируются из шаблонов, поэтому маловероятно, что будут иметься какие-то общие значения. Однако если в качестве аннотаций используются простые строки, то они вполне могут появиться здесь.

`ExternalURL`

Поле `ExternalURL` будет содержать *внешний URL* диспетчера Alertmanager, что может упростить доступ к Alertmanager для отключения оповещений. Его также можно использовать для выяснения, какой именно диспетчер Alertmanagers из имеющихся в кластере отправил уведомление. Дополнительные сведения о внешних URL вы найдете в разделе «Сети и аутентификация» главы 21.

`Status`

Поле `Status` будет иметь значение `firing`, если хотя бы одно оповещение в уведомлении активно; если все оповещения устраниены, то поле получит значение `resolved`. Об устраниенных уведомлениях рассказывается в разделе «Устраниенные уведомления» ниже.

`Receiver`

Имя приемника, такое как `frontend-pager` в предыдущем примере.

GroupKey

Строка с уникальным машиночитаемым идентификатором группы. Это поле бесполезно для людей, но помогает системам регистрации инцидентов и обмена сообщениями связывать уведомления из группы с предыдущими уведомлениями. Это может помочь предотвратить создание нового инцидента в вашей системе, если он уже был создан для той же группы.

Alerts

`Alerts` – это основная часть уведомления, список всех оповещений в нем.

Каждое оповещение в списке `Alerts` тоже имеет несколько полей.

Labels

Как нетрудно догадаться, это поле содержит метки оповещения.

Annotations

Имя этого поля говорит само за себя, не допуская никаких разнотечений.

Status

Имеет значение `firing`, если оповещение активно, иначе имеет значение `resolved`.

StartsAt

Это активизации оповещения в формате объекта `time.Time` в языке Go. Учитывая особенности работы Prometheus и протокола оповещения, это время не всегда совпадает с временем первого выполнения условия оповещения. На практике это поле не имеет большого значения.

EndsAt

Это время, когда оповещение перестает быть активным. Оно бесполезно для активных оповещений, но сообщает время, когда оповещение было устраниено.

GeneratorURL

Для оповещений из Prometheus¹ это ссылка на правило оповещения в веб-интерфейсе Prometheus, которую можно использовать для отладки. Но настоящая причина появления этого поля – создание в будущем функции в Alertmanager, которая позволит отбрасывать оповещения, поступающие из определенного источника, например из неправильно действующей системы Prometheus, которую нет возможности отключить, отправляющей недостоверные оповещения в Alertmanager.

Вы можете использовать эти поля в шаблонах, как вам заблагорассудится. Можно, к примеру, во все уведомления включить все метки, ссылку на вики и ссылку на дашборд:

```
receivers:
- name: frontend-pager
slack_configs:
```

¹ Для других систем это должна быть ссылка на механизм, генерирующий оповещение.

```

- api_url: https://hooks.slack.com/services/XXXXXXX
  channel: '#pages'
  title: 'Alerts in {{ .GroupLabels.region }} {{ .GroupLabels.env }}!'
  text: >
    {{ .Alerts | len }} alerts:
    {{ range .Alerts }}
      {{ range .Labels.SortedPairs }}{{ .Name }}={{ .Value }} {{ end }}
      {{ if eq .Annotations.wiki "" -}}
        Wiki: http://wiki.mycompany/{{ .Labels.alertname }}
      {{- else -}}
        Wiki: http://wiki.mycompany/{{ .Annotations.wiki }}
      {{- end }}
      {{ if ne .Annotations.dashboard "" -}}
        Dashboard: {{ .Annotations.dashboard }}&region={{ .Labels.region }}
      {{- end }}
    {{ end }}
  {{ end }}

```

Давайте рассмотрим этот пример подробнее:

```
{{ .Alerts | len }} alerts:
```

.Alerts – это список, а len – встроенная функция шаблонов Go, подсчитывающая количество оповещений в списке. Функция len – это практически единственная математическая операция, которую можно выполнить с шаблонами Go. Они не имеют математических операторов, поэтому для любых вычислений и форматирования используйте шаблоны оповещений Prometheus, как обсуждалось в разделе «Аннотации и шаблоны» главы 18:

```

{{ range .Alerts }}
{{ range .Labels.SortedPairs }}{{ .Name }}={{ .Value }} {{ end }}

```

Выполняет обход оповещений и сортировку меток в каждом них.

range в шаблонах Go использует оператор точки (.) как итератор, поэтому оригинальный оператор . становится недоступен внутри цикла¹. Вы можете перебирать пары ключ–значение меток обычным для Go способом, но при этом не гарантируется какой-то определенный порядок их следования. Метод SortedPairs позволяет отсортировать метки и аннотации по именам и затем выполнить обход списка в предсказуемом порядке:

```

{{ if eq .Annotations.wiki "" -}}
Wiki: http://wiki.mycompany/{{ .Labels.alertname }}
{{- else -}}
Wiki: http://wiki.mycompany/{{ .Annotations.wiki }}
{{- end }}

```

Метки с пустыми значениями интерпретируются как отсутствующие, поэтому здесь сначала проверяется наличие аннотации wiki. И если она существует, то ее значение используется как ссылка на вики-страницу; в против-

¹ Для решения этой проблемы вы можно установить переменную, например {{ \$dot := . }}, а затем обращаться к \$dot.

ном случае используется имя оповещения. Так можно определить разумное значение по умолчанию, избавить себя от необходимости добавлять аннотацию `wiki` в каждое отдельное правило оповещения и сохранить возможность переопределить его в одном или двух оповещениях. Операторные скобки `{{- и -}}` указывают, что пробельные символы до или после фигурных скобок должны игнорироваться. Это позволяет записывать шаблоны в нескольких строках для удобства чтения, не добавляя лишних пробелов в вывод:

```
{{ if ne .Annotations.dashboard "" -}}
Dashboard: {{ .Annotations.dashboard }}&region={{ .Labels.region }}
{{- end }}
```

Если присутствует аннотация `dashbord`, она будет добавлена в уведомление, а кроме того, регион будет добавлен в URL как параметр. Если у вас есть переменная шаблона Grafana с таким именем, то она получит правильное значение. Как обсуждалось в разделе «Внешние метки» главы 18, правила оповещения не имеют доступа к внешним меткам, которые обычно определяют такие характеристики, как регион, поэтому именно так можно добавлять архитектурные детали в свои уведомления и избавить правила оповещения от необходимости знать о том, как развертываются ваши приложения.

Конечным результатом этого шаблона является уведомление, изображенное на рис. 19.2. При рассылке уведомлений через системы мгновенных сообщений желательно сохранить их максимально короткими. Это снизит вероятность того, что экран вашего компьютера или мобильного телефона будет заполнен длинными подробностями, затрудняющими получение общего представления о происходящем. Уведомления, подобные этому, должны помогать начать отладку, ссылаясь на потенциально полезные дашборд и инструкцию, где можно найти дополнительную информацию, а не пытаться передать всю информацию, которая может пригодиться.

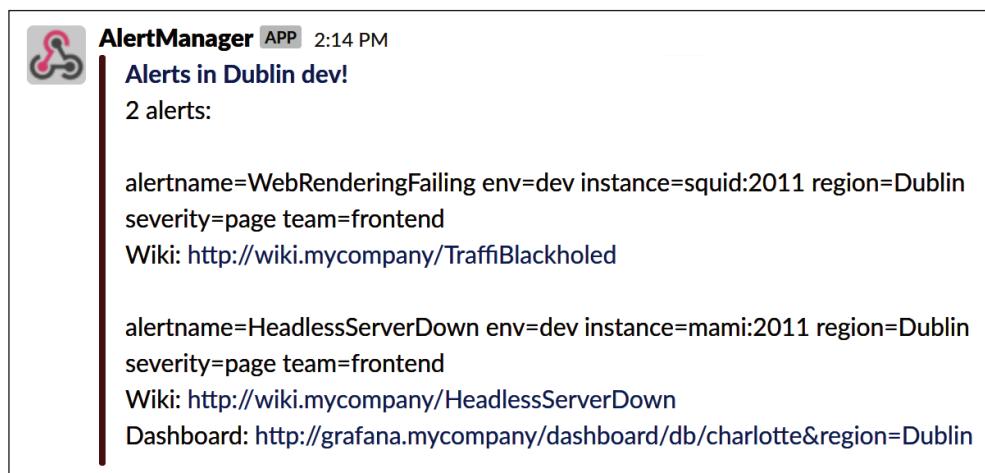


Рис. 19.2 ♦ Настроенное сообщение в Slack

В дополнение к шаблонам текстовых полей получатель уведомлений тоже может определяться шаблоном. Обычно каждая из команд разработчиков имеет свою часть дерева маршрутизации и свои приемники. Если другая команда пожелает организовать отправку оповещений вашей команде, то она может установить соответствующие метки, чтобы использовать дерево маршрутизации вашей команды. В случаях, когда вы предлагаете услугу внешним клиентам, определение приемника для каждого потенциального получателя может оказаться довольно утомительной задачей¹.

Сочетая возможности PromQL, меток и шаблонов уведомлений для определения получателей, можно даже определить пороговое значение для каждого клиента и адрес для передачи уведомлений в метрике, а Alertmanager доставит их по этому адресу. Первый шаг – создать оповещение, включающее адрес назначения в виде метки:

```
groups:
- name: example
  rules:
    - record: latency_too_high_threshold
      expr: 0.5
      labels:
        email_to: foo@example.com
        owner: foo
    - record: latency_too_high_threshold
      expr: 0.7
      labels:
        email_to: bar@example.com
        owner: bar
    - alert: LatencyTooHigh
      expr: |
        # Оповещение, основанное на пороговых значениях,
        # уникальных для каждого клиента.
        owner:latency:mean5m
        > on (owner) group_left(email_to)
          latency_too_high_threshold
```

Здесь по метрике, содержащей метку `email_to`, для разных клиентов определяются разные пороговые значения. Это прием хорошо подходит для внутренних клиентов, которые могут добавить свой параметр `latency_too_high_threshold` в ваш файл правил; пороговые значения для внешних клиентов можно определять с помощью экспортера, а адреса брать из базы данных.

Затем в Alertmanager по метке `email_to` можно назначить адрес для отправки уведомления:

```
global:
  smtp_smarthost: 'localhost:25'
  smtp_from: 'youraddress@example.org'
```

¹ Предполагается, что конфигурация Alertmanager будет меняться относительно редко, так как структура меток не должна меняться часто. С другой стороны, правила оповещения постоянно обновляются и корректируются.

```

route:
  group_by: [email_to, alertname] ①
  receiver: customer_email

receivers:
- name: customer_email
  email_configs:
    - to: '{{ .GroupLabels.email_to }}'
      headers:
        subject: 'Alert: {{ .GroupLabels.alertname }}'

```

- ① Параметр `group_by` должен включать метку `email_to`, определяющую адресата, потому что для каждого адресата должна иметься своя группа оповещений. Тот же подход можно использовать с другими уведомителями. Обратите внимание, что любой, у кого есть доступ к Prometheus или Alertmanager, сможет увидеть адреса назначения, поскольку метки доступны всем. Это может быть проблемой, если адреса назначения являются конфиденциальной информацией.

Устраниенные уведомления

Все уведомители имеют поле `send_resolved` с различными значениями по умолчанию. Если для него установлено значение `true`, то в уведомления, создаваемые при активации оповещений, будут включаться оповещения, которые были устраниены. На практике, когда Prometheus информирует Alertmanager о том, что оповещение устраниено¹, уведомитель с параметром `send_resolved`, имеющим значение `true`, включит это оповещение в следующее уведомление и даже отправит уведомление, состоящее только из устраниенных оповещений, если не появится никаких других активных оповещений.

На первый взгляд возможность узнать, что оповещения устраниено, кажется очень удобной, но мы советуем проявлять осторожность, поскольку устранение оповещения не означает, что проблема, вызвавшая его отправку, была исправлена. В разделе «Что такое хорошее оповещение?» главы 18 мы упомянули, что ответ на оповещение фразой «оно исчезло» может служить признаком, что такое оповещение, вероятно, вообще не должно было появиться. Получение уведомления с устраниенным оповещением может указывать на улучшение ситуации, но вам, как оперативному дежурному, все равно нужно разобраться в проблеме и убедиться, что она устранина и вряд ли вернется. Отказ от обработки инцидента из-за того, что оповещение перестало появляться, по сути, равнозначно заявлению «оно исчезло». Alertmanager работает с оповещениями, а не с инцидентами, поэтому неуместно считать инцидент решенным только потому, что оповещения прекратили поступать.

Например, устранившееся оповещение о сбое машины может означать только то, что машина, где работает Prometheus, тоже вышла из строя. То есть, несмотря на то что ситуация усугубляется, вы не получите оповещений об этом².

¹ Устраниенные оповещения будут иметь аннотации, полученные при последней их активации.

² Подходы к обнаружению подобных ситуаций описаны в разделе «Мета- и перекрестный мониторинг» главы 21, но в данном примере я хотел подчеркнуть, насколько важно расследовать оповещение сразу после его появления.

Еще одна проблема с устранимыми уведомлениями – они могут быть избыточными. Если включить их для уведомителя, посылающего уведомления по электронной почте или в Slack, то вы фактически удвоите объем сообщений и вдвое уменьшите соотношение сигнал/шум. Как обсуждалось во врезке «У каждого оповещения должен быть хозяин» в главе 18, рассылка уведомлений по электронной почте и без того не самый эффективный способ, и лишний шум не поможет привлечь внимание того, кто ее получает.

Если у вас есть уведомитель с параметром `send_resolved`, имеющим значение `true`, то в шаблонах уведомлений список `.Alerts` может содержать сочетание активных и устранимых оповещений. Конечно, вы можете сами фильтровать оповещения по полю `Status`, но есть более удобные решения: `.Alert.Firing` даст вам список только активных предупреждений, а `.Alert.Resolved` – только устранимых.

Подавление

Подавление позволяет деактивировать некоторые оповещения, если активируются другие оповещения. Например, если во всем центре обработки данных возникли проблемы, но пользовательский трафик был перенаправлен в другое место, то нет особого смысла отправлять оповещения для этого центра обработки данных.

В настоящее время подавление¹ настраивается на верхнем уровне в файле `alertmanager.yml`. В настройках нужно указать, какие оповещения искать, какие оповещения они будут подавлять и какие метки должны совпадать в них:

```
inhibit_rules:
  - source_matchers:
      - severity = page-regionfail
    target_matchers:
      - severity = page
      equal: ['region']
```

В данном случае, если активируется оповещение с меткой `severity`, имеющей значение `page-regionfail`, то подавляются все оповещения с тем же значением метки `region`, имеющие метку `severity` со значением `page`².

Старайтесь избегать дублирования между `source_match` и `target_match`, потому что их будет сложно понять и поддерживать. Наличие разных значений в метке `severity` – один из способов избежать дублирования. При дублировании любые оповещения, соответствующие `source_match`, не будут подавляться.

¹ В будущем, возможно, функцию подавления реализуют на уровне отдельных маршрутов (ее настройка на глобальном уровне увеличивает вероятность случайно подавить не то, что нужно).

² Использование `match_ge` в маршрутах упрощает анализ более конкретных значений метки `severity`, при этом все оповещения обрабатываются в одном маршруте. Если исходные оповещения не должны порождать, то для них предпочтительнее использовать пустой приемник, как показано в разделе «Приемники» выше.

Мы рекомендуем очень аккуратно использовать эту функцию. При оповещениях на основе симптомов (как обсуждалось в разделе «Что такое хорошее оповещение?» главы 18) оповещения не должны создавать цепочек зависимостей. Используйте правила подавления только для крупномасштабных проблем, таких как перебои в работе центров обработки данных.

Веб-интерфейс Alertmanager

Как было показано в разделе «Оповещение» главы 2, Alertmanager позволяет просматривать оповещения, активные в данный момент, а также группировать и фильтровать их. На рис. 19.3 показано несколько оповещений в Alertmanager, сгруппированных по alertname; здесь также можно увидеть все другие метки оповещений.

The screenshot shows the Alertmanager status interface. At the top, there's a navigation bar with links for Alertmanager, Alerts, Silences, Status, Help, and a 'New Silence' button. Below the navigation is a search bar with a placeholder 'Custom matcher, e.g. env="production"'. To the right of the search bar are buttons for 'Receiver: All', 'Silenced' (unchecked), and 'Inhibited' (unchecked). A 'Filter' tab is selected, and a 'Group' tab is available. Below the search bar is a section for the 'alertname="InstanceDown"' group, which contains two alerts. Each alert entry shows a timestamp (2022-11-16T09:14:59.499Z), a 'Source' link, and a 'Silence' link. Below the timestamp are filter buttons for 'env="prod"', 'instance="bar:9100"', 'job="node"', 'region="Dublin"', 'severity="ticket"', and 'team="frontend"'. The second alert entry for 'alertname="ManyInstancesDown"' shows a single alert with the same timestamp, source, silence, and filter buttons.

Рис. 19.3 ♦ Страница состояния Alertmanager с несколькими оповещениями

На странице состояния можно щелкнуть по кнопке **New Silence** (Новое подавление), чтобы определить правило подавления с нуля, или по кнопке

Silence (Подавить), чтобы получить форму, заполненную метками выбранного оповещения. В этой форме вы сможете настроить метки для правила подавления. При работе с существующим оповещением обычно достаточно удалить некоторые метки, чтобы охватить больше, чем одно оповещение. Для отслеживания правила подавления также следует ввести его имя и комментарий. В заключение можно выполнить предварительный просмотр правила, как показано на рис. 19.4, чтобы убедиться, что оно не слишком широкое.

New Silence

Start **Duration** **End**

2022-11-16T09:17:33.679Z 2h 2022-11-16T11:17:33.679Z

Matchers Alerts affected by this silence

alername="InstanceDown" env="prod" job="node" region="Dublin"
 severity="ticket" team="frontend"

Custom matcher, e.g. env="production"

Creator
Julien

Comment
Example comment

Affected alerts: 2

2022-11-16T09:14:59.499Z [Source](#)
 alername=InstanceDown env=prod instance=foo:9100 job=node region=Dublin severity=ticket
 team=frontend

2022-11-16T09:14:59.499Z [Source](#)
 alername=InstanceDown env=prod instance=bar:9100 job=node region=Dublin severity=ticket
 team=frontend

Рис. 19.4 ❖ Предварительный просмотр правила подавления перед вводом его в действие

Открыв страницу **Silences** (Правила подавления), можно увидеть все правила, активные в настоящее время, правила, которые еще не были применены, и правила, срок действия которых истек (см. рис. 19.5). Здесь также можно отменить подавление, которое больше не применяется, и заново активировать правило, срок действия которого истек.

The screenshot shows the 'Silences' section of the Alertmanager web interface. At the top, there are navigation links: Alertmanager, Alerts, Silences, Status, Help, and a 'New Silence' button. Below the header is a 'Filter' input field with a '+' button to add custom matchers, followed by the placeholder 'Custom matcher, e.g. env="production"'. Underneath the filter is a row of status buttons: 'Active' (selected, with a count of 1), 'Pending', and 'Expired'. To the right of these buttons are three buttons: 'View', 'Edit', and 'Expire'. Below these buttons is a list of active suppression rules, each represented by a small card. The first rule in the list has the following details: alertname=InstanceDown, env=prod, job=node, region=Dublin, severity=ticket, and team=frontend. The timestamp for this rule is Ends 2022-11-16T11:17:33.679Z.

Рис. 19.5 ❖ Страница **Silences** (Правила подавления) в веб-интерфейсе Alertmanager показывает список активных правил

Подавление не позволяет рассматривать оповещения с определенными метками как потенциальные уведомления. Правила подавления можно определить заранее, если известно, что в определенное время будет выполняться техническое обслуживание и не хотелось бы в этот период беспокоить дежурного оператора бессмысленными сообщениями. Дежурный оператор тоже может использовать функцию подавления для отключения оповещений, расследованием которых он уже занимается. Функцию подавления можно представить как кнопку повтора на будильнике. При создании правила подавления вы должны ввести комментарий, в котором указать причину подавления, чтобы она не была забыта или неправильно понята.

Если требуется останавливать рассылку оповещений на определенное время каждый день, то это желательно делать не с помощью правил подавления, а путем добавления условия в правила оповещения, проверяющего значение, которое возвращает функция `hour`, как показано в разделе «Правила оповещения» главы 18.

Теперь, когда вы познакомились со всеми ключевыми компонентами Prometheus, пришло время посмотреть, как они сочетаются друг с другом на более высоком уровне. В следующей главе вы узнаете, как развернуть Prometheus.

Часть VI

РАЗВЕРТЫВАНИЕ

Экспериментировать с Prometheus на собственной машине – это одно, а развернуть его в реальном промышленном окружении – совсем другое.

В главе 20 мы рассмотрим имеющиеся функции безопасности, доступные для защиты сервера Prometheus, а в главе 21 перечислим практические аспекты запуска Prometheus в промышленном окружении и способы его развертывания.

Глава 20

Безопасность на стороне сервера

В этой главе вы познакомитесь с функциями безопасности, имеющимися в Prometheus, такими как TLS и базовая аутентификация.

Функции безопасности в Prometheus

При работе с Prometheus многие операторы предпочитают использовать обратный прокси для защиты своих конечных точек. Как мы знаем, конечные точки API сервера Prometheus доступны через HTTP, что упрощает их интеграцию с любым обратным прокси, поддерживающим HTTP.

Сама система Prometheus имеет средства безопасности на стороне сервера, что позволяет предоставлять пользователям уже защищенную версию Prometheus либо защищать трафик между Prometheus и этими обратными прокси.

Настройки безопасности на стороне сервера, описываемые в этой главе, применимы к серверу Prometheus и большинству официальных экспортёров. Для этого могут использоваться одни и те же флаги и параметры командной строки, поэтому нижеследующее относится не только к Prometheus.

Параметры, описанные в этой главе, должны настраиваться в отдельном файле, путь к которому можно передать в `--web.config.file`. Содержимое этого файла читается автоматически при получении каждого запроса, а это означает, что нет необходимости перезапускать Prometheus или экспортёры после изменения этого файла.

Включение TLS

TLS широко используется для защиты сетевых соединений между клиентами и серверами. Выражаясь простыми словами, TLS позволяет клиенту

убедиться, что сервер, к которому он подключается, распознается известным центром сертификации (Certificate Authority, CA), а затем зашифровать последующий трафик. Также TLS можно использовать для аутентификации клиентов, заставляя их предоставлять действительный сертификат TLS при подключении к серверу.

Чтобы включить TLS на экземпляре Prometheus, для начала нужно получить сертификаты. В этом примере мы используем самозаверенные (или самоподписанные) сертификаты. Однако в реальных условиях следует использовать внутренний CA вашей компании или общедоступный CA, такой как Let's Encrypt, который будет использоваться вашими пользователями.

Сначала создадим самоподписанный сертификат с помощью OpenSSL:

```
$ openssl req -new -newkey rsa:2048 -days 365 -nodes -x509 \
-keyout prometheus.key -out prometheus.crt \
-subj "/CN=localhost" -addext "subjectAltName = DNS:localhost"
```

Эта команда создаст два файла: *prometheus.key* и *prometheus.crt*.

Чтобы включить TLS только со сгенерированными сертификатом и закрытым ключом, создадим файл *web.yml* с содержимым, показанным в примере 20.1.

Пример 20.1 ♦ web.yml

```
tls_server_config:
  cert_file: prometheus.crt
  key_file: prometheus.key
```



Проверить отсутствие ошибок в файлах конфигурации можно с помощью *rgomtool*:

```
$ ./rgomtool check web-config web.yml
web.yml SUCCESS
```

Теперь можно запустить сервер Prometheus и передать ему этот файл, как показано ниже:

```
$ ./prometheus --web.config.file web.yml
```

и получить доступ к нему с помощью TLS, используя следующую команду:

```
$ curl --cacert prometheus.crt https://127.0.0.1:9090/metrics
```

Поскольку сервер Prometheus обычно опрашивает сам себя, конфигурацию опроса в основном файле *prometheus.yml* тоже нужно адаптировать, как показано в примере 20.2.

Пример 20.2 ♦ prometheus.yml

```
scrape_configs:
  - job_name: 'prometheus'
    scheme: https
    tls_config:
      ca_file: prometheus.crt
```

```
static_configs:
- targets: ['localhost:9090']
```

Не забудьте перезагрузить конфигурацию Prometheus после внесения изменений в файл:

```
$ killall -HUP prometheus
```

Дополнительные параметры TLS

Конфигурация TLS в Prometheus поддерживает еще целый ряд настроек. В частности, можно настроить аутентификацию клиента, как показано в примере 20.3.

Пример 20.3 ◊ web.yml

```
tls_server_config:
  client_auth_type: RequireAndVerifyClientCert
  client_ca_file: client_ca.crt
```

Другие доступные настройки включают:

- `min_version` и `max_version` – описывают минимальную и максимальную версию TLS, поддерживаемую с сервером. Названия версий TLS10, TLS11, TLS12 и TLS13 соответствуют версиям TLS 1.0, 1.1, 1.2 и 1.3;
- `cipher_suites` – описывает набор шифров, используемый сервером. Этот параметр не действует в версии TLS 1.3;
- `prefer_cipher_suites` – определяет, какой предпочтительный набор шифров должен выбирать сервер: клиента или сервера;
- `curve_preferences` – перечисляет эллиптические кривые, которые будут использоваться в процедуре ECDHE¹ инициализации соединения, в порядке предпочтения.

Эти конфигурационные параметры позволяют полностью контролировать поведение базовой библиотеки TLS.

! Prometheus уже имеет безопасные настройки по умолчанию. Неразумно изменять их, если вы не знаете, что делаете, потому что можно непреднамеренно поставить под угрозу безопасность сервера.

Включение базовой аутентификации

Базовая аутентификация (Basic Authentication) требует аутентификации каждого запроса к серверу Prometheus с помощью имени пользователя и пароля. Для этого серверу Prometheus предоставляется список пользователей

¹ Elliptic Curve Diffie-Hellman Ephemeral (эллиптическая кривая Диффи–Хеллмана).

и хешированных паролей, который затем проверяет каждый входящий запрос по этому списку.

Расширенные механизмы авторизации, например ограничивающие доступ пользователей к страницам или получающие списки пользователей из других источников, таких как OAuth или LDAP, не поддерживаются Prometheus. Если вам потребуются такие тонкие настройки, то поставьте перед Prometheus обратный прокси. С помощью средств TLS и базовой аутентификации можно заставить обратный прокси аутентифицировать себя в Prometheus и тем самым защитить сервер и реализовать соответствующее управление доступом на прокси-сервере.

В конфигурации Prometheus пароли не хранятся в открытом виде. Они хешируются, а это значит, что если кто-то получит доступ к конфигурационному файлу, то он не сможет узнать пароль. Для хеширования паролей в Prometheus применяется bcrypt.

! Механизм базовой аутентификации передает пароли в открытом виде в заголовках HTTP. Чтобы предотвратить перехват паролей, мы настоятельно рекомендуем использовать TLS для шифрования трафика между клиентом и сервером.

Чтобы добавить пользователя и включить обычную аутентификацию, сначала нужно создать хеш его пароля. Воспользуемся для этого утилитой htpasswd, но то же самое можно проделать с другими инструментами:

```
$ htpasswd -nBC 10 "" | tr -d '\n'  
New password:  
Re-type new password:  
$2y$10$LbwE60VsPc4PqDFaYwvw/u0kMMficVQrQjtY5KT/BGnAKPa0vK45C
```

В этом примере я выбрал пароль *demo*.

i 10 – это уровень сложности шифрования bcrypt. Обычно рекомендуется использовать сложность от 10 до 12 при текущей вычислительной мощности. Увеличение этого числа может повысить безопасность пароля, но за счет повышенного потребления вычислительных ресурсов.

Теперь этот пароль можно указать в файле *web.yml*, как показано в примере 20.4.

Пример 20.4 ♦ web.yml

```
tls_server_config:  
  cert_file: prometheus.crt  
  key_file: prometheus.key  
basic_auth_users:  
  julien: $2y$10$LbwE60VsPc4PqDFaYwvw/u0kMMficVQrQjtY5KT/BGnAKPa0vK45C
```

Мы настроили пользователя *julien* с паролем *demo* на сервере Prometheus, защищенном сертификатом TLS. Чтобы использовать эти учетные данные, откройте в браузере страницу <http://127.0.0.1:9090>. В ответ должен появиться диалог, предлагающий ввести имя пользователя и пароль. Введите

julien и **demo**, после чего должен открыться веб-интерфейс вашего сервера Prometheus.

Сам Prometheus тоже нужно настроить на опрос самого себя с использованием базовой аутентификации, как показано в примере 20.5.

Пример 20.5 ✦ prometheus.yml

```
scrape_configs:
  - job_name: 'prometheus'
    scheme: https
    tls_config:
      ca_file: prometheus.crt
    basic_auth:
      username: julien
      password: demo
    - targets: ['localhost:9090']
```

Аналогично можно передать имя пользователя с помощью cURL:

```
$ curl --cacert prometheus.crt -u julien:demo https://127.0.0.1:9090/metrics
```

Теперь, узнав, как защитить сервер Prometheus, в следующей главе мы рассмотрим порядок его развертывания в промышленном окружении.

Глава 21

Собираем все вместе

В предыдущих главах вы познакомились со всеми компонентами системы Prometheus: инструментированием, дашбордами, обнаружением служб, экспортерами, PromQL, оповещениями и диспетчером уведомлений Alertmanager. В этой последней главе вы узнаете, как собрать все это вместе, спланировать развертывание Prometheus и поддерживать его в будущем.

Планирование развертывания

При знакомстве с новой технологией лучше всего начать с развертывания¹ чего-то небольшого, что не потребует слишком много усилий, и не брать на себя преждевременное обязательство по полному внедрению. Приступающим к внедрению Prometheus в существующую систему мы рекомендуем сначала запустить Node Exporter² и Prometheus, как было показано в главе 2.

Node Exporter поддерживает все метрики машинного уровня, которые могут извлекаться из других систем мониторинга, как рассказывалось в главе 7. На этом этапе вы без особых усилий получите достаточно широкий набор метрик, чтобы освоиться с Prometheus, настроить некоторые дашборды и, возможно, даже создать некоторые оповещения.

Затем посмотрите, какие сторонние системы у вас используются, какие экспортёры для них существуют, и начните их развертывание. Например, если у вас есть сетевые устройства, то можно установить SNMP Exporter; если есть приложения на основе JVM, такие как Kafka или Cassandra, то можно задействовать JMX Exporter; а если вам нужен мониторинг в стиле «черного ящика», то можно использовать Blackbox Exporter, как описано в главе 10. Цель данного этапа – получить метрики о как можно большем количестве различных частей вашей системы с минимальными усилиями.

К настоящему моменту вы должны в достаточной степени знать Prometheus, чтобы выбрать свой подход к таким аспектам, как обнаружение служб,

¹ Развертывание – это процесс выпуска новой версии приложения или системы для пользователей.

² Работающие в Windows должны использовать Windows Exporter вместо Node Exporter.

обсуждавшимся в главе 8. Все предыдущие этапы вы наверняка сможете выполнить в одиночку. Следующий шаг – инструментирование своих приложений, как описано в главе 3. На этом этапе, вероятно, потребуется привлечь других людей и попросить их найти время для организации мониторинга. Возможность продемонстрировать все настроенные вами системы мониторинга и дашборды¹ (использующие информацию, собранную экспортерами) облегчит агитацию будущих сторонников Prometheus; без этого ваше желание организовать широкое инструментирование всего вашего кода вряд ли получит поддержку.

Как обычно, приступая к инструментированию, начните с метрик, которые принесут вам наибольшую выгоду. Ищите в своих приложениях узкие места, через которые проходит значительная часть трафика. Например, если у вас есть общие библиотеки HTTP, которые используются всеми вашими приложениями для связи друг с другом, то, снабдив их базовыми метриками RED, как описано в разделе «Инструментирование служб» главы 3, вы получите ключевые метрики производительности для больших сегментов ваших систем онлайн-обслуживания.

При наличии инструментированных метрик для другой системы мониторинга можно развернуть такие средства интеграции, как StatsD и Graphite Exporters, описанные в главе 11, чтобы воспользоваться тем, что уже есть. Но со временем следует не только полностью перейти на использование средств инструментирования Prometheus, как описано в главе 3, но также добавить дополнительные инструментированные метрики в свои приложения.

По мере расширения внедрения системы Prometheus для мониторинга своей инфраструктуры вы должны начать отказываться от других систем мониторинга. Нередко со временем в компании оказывается более 10 различных систем мониторинга, поэтому консолидация там, где это целесообразно, всегда выгодна.

Этот план является общим руководством, которое может и должно адаптировать к конкретным обстоятельствам. Например, если вы разработчик, то можете сразу начать инструментировать свои приложения. Можно даже добавить клиентскую библиотеку в свое приложение, чтобы внедрить уже готовые метрики, такие как потребление процессорного времени и сборка мусора.

Расширение Prometheus

Обычно внедрение Prometheus начинается с установки одного сервера в каждый центр обработки данных. Prometheus предназначен для работы в той же сети, мониторинг которой он осуществляет, потому что это снижает количество возможных сбоев и ограничивает их области, а также обеспечивает

¹ Мы по-прежнему поражаемся привлекательности красивых дашбордов, особенно по сравнению с другими факторами, такими как полезность метрик, отображаемых в них. Не стоит недооценивать это, пытаясь убедить других использовать Prometheus.

низкую задержку и высокую пропускную способность при доступе к целям, которые опрашивает Prometheus¹.

Сервер Prometheus действует достаточно эффективно, поэтому в большинстве случаев одного сервера Prometheus вполне достаточно для мониторинга всего центра обработки данных. Однако в какой-то момент операционные издержки, недостаточная производительность или просто социальные соображения заставят вас разграничить области мониторинга и разнести их по отдельным серверам Prometheus. Например, для мониторинга сети, инфраструктуры и приложений часто используются отдельные серверы Prometheus. Этот подход известен как *вертикальное сегментирование* и считается лучшим способом масштабирования Prometheus.

В долгосрочной перспективе можно потребовать, чтобы каждая команда запускала свои серверы Prometheus, что позволит им выбирать целевые метки и интервалы опроса, имеющие для них смысл (как обсуждалось в главе 8). Также можно запускать общие серверы для команд, но тогда вы должны быть готовы к тому, что команды будут слишком увлечены метками.

Брайан много раз наблюдал такую закономерность: сначала приходится долго убеждать команды в необходимости инструментировать их код или развернуть экспортеры, но потом, как по щелчку, они начинают осознавать силу меток. В результате через достаточно короткое время вы замечаете, что у вашего сервера Prometheus возникают проблемы с производительностью из-за кардинальности метрик, намного превышающей разумные пределы (как описано в разделе «Кардинальность» главы 5). Если вы используете Prometheus как общую службу, а команда, использующая эти метрики, не относится к числу получающих уведомления, то вам придется постараться убедить их в необходимости сократить кардинальность. Но если они запустят свой собственный сервер Prometheus и будут просыпаться в 3 часа ночи, то, вероятно, на собственном опыте уяснят различие между мониторингом на основе метрик и мониторингом на основе журналов.

Если у вашей команды имеются крупные системы, то вам может понадобиться развернуть по нескольку серверов Prometheus в каждом центре обработки данных. Команда инфраструктуры может иметь один Prometheus для экспортёров узлов, один для обратных прокси-серверов и один для всего остального. Для простоты управления серверы Prometheus можно запускать внутри каждого из имеющихся кластеров Kubernetes, а не пытаться контролировать их извне.

С чего вы начнете и чем закончите, будет зависеть от масштаба и культуры в вашей организации. Как показывает наш опыт, социальные факторы² часто

¹ Мониторинг через границы областей сбоев, например между центрами обработки данных, тоже возможен, но это довольно сложно, потому что на мониторинг начинает влиять множество дополнительных факторов, связанных с сетью. Если у вас сотни крошечных центров обработки данных с несколькими машинами в каждом, то установка одного сервера Prometheus на регион/континент может быть приемлемым компромиссом.

² Например, разумно определить в Prometheus только одну иерархию целевых меток. Если у какой-то команды сложится свое представление о том, что такое регион, отличное от представления в других командах, то ей следует запустить собственный сервер Prometheus.

приводят к тому, что серверы Prometheus разделяются до того, как возникают какие-либо проблемы с производительностью.

Выход на глобальный уровень с помощью механизма федерации

Как реализовать глобальное агрегирование при наличии отдельного сервера Prometheus в каждом центре обработки данных (ЦОД)?

Надежность – ключевое свойство хорошей системы мониторинга и главная ценность Prometheus. Когда дело доходит до отображения графиков и рассылки оповещений, желательно, чтобы имелось как можно меньше движущихся частей, потому что чем проще система, тем она надежнее. Чтобы построить график изменения задержек в приложении в центре обработки данных, можно связать Grafana с сервером Prometheus в этом центре обработки данных, который опрашивает это приложение. Аналогично можно реализовать оповещение о задержке приложения в каждом отдельном центре обработки данных.

Но такой подход не годится для глобальной задержки, потому что каждый из серверов Prometheus имеет только часть данных. Решить задачу в таких случаях вам поможет *федерация*. Федерация позволяет запустить глобальный сервер Prometheus, извлекающий агрегированные метрики из серверов Prometheus в центрах обработки данных, как показано на рис. 21.1.

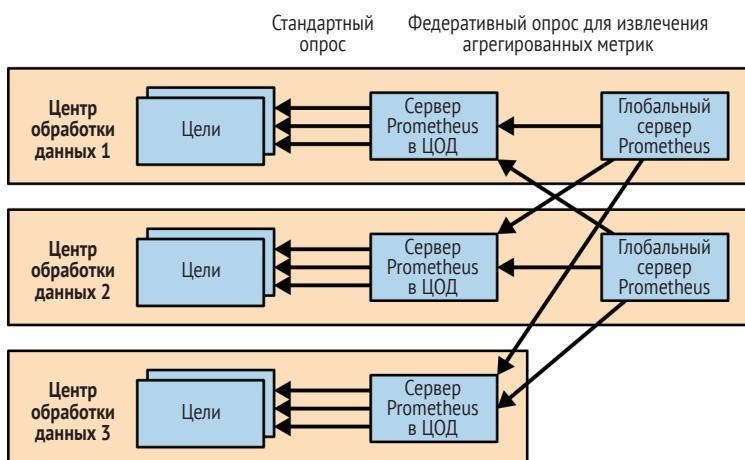


Рис. 21.1 ♦ Архитектура глобальной федерации

Например, для получения всех метрик, агрегированных на уровне задания, можно определить такие настройки в файле *prometheus.yml*:

```
scrape_configs:
- job_name: 'federate'
```

```
honor_labels: true
metrics_path: '/federate'
params:
'match[]':
- '{__name__=~"job:.*"}'
static_configs:
- targets:
- 'prometheus-dublin:9090'
- 'prometheus-berlin:9090'
- 'prometheus-new-york:9090'
```

Конечная точка */federate* сервера Prometheus принимает список селекторов (описываются в разделе «Селекторы» главы 13) в параметрах URL *match[]* и возвращает все соответствующие временные ряды, согласно семантике селекторов мгновенных векторов, включая устаревшие, как описано в разделе «Мгновенный вектор» главы 13. Если передать несколько параметров *match[]*, то будут возвращены все образцы, соответствующие любому из них. Чтобы избежать агрегированных метрик, имеющих целевую метку *instance*, здесь используется параметр *honor_labels* (обсуждавшийся в разделе «Конфликты меток и *honor_labels*» главы 8)¹. В федеративные метрики также добавляются внешние метки Prometheus (обсуждаются в разделе «Внешние метки» главы 18), благодаря которым можно определить, откуда получен каждый временной ряд.

- !** К сожалению, некоторые пользователи используют федерацию не только для сбора агрегированных метрик. Чтобы не попасть в эту ловушку, важно понимать следующее:
- механизм федерации не предназначен для копирования всего содержимого сервера Prometheus;
 - механизм федерации не предназначен для использования одного сервера Prometheus в роли прокси для другого сервера Prometheus;
 - механизм федерации не предназначен для получения метрик с меткой *instance*.

Позвольте нам пояснить, почему федерацию следует использовать только в рамках ее предназначения. Во-первых, для надежности важно иметь как можно меньше движущихся частей. Передача всех метрик через интернет в глобальный сервер Prometheus, откуда вы затем сможете брать информацию для построения графиков и рассыпать оповещения, будет означать жесткую зависимость всей системы мониторинга от надежности интернет-соединений с другими центрами обработки данных. В общем случае желательно ограничить области отказа, чтобы графики и оповещения для конкретного центра обработки данных не зависели от работы другого центра обработки данных. То есть отправлять оповещения, касающиеся определенного круга целей, должен тот же сервер Prometheus, который опрашивает эти цели. Это особенно важно, если учесть возможность сбоя сети.

Вторая проблема – масштабирование. Для надежности сервер Prometheus разрабатывался для работы в автономном режиме и на единственной маши-

¹ Конечная точка */federate* автоматически включает пустую метку *instance* во все выходные метрики, не имеющие ее, как это делает Pushgateway (см. раздел «Pushgateway» главы 4).

не, поэтому он ограничен возможностями машины. Prometheus достаточно эффективен, поэтому, даже будучи ограниченным возможностями одной машины, в большинстве случаев он вполне в состоянии контролировать весь центр обработки данных. По мере развертывания новых центров обработки данных вам просто нужно добавить в каждый из них свой сервер Prometheus. Глобальный сервер Prometheus, использующий только агрегированные метрики, помогает значительно уменьшить количество данных, с которыми приходится работать, по сравнению с серверами Prometheus в центрах обработки данных¹, и тем самым избежать создания узких мест. И наоборот, если бы глобальный сервер Prometheus собирал все метрики из серверов Prometheus всех центров обработки данных, то он стал бы узким местом и сильно ограничил ваши возможности масштабирования. Иными словами, для масштабирования федерации необходимо использовать тот же подход, который обсуждался в разделе «Уменьшение кардинальности» главы 17.

В-третьих, Prometheus предназначен для опроса многих тысяч целей малого и среднего размера². Распределяя опрос целей по настроенному интервалу, Prometheus может справляться с большими объемами данных при равномерной нагрузке. Если, напротив, заставить его опрашивать несколько целей с огромным количеством временных рядов, таких как массивные конечные точки механизма федерации, то это может вызвать скачки нагрузки, и сервер может даже не успеть завершить обработку одного массивного количества данных за время, выделенное для одного цикла опроса.

Четвертая проблема – семантика. Передавая все данные через дополнительный сервер Prometheus, вы ненамеренно создадите дополнительные условия гонки. На ваших графиках могут появиться дополнительные артефакты, и вы лишитесь преимуществ обработки семантики устаревания.

Один из контраргументов в пользу этой архитектуры: если все метрики не будут собраны в одно место, то как узнать, какой сервер Prometheus содержит некоторую искомую метрику? Как оказывается, такой проблемы не существует на практике. Если все серверы Prometheus будут следовать общей архитектуре, то обычно вы с легкостью сможете сказать, какие цели опрашивает каждый из них и, следовательно, без труда определите, где находится конкретная метрика. Например, метрики Node Exporter для цели Dublin будут находиться на сервере Prometheus в инфраструктуре Dublin. Grafana поддерживает как шаблоны источников данных, так и графики с метриками из разных источников данных, так что это не проблема и для дашбордов.

Обычно федерации имеют всего два уровня: нижний уровень с серверами Prometheus в центрах обработки данных и уровень с глобальными серверами. Глобальные серверы Prometheus обрабатывают запросы PromQL, которые

¹ Допустим, вы агрегировали каждую метрику из приложения с сотней экземпляров, и глобальный сервер Prometheus извлекает эти агрегированные метрики. Используя те же ресурсы, которые имеются в распоряжении сервера Prometheus в центре обработки данных, глобальный Prometheus сможет собрать метрики из сотен центров обработки данных. На самом деле глобальный Prometheus может обрабатывать гораздо больше, так как не все метрики будут агрегированы.

² Точных цифр нет, но по нашему опыту нехватка ресурсов начинает сказываться где-то в районе 10 000 временных рядов.

нельзя обработать на серверах нижнего уровня, например определяющие глобальный объем трафика.

Иногда также добавляется третий, дополнительный уровень. Например, обычным делом считается запускать сервер Prometheus внутри каждого кластера Kubernetes. Если в центрах обработки данных у вас организовано несколько кластеров Kubernetes, то вы можете объединить их агрегированные метрики в серверах Prometheus центров обработки данных, а затем в глобальном сервере Prometheus.

Еще одно применение механизма федерации – извлечение ограниченных агрегированных метрик из серверов Prometheus другой команды. При этом нелишним будет сначала спросить разрешения. А если это может перерастти в обычную практику, то желательно применить соображения, изложенные в разделе «Правила для API» главы 17. Однако нет необходимости делать это, если метрики нужны только для отображения в дашбордах, поскольку Grafana поддерживает использование нескольких источников данных в дашбордах и панелях.

Долгосрочное хранение

В разделе «Что такое мониторинг?» главы 1 мы упомянули, что мониторинг – это оповещение, отладка, анализ тенденций и решение практических задач. Для большинства случаев оповещения, отладки и решения практических задач вполне достаточно данных, накопленных за несколько последних дней или недель¹. Но когда дело доходит до анализа тенденций, например планирования емкости, то обычно нужны данные за годы.

Один из подходов к долговременному хранению – рассматривать Prometheus как традиционную базу данных и регулярно создавать резервные копии на случай сбоя. Prometheus, принимающий 10 000 образцов в секунду с консервативными 2 байтами на образец, использует чуть меньше 600 Гбайт дискового пространства в год, что не так много для современной машины.

Резервные копии можно создавать, отправляя HTTP-запросы POST конечной точке `/api/v1/admin/tsdb/snapshot`, которая вернет имя моментального снимка, созданного в каталоге хранилища Prometheus. При создании резервной копии используются жесткие ссылки, поэтому она не потребует много дополнительного дискового пространства, поскольку на диске хранится только одна фактическая копия данных. После копирования моментального снимка его лучше удалить, чтобы не занимать на диске больше места, чем необходимо. Чтобы восстановить базу данных из моментального снимка, достаточно скопировать его в каталог хранилища Prometheus.

Для анализа тенденций в долгосрочной перспективе интерес будет представлять лишь очень малая часть метрик. В основном это будут агрегирован-

¹ Брайан слышал от многих, что до 90 % из востребованных метрик в системах мониторинга получены в последние 24 часа. Проблема в том, чтобы заранее узнать, какие 90 % вам больше не понадобятся по прошествии суток.

ные метрики. Как правило, не все данные заслуживают, чтобы их хранили вечно, поэтому вы можете сэкономить много места, сохраняя только метрики из глобального сервера Prometheus или удаляя неаггрегированные метрики до определенного времени¹.

Конечная точка `/api/v1/admin/tsdb/delete` принимает селекторы в своем параметре URL `match[]`² и параметры `start` и `end` для ограничения временного диапазона. Данные, попавшие в этот диапазон, будут удалены с диска при следующем скатии. Было бы разумно удалять старые данные, скажем, раз в месяц.

Из соображений безопасности для включения конечных точек `snapshot` и `delete` необходимо передать в Prometheus флаг `--web.enableadmin-api`.

Другой подход – отправка образцов из Prometheus в какую-либо кластерную систему хранения, которая может использовать ресурсы многих машин. Удаленная запись отправляет образцы по мере их поступления в другую систему. Удаленное чтение позволяет PromQL прозрачно использовать образцы из другой системы, как если бы они хранились локально в Prometheus. Оба механизма настраиваются на верхнем уровне в `prometheus.yml`:

```
remote_write:
  - url: http://localhost:1234/write
remote_read:
  - url: http://localhost:1234/read
```

Удаленная запись поддерживает изменение меток через конфигурацию `write_relabel_configs`, действующую подобно описанной в разделе «`metric_relabel_configs`» главы 8. В основном она используется для выбора метрик, отправляемых в *удаленную конечную точку записи*, так как есть вероятность столкнуться с ограничениями по стоимости. При использовании удаленного чтения следует проявлять осторожность и не извлекать большое количество временных рядов за длительные периоды времени, так как можно столкнуться с ограничениями пропускной способности. При использовании удаленной записи важно, чтобы каждый сервер Prometheus имел уникальные внешние метки для предотвращения конфликтов метрик.

При использовании удаленного чтения и записи серверы Prometheus можно считать времененным кешем, а удаленное хранилище – основным хранилищем³. Если сервер Prometheus перезапускается с пустым хранилищем данных, то для построения графиков изменения метрик в прошлом будет использоваться удаленное чтение. Вам также придется разработать оповещения так, чтобы они были устойчивыми к такому перезапуску, что в любом случае является хорошей идеей.

В настоящее время существует много проектов, интегрирующихся с Prometheus Remote Write. Из проектов с открытым исходным кодом стоит упо-

¹ Как обсуждалось в разделе «Выход на глобальный уровень с помощью механизма федерации» выше, глобальный сервер Prometheus будет получать только агрегированные метрики.

² По аналогии с параметром URL `match[]` для федерации.

³ Обычно многонедельный кеш.

мянуть: CNCF Thanos, CNCF Cortex и Grafana Mimir, которые используют библиотеку с кодом из Prometheus и S3-совместимое хранилище. Все эти системы являются распределенными, многопользовательскими и пользуются популярностью в сообществе Prometheus.

 Оценивая варианты, учитывайте, что нагрузка, считающаяся небольшой для одного сервера Prometheus, может оказаться тяжелой для другой системы, действующей на нескольких машинах. Всегда разумно проводить нагружочное тестирование систем на основе вашего варианта использования, а не полагаться на общие показатели, представленные производителем, потому что разные системы разрабатываются с учетом разных моделей данных и шаблонов доступа. Более простые решения могут оказаться более эффективными и простыми в эксплуатации. Наличие слова «кластерный» в названии не означает автоматически лучший.

Обычно кластерные системы хранения требуют в пять раз больше ресурсов, чем эквивалентный сервер Prometheus при той же нагрузке. Это связано с тем, что большинство систем реплицируют данные три раза, а также принимают и обрабатывают все данные. Поэтому нужно очень тщательно выбирать, какие метрики будут храниться локально, а какие в кластерном хранилище.

 В Prometheus есть экспериментальный режим агента, который можно включить с помощью флага `--enable-feature=agent`. При использовании этого режима Prometheus выбирает оптимизированные алгоритмы для сценариев удаленной записи, и данные сохраняются локально только до момента их передачи в удаленное хранилище. Имейте в виду, что при работе в режиме агента нет возможности применять локальные правила записи или запрашивать данные. В этом режиме Prometheus использует меньше ресурсов по сравнению с режимом сервера, что делает это решение более легковесным и пригодным для определенных случаев.

Эксплуатация Prometheus

Когда дело доходит до фактического запуска сервера Prometheus, необходимо учитывать возможности аппаратного обеспечения, механизмы управления конфигурацией и особенности настройки сети.

Аппаратное обеспечение

Первый вопрос, который обычно возникает, когда дело доходит до запуска Prometheus, – какое аппаратное обеспечение необходимо? Лучше всего запускать Prometheus на машинах с твердотельными накопителями, хотя они не являются строго необходимыми для небольших конфигураций. Пространство для хранения – один из основных ресурсов, о котором нужно позаботиться. Чтобы оценить объем необходимого пространства, нужно определить, сколько данных вы будете принимать. В текущей версии Pro-

metheus¹ это можно узнать с помощью запроса PromQL, который сообщает количество образцов, загружаемых в секунду:

```
rate(prometheus_tsdb_head_samples_appended_total[5m])
```

В процессе работы Prometheus может достичь степени сжатия данных до 1.3 байта на образец, но при оценке мы используем более консервативную величину – 2 байта на образец. По умолчанию Prometheus хранит данные 15 дней, поэтому для хранения данных при скорости их поступления 100 000 образцов в секунду потребуется около 240 Гбайт. Срок хранения можно увеличить с помощью флага `--storage.tsdb.retention.time`, а путь к хранилищу можно указать с помощью флага `--storage.tsdb.path`. Также можно ограничить размер TSDB (хранилища временных последовательностей) с помощью флага `--storage.tsdb.retention.size`². Prometheus не рекомендует и не требует никакой конкретной файловой системы, и многие пользователи успешно используют сетевые блочные устройства, такие как Amazon EBS. Однако Prometheus не поддерживает NFS, в том числе Amazon EFS, потому что ожидает получить POSIX-совместимую файловую систему, а реализации NFS никогда не славились точностью поддержки семантики POSIX. Каждому серверу Prometheus требуется выделить свой каталог хранения; нельзя создать в сети единый каталог и использовать его совместно несколькими серверами.

Следующий вопрос – необходимый объем ОЗУ. Механизм хранения в Prometheus 2.x записывает данные блоками каждые два часа, которые впоследствии уплотняются в более крупные временные диапазоны. Он не осуществляет внутреннего кеширования и использует кеш страниц в ядре. По этой причине вам потребуется объем оперативной памяти, достаточный для хранения блока, плюс накладные расходы, плюс оперативная память, используемая во время запросов. Хорошей отправной точкой является 12-часовой интервал приема образцов, поэтому при скорости поступления данных 100 000 образцов в секунду необходимый объем памяти составит 8 Гбайт.

Сервер Prometheus нетребователен к процессору. Простой и приблизительный тест на нашей машине (с процессором i7-3770k) показывает, что только для обработки 100 000 образцов в секунду используется 0.25 от общего объема процессорного времени. Но это только для приема данных – дополнительное процессорное время потребуется также для обработки правил запросов и записи. Из-за всплесков потребления процессора сборщиком мусора в Go у вас всегда должно быть как минимум на одно ядро больше, чем показывают ваши расчеты.

Пропускная способность сети – еще одно немаловажное соображение. Prometheus 2.x может обрабатывать миллионы образцов в секунду, что сравнимо

¹ Для Prometheus 1.x используйте метрику `prometheus_local_storage_ingested_samples_total`.

² Это значение может не учитывать уплотняемые блоки, поэтому лучше не устанавливать его равным полной емкости физического хранилища. Кроме того, ограничивая размер хранилища, вы не сможете точно сказать, как долго будут храниться данные. В этом случае может оказаться полезным создать правило оповещения на основе метрики `prometheus_tsdb_lowest_timestamp`.

с ограничением одной машины во многих других подобных системах. Обычно при опросе Prometheus использует сжатие, поэтому для передачи образца данных требуется примерно 20 байт сетевого трафика. Соответственно, для передачи миллиона образцов в секунду нужна сеть с пропускной способностью 160 Мбит/с. Это весомая такая доля возможностей гигабитной сетевой карты и общей пропускной способности, доступной для целой стойки машин.

Еще один ресурс, о котором следует помнить, – файловые дескрипторы. Мы могли бы дать уравнение и коэффициенты для расчетов, но в наши дни файловые дескрипторы не являются дефицитным ресурсом, поэтому мы просто советуем установить параметр `ulimit` равным миллиону и забыть об этом.



Изменения параметра `ulimit` для файловых дескрипторов имеют странную привычку не применяться, в зависимости от особенностей запуска службы. Сервер Prometheus регистрирует ограничение `ulimit` при запуске, и вы можете проверить зарегистрированное значение `process_max_fds` в `/metrics`.

Эти цифры являются лишь отправной точкой. Обязательно проверяйте, насколько хорошо они соответствуют вашему окружению. Обычно мы рекомендуем выделять ресурсы для Prometheus с прицелом на удвоение их потребления, чтобы иметь время для приобретения нового оборудования по мере роста, а также некоторый запас прочности на случай внезапного увеличения кардинальности.

Управление конфигурацией

Prometheus решает только одну задачу – мониторинг на основе метрик, – и решает ее хорошо. Он не пытается брать на себя управление конфигурацией, секретами или обслуживающей базой данных. В этом отношении Prometheus старается не путаться под ногами и позволяет использовать стандартные подходы к управлению конфигурацией, не заставляя изучать и использовать какие-то специфические для Prometheus инструменты управления конфигурацией.

Если вы еще не выбрали инструмент управления конфигурацией, то мы рекомендуем Ansible для традиционных окружений. Для Kubernetes многообещающим выглядит Pulumi (<https://oreil.ly/TwdL>), но вообще в этом пространстве есть десятки неплохих инструментов.

Тот факт, что Prometheus поддерживает стандартные решения, не означает, что он будет автоматически идеально работать в вашем окружении. Универсальность лишь помогает избегать искушения учитывать нюансы, характерные для конкретной платформы. Это означает, что если у вас имеется зрелое окружение с устоявшимся набором инструментов, то вы довольно легко сможете развернуть Prometheus. Вы можете рассматривать Prometheus как тест на зрелость вашего инструмента управления конфигурацией, потому что Prometheus – это стандартный двоичный файл Unix, который работает

именно так, как можно было бы ожидать. Он принимает сигналы SIGTERM и SIGHUP, выводит сообщения в стандартный вывод ошибок и использует простые текстовые файлы для настройки¹.

Например, файлы правил Prometheus (обсуждаемые в главе 17) могут храниться только в файлах на диске. Если вам понадобится API, с помощью которого можно было бы отправлять правила, то ничто не мешает вам создать такую систему² и заставить ее выводить файлы правил в стандартном формате YAML. Однако сам сервер Prometheus не имеет такого API. Поддерживая только файлы на диске, Prometheus упрощает отладку, поскольку вы точно знаете, с какими настройками он работает. Тем, у кого простые настройки, не нужно беспокоиться о сложностях управления конфигурацией, а тем, кому нужно что-то более причудливое, создают интерфейс, позволяющий им делать все что заблагорассудится. Иными словами, бремя использования сложных и нестандартных конфигураций несут те, кому это необходимо, а не все остальные.

В более простых конфигурациях можно обойтись без статического файла *prometheus.yml*. Но по мере расширения вам придется создать шаблон с помощью вашей системы управления конфигурацией, чтобы как минимум указать разные *external_labels* для каждого сервера Prometheus, поскольку сам Prometheus не поддерживает шаблоны конфигурационных файлов. Для тех, кто еще не полностью освоил систему управления конфигурацией³, отметим, что некоторые среды выполнения могут предоставлять переменные окружения для приложений, работающих в них. Для создания простейших шаблонов можно использовать такие инструменты, как *sed* или *envsubst*⁴. С другой стороны в вашем распоряжении есть такие инструменты, как Prometheus Operator (кратко упоминавшийся в главе 9), способные управлять не только файлом конфигурации, но и сервером Prometheus, работающим в Kubernetes.

В главе 10 мы упоминали, что экспортёры должны размещаться как можно ближе к приложениям, из которых они экспортят метрики. Тому же принципу желательно следовать при работе с любыми демонами, предоставляемыми конфигурационные данные серверу Prometheus, такими как механизм обнаружения служб на основе файлов (обсуждается в разделе «Обнаружение служб на основе файла» главы 8). Запуская такие демоны рядом с каждым сервером Prometheus, вам придется заботиться только о машине, на которой работает Prometheus.

Для тестирования изменений в конфигурации Prometheus можно запустить тестовый сервер Prometheus. Поскольку Prometheus основан на идеи извлечения метрик, вашим целям не нужно знать или заботиться о поддерж-

¹ Пользователи Windows могут использовать HTTP вместо SIGTERM и SIGHUP, для чего сервер должен запускаться с флагом `--web.enable-lifecycle`.

² Такая система действительно существует – mixtool server (<https://oreil.ly/Dqc2B>), – но она пока очень нестабильная.

³ Во избежание путаницы отметим, что такие системы, как Docker, Docker Compose и Kubernetes, не являются системами управления конфигурациями; они являются потенциальными источниками данных для системы управления конфигурацией.

⁴ Часть библиотеки gettext.

ке мониторинга. При этом перед тестированием целесообразно удалить из конфигурационного файла все диспетчеры уведомлений Alertmanager или удаленные конечные точки записи.

Сети и аутентификация

Система мониторинга Prometheus разрабатывалась с учетом того, что она будет находиться в той же сети, что и цели мониторинга, сможет напрямую взаимодействовать с ними через HTTP и запрашивать их метрики. Эта модель называется мониторингом на основе извлечения метрик по запросу (pull) и имеет такие преимущества, как индикация успешности опроса, возможность запустить тестовый Prometheus без необходимости настраивать все цели для активной отправки (push) метрик, а также дополнительные тактические возможности для обработки внезапного увеличения нагрузки, как описано в разделе «Управление производительностью» ниже.

Если в вашем окружении присутствует брандмауэр, осуществляющий преобразование сетевых адресов (NAT) или фильтрацию сетевого трафика, то запускайте сервер Prometheus за ним, чтобы он мог напрямую обращаться к целям. Существуют также такие варианты, как PushProx (https://oreil.ly/nq_fp), туннели SSH или организация доступа Prometheus к целям через прокси-сервер с настройкой в поле конфигурации `proxy_url`.

 Не пытайтесь использовать Pushgateway для обхода сетевой архитектуры и вообще для превращения Prometheus в систему на основе модели push.

Как отмечалось в разделе «Pushgateway» главы 4, кеш метрик Pushgateway предназначен для мониторинга пакетных заданий сервисного уровня, которые отправляют метрики один раз непосредственно перед завершением. Он не предназначен для мониторинга экземпляров приложений, регулярно передающих метрики. И вы сами никогда не должны отправлять в Pushgateway метрики с меткой `instance`, потому что в таких случаях появляется узкое место¹, отметки времени образцов будут устанавливаться неправильно (что приведет к артефактам на графике), и теряется метрика `up`, что не затрудняет определение причины остановки процесса – намеренно или из-за сбоя. Pushgateway также не имеет логики определения устаревших данных, потому что для пакетных заданий сервисного уровня запуск из cron месяц назад не изменяет достоверности последней метрики с временем последнего успешного выполнения.

Кроме того, в основу Prometheus уже заложена модель извлечения метрик по опросу (pull), поэтому работайте с ней, а не против нее.

Как рассказывалось в главе 20, Prometheus предлагает некоторую встроенную поддержку безопасности. Однако общепринято защищать сервер Prometheus, располагая его за обратным прокси, что обеспечивает большую

¹ По тем же причинам, почему желательно запускать StatsD Exporter для каждого экземпляра приложения, а не для всего центра обработки данных.

гибкость и дополнительные функциональные возможности, такие как встроенная поддержка шифрования, отсутствующая в Prometheus. Обычно для этого используются такие прокси-сервера, как Traefik, Caddy, nginx или httpd, предлагающие широкий спектр функций поддержки безопасности. Вам может также понадобиться использовать прокси-сервер для блокировки доступа к конечным точкам администрирования и жизненного цикла Prometheus, защиты от подделки межсайтовых запросов (Cross-Site Request Forgery, XSS) и использования заголовков HTTP для защиты от межсайтового скрипtingа (Cross-Site Scripting, XSS).

Чтобы запустить Prometheus за обратным прокси, необходимо передать серверу Prometheus флаг `--web.external-url` с URL, через который он будет доступен. Это необходимо для корректной работы пользовательского интерфейса Prometheus и генератора URL в механизме оповещений. Если обратный прокси изменяет путь в HTTP-запросе перед пересылкой его в Prometheus, то передайте флаг `--web.route-prefix` с новым префиксом путей.

 Как и Prometheus, Alertmanager тоже поддерживает флаги `--web.external-url` и `--web.route-prefix`.

Prometheus и Alertmanager не поддерживают аутентификацию для обслуживания, но поддерживают ее для взаимодействия с другими системами, включая отправку оповещений, рассылку уведомлений, обнаружение служб, удаленное чтение, удаленную запись и опрос, как было описано в разделе «Как извлекать метрики» главы 8.

Планирование сбоев

В распределенных системах сбой – это жизненный факт. Разработчики Prometheus не пошли по пути кластерного дизайна для обработки сбоев машины, поскольку такие решения сложно реализовать правильно, и они часто оказываются менее надежными, чем некластерные. Также Prometheus не пытается повторно получить данные в случае завершения опроса неудачей. Если сбой произошел из-за перегрузки, то повторная попытка после снижения нагрузки может привести к повторной перегрузке. Лучше, когда системы мониторинга оказывают предсказуемую нагрузку и не усугубляют сбои.

Согласно такому решению если опрос завершится неудачей, то метрика `up` для этого опроса получит значение `0`, и во временном ряду появится пропуск. Но нет причин беспокоиться об этом. Вас мало будет заботить подавляющее большинство образцов, включая пропуски, через неделю после их сбора (если не раньше). Что касается мониторинга, то Prometheus придерживается позиции, что гораздо важнее, чтобы мониторинг в целом осуществлялся надежно и его данные были доступны, а не на 100 % точны. Для мониторинга на основе метрик в подавляющем большинстве случаев достаточно точности 99.9 %. Обычно гораздо полезнее знать, что задержка увеличилась на милли-

секунду, чем конкретное число миллисекунд – 101.2 или 101.3, – до которого она увеличилась. Функция `rate` устойчива к случайным неудачным попыткам опроса, если оцениваемый диапазон хотя бы в четыре раза превышает интервал опроса, как обсуждалось в разделе «`rate`» главы 16.

При обсуждении надежности первым делом следует определить, насколько надежным должен быть мониторинг. Осуществляя мониторинг системы с уровнем доступности 99.9 %, нет смысла тратить время и силы на разработку и поддержку системы мониторинга с уровнем доступности 99.9999 %. Даже если вы сможете построить такую систему, все равно ни интернет-соединения ваших пользователей, ни скорость реакции дежурного персонала не смогут обеспечить такой надежности.

В Европе, например, для рассылки уведомлений принято использовать короткие сообщения SMS, так как это быстро, дешево и надежно. Однако каждый год этот канал связи становится ненадежным на несколько часов, когда вся страна поздравляет друг друга с Новым годом. Из-за этого общая годовая надежность SMS оказывается ниже 99.95 %. У вас могут быть предусмотрены резервные каналы передачи информации для подобных случаев, но пока вы пытаетесь инициализировать их и отправить уведомления, часы продолжают тикать. Как упоминалось в разделе «`for`» главы 18, если у вас есть проблема, требующая решения менее чем за 5 мин, то ее исправление необходимо автоматизировать, а не надеяться, что дежурные инженеры сумеют решить ее вовремя.

В этом контексте мы хотели бы поговорить о надежности оповещения. На случай остановки сервера Prometheus по той или иной причине у вас должен иметься механизм его автоматического перезапуска. При этом никакие данные не должны потеряться, за исключением текущего состояния (как описано в разделе «`for`» главы 18)¹. Но если выйдет из строя машина, где работал сервер Prometheus, и нет возможности перезапустить ее, то вы не получите уведомлений, пока не замените ее. При использовании кластерного планировщика, такого как Kubernetes, можно ожидать, что это произойдет довольно быстро². Если же замена осуществляется вручную, то этот вариант, вероятно, будет неприемлем.

Вы можете легко повысить надежность оповещения, устранив единую точку отказа. Если запустить два одинаковых сервера Prometheus, то, пока хотя бы один из них работает, оповещения будут формироваться, а Alertmanager автоматически удалит повторяющиеся оповещения, имеющие одинаковые метки.

Как упоминалось в разделе «Внешние метки» главы 18, каждый сервер Prometheus должен иметь уникальные внешние метки, для чего можно использовать файлы `alert_relabel_configs` (как описано в разделе «Настройка диспетчеров уведомлений в Prometheus» главы 18):

¹ По этой причине мы советуем спроектировать критически важные оповещения так, чтобы они запускались в свежем экземпляре Prometheus в течение часа, если не раньше.

² При использовании сетевого хранилища, такого как Amazon EBS, Prometheus может даже продолжить работу с данными, оставшимися от предыдущего запуска.

```

global:
  external_labels:
    region: dublin1
alerting:
  alertmanagers:
    - static_configs:
      - targets: ['localhost:9093']
  alert_relabel_configs:
    - source_labels: [region]
      regex: (.+)\d+
      target_label: region

```

С такими настройками Prometheus удалит 1 из значения `dublin1` перед отправкой оповещения в Alertmanager. Второй Prometheus будет иметь внешнюю метку `region` со значением `dublin2`.

Я уже несколько раз упоминал, что внешние метки должны быть уникальными для всех серверов Prometheus. Причина этого требования проста: если в вашем окружении имеется несколько серверов Prometheus с конфигурацией, подобной той, что показана выше, и вы используете удаленную запись или федерацию, то метрики, поступающие с разных серверов Prometheus, не будут конфликтовать между собой. Даже в идеальных условиях разные серверы Prometheus будут иметь немного разные данные, которые можно неверно истолковать, например как сброс счетчика. В менее оптимальных условиях, как, например, при работе в сегментированной сети, резервные серверы Prometheus могут получать совершенно разную информацию.

Эти рассуждения подводят нас к вопросу надежности дашбордов, федерации и удаленной записи. Не существует общего способа автоматического синтеза «правильных» данных на основе информации, получаемой с разных серверов Prometheus, а использование балансировщика нагрузки для Grafana или федерации приведет к появлению артефактов. Я предлагаю более простой путь и использовать информацию для дашбордов/федерации/записи только с одного из серверов Prometheus, а если он не работает, то смириться с возникшим пропуском. В том редком случае, когда пропуск в данных охватывает интересующий вас период, всегда можно просмотреть данные в другом Prometheus вручную.

Компромиссы для глобальных серверов Prometheus немного отличаются, как обсуждалось в разделе «Выход на глобальный уровень с помощью механизма федерации» выше. Поскольку глобальные серверы Prometheus осуществляют мониторинг в зонах отказа, то вполне вероятно, что какой-то из глобальных серверов может отключиться на несколько часов или дней, если, например, в центре обработки данных произошло отключение электроэнергии из-за серьезной аварии. Для серверов Prometheus в центрах обработки данных в этом нет ничего плохого, потому что не работают не только они, но и все остальное, за чем они должны наблюдать. Мы рекомендуем всегда запускать как минимум два глобальных сервера Prometheus в разных центрах обработки данных, чтобы графики были доступны со всех глобальных серверов.

То же относится и к удаленной записи¹. Ответственность за интерпретацию данных из различных источников лежит на лице, использующем дашборды.

Кластеризация диспетчеров уведомлений

Обычно желательно иметь один централизованный диспетчер уведомлений Alertmanager во всей организации, чтобы иметь одно место для просмотра и отключения оповещений и получить максимальные преимущества от группировки оповещений. Но если организация слишком большая для этого, то можно воспользоваться поддержкой кластеризации Alertmanager, архитектура которой показана на рис. 21.2.

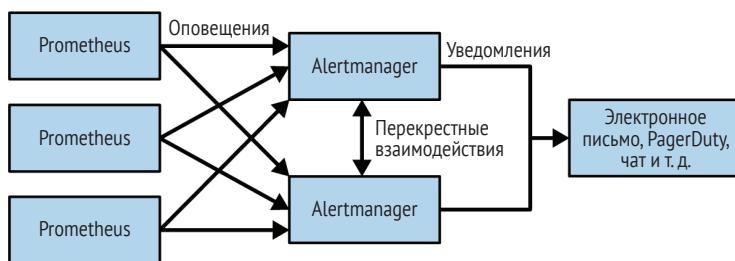


Рис. 21.2 ♦ Архитектура кластеризации Alertmanager

Для распространения уведомлений и отключения оповещений² Alertmanager использует список участников HashiCorp (<https://oreil.ly/fbRs>)³. Эта архитектура не основана на консенсусе, поэтому необязательно иметь нечетное количество диспетчеров Alertmanagers. Она известна как архитектура AP (Availability and Partition-tolerant – доступная и устойчивая к сегментированию), поэтому, пока сервер Prometheus сможет взаимодействовать хотя бы с одним диспетчером Alertmanager, способным благополучно отправлять уведомления, ваши уведомления будут отправляться. В редких случаях при наличии таких сложностей, как сегментирование сети, вы можете получать повторяющиеся уведомления, но это лучше, чем вообще ничего не получать.

Каждый сервер Prometheus, работающий в кластере, должен отправлять свои оповещения каждому диспетчеру Alertmanager. В таких случаях диспетчеры Alertmanager сами определяют приоритеты. Первый Alertmanager рассыпает уведомления как обычно и в случае успеха сообщает другим, что

¹ Глобальные серверы Prometheus находятся на вершине иерархии федерации, поэтому от них обычно ничего не зависит.

² Помимо перекрестных взаимодействий, Alertmanager также хранит данные на локальном диске, поэтому даже в некластерном окружении вы не потеряете текущего состояния при перезапуске Alertmanager.

³ До версии Alertmanager 0.15.0 использовалась библиотека Weaveworks Mesh.

уведомление отправлено. Второй Alertmanager выполняет небольшую задержку перед отправкой уведомлений. Если в течение этого времени он не получит сообщения от первого Alertmanager об отправке уведомления, то отправит его. Третий Alertmanager выполняет еще большую задержку и т. д. Все диспетчеры Alertmanager должны иметь одинаковые файлы `alertmanager.yml`, но самое худшее, что может случиться при использовании отличающихся конфигураций, – они будут отправлять повторяющиеся уведомления.

Вот как можно запустить два экземпляра Alertmanager 0.24.0 на машинах с именами `foo` и `bar`:

```
# На машине foo
alertmanager --cluster.peer bar:9094

# На машине bar
alertmanager --cluster.peer foo:9094
```

Самый простой способ проверить правильную работу в кластере – отключить оповещение в одном Alertmanager и посмотреть, появится ли оно в другом. Также можно посмотреть список всех членов кластера на странице состояния Alertmanager.

Мета- и перекрестный мониторинг

К настоящему моменту мы рассмотрели организацию мониторинга систем разных типов, но пока не рассмотрели мониторинг самой системы мониторинга. Обычно каждый сервер Prometheus опрашивает сам себя, но это ничем не поможет, если проблема возникнет на самом сервере Prometheus. Контролирование системы мониторинга называется *метамониторингом*.

Типичное решение метамониторинга заключается в запуске одного сервера Prometheus в каждом центре обработки данных, который наблюдает за всеми остальными серверами Prometheus в этом центре. Это не обязательно должен быть выделенный сервер Prometheus, так как мониторинг в Prometheus обходится довольно дешево, но, даже если в вашем окружении каждая команда несет полную ответственность за работу своих серверов Prometheus, все равно целесообразно организовать метамониторинг в виде общей центральной службы.

В таком случае глобальный сервер Prometheus может опрашивать остальные серверы метамониторинга в центрах обработки данных через конечные точки `/metrics` и объединять агрегированные метрики, информирующие о состоянии всех серверов Prometheus в вашей организации.

Но остается открытый вопрос: как контролировать работоспособность глобальных серверов Prometheus. *Перекрестный мониторинг* – это метамониторинг, когда серверы Prometheus контролируют друг друга, причем не только одного «уровня». Например, во многих окружениях имеется два глобальных сервера Prometheus, которые опрашивают конечные точки `/metrics` друг друга и генерируют оповещение, если какой-то из них оказался недо-

ступным. Однако также можно сделать так, чтобы серверы Prometheus центра контролировали работоспособность глобальных серверов Prometheus¹.

Даже после настройки мета- и перекрестного мониторинга вы остаетесь зависимыми от работоспособности серверов Prometheus, контролирующих серверы Prometheus. В худшем сценарии ошибка может вывести из строя все ваши серверы Prometheus одновременно, поэтому было бы разумно иметь механизм, способный это обнаружить. Один из возможных подходов – сквозное тестирование оповещений. Постоянное поступающее оповещение будет постоянно отправлять уведомление через вашего провайдера передачи мгновенных сообщений на аварийный переключатель. Этот переключатель отправит вам сообщение², если не получит уведомление в течение достаточно долгого периода времени. Это поможет вам контролировать работоспособность цепочки: Prometheus, Alertmanager, сеть и провайдер передачи мгновенных сообщений.

При разработке схемы метамониторинга не забудьте предусмотреть опрос других компонентов, связанных с мониторингом, таких как Alertmanager и /metrics экспортёров Blackbox/SNMPstyle.

Управление производительностью

В любой достаточно крупной системе появление проблемы с производительностью – это лишь вопрос времени. Как обсуждалось в разделе «Кардинальность» главы 5, одна из основных причин проблем с производительностью – метрики с высокой кардинальностью.

Недостаточная производительность также может быть обусловлена использованием в правилах записи и дашбордах дорогостоящих запросов, например обрабатывающих большие объемы данных в протяженных диапазонах времени, как описано в разделе «Гистограммы» главы 13. Для поиска дорогостоящих правил записи можно использовать страницу состояния правил, как показано на рис. 17.1³.

Обнаружение проблем

Prometheus предоставляет множество метрик, характеризующих его работу, поэтому вам не придется полагаться только на субъективную оценку про-

¹ При этом вы должны гарантировать, что оповещения о недоступности глобального сервера Prometheus будут иметь одинаковые метки и дубликаты оповещений автоматически отбрасываются диспетчером уведомлений Alertmanager. Одно из возможных решений – добавление в оповещение явной метки `datacenter: global` (или какой-то другой), чтобы предотвратить применение внешней метки `datacenter`.

² Желательно не только через обычного провайдера мгновенных сообщений, так как он тоже может оказаться недоступным.

³ Например, по продолжительности выполнения.

изводительности ваших дашбордов или на появление тайм-аутов. Метрики могут и меняют названия и значения от версии к версии, но обычно не исчезают полностью.

Метрика `prometheus_rule_group_iterations_missed_total` может служить индикатором, что для оценки некоторых групп правил требуется слишком много времени. Сравнение `prometheus_rule_group_last_duration_seconds` с `prometheus_rule_group_interval_seconds` поможет определить группу, ответственную за замедление, и случилось ли это замедление недавно.

Метрика `prometheus_notifications_dropped_total` указывает на проблемы взаимодействия с Alertmanager, а при приближении значения `prometheus_notifications_queue_length` к значению `prometheus_notifications_queue_capacity` могут начать теряться оповещения.

Каждый механизм обнаружения служб, как правило, имеет метрику, такую как `prometheus_sd_file_read_errors_total` или `prometheus_sd_ec2_refresh_failures_total`, указывающую на проблемы. Следите за счетчиками используемых вами механизмов обнаружения служб.

Метрики `prometheus_rule_evaluation_failures_total`, `prometheus_tsdb_compactions_failed_total` и `prometheus_tsdb_wal_corruptions_total` сообщают о проблемах на уровне хранилища. В худшем случае вы всегда можете остановить Prometheus, удалить¹ каталог хранилища и снова запустить сервер.

Поиск дорогостоящих метрик и целей

Как упоминалось в разделе «by» главы 14, есть возможность использовать такие запросы:

```
topk(10, count by(__name__)({__name__=~".+"}))
```

для поиска метрик с высокой кардинальностью. Также можно выполнить агрегирование по метке `job`, чтобы определить, какие приложения генерируют наибольшее количество временных рядов. Но эти запросы могут быть очень дорогими, потому что просматривают каждый временной ряд, и их следует использовать с осторожностью.

В дополнение к `up` Prometheus предоставляет еще три значения для каждой опрашиваемой цели. `scrape_samples_scraped` – это количество образцов в `/metrics`. Поскольку это один временной ряд для каждой цели, его анализ обходится намного дешевле, чем выполнение предыдущего выражения PromQL. `scrape_samples_post_metric_relabeling` – похожая метрика, но не включает образцы, которые были удалены настройками в `metric_relabel_configs`.

И последний дополнительный образец – `scrape_duration_seconds` – определяет время, затраченное на опрос. Эта метрика может пригодиться для проверки появления тайм-аутов, которые могут указывать на перегруженность цели.

¹ Или переименовать, чтобы создать резервную копию существующих данных.

Hashmod

Если ваш сервер Prometheus настолько перегружен извлечением огромного объема данных во время опроса, что нет возможности выполнять запросы, то можно попробовать организовать опрос подмножества целей. Механизм изменения меток поддерживает еще одно действие – `hashmod`, которое вычисляет хеш метки и берет остаток от деления на заданное число. В сочетании с `drop` это действие можно использовать для опроса произвольных 10 % целей:

```
scrape_configs:
  - job_name: my_job

    # Здесь находятся настройки обнаружения служб и т. д.

  relabel_configs:
    - source_labels: [__address__]
      modulus: 10
      target_label: __tmp_hash
      action: hashmod
    - source_labels: [__tmp_hash]
      regex: 0
      action: keep
```

Теперь, когда в каждом цикле опрашивается всего 10 % целей, можно запустить тестирование сервера Prometheus и выяснить, какая метрика ответственна за потерю производительности. Если проблему вызывают только некоторые цели, то можно указать, какие 10 % целей следует опрашивать, изменив регулярное выражение в `regex` на 1, 2 и т. д. до 9.

Снижение нагрузки

Выявив дорогостоящие метрики, у вас на выбор есть несколько вариантов. Первое, что можно сделать, – попытаться исправить метрику в исходном коде, чтобы уменьшить ее кардинальность.

Но до этого момента можно применить несколько тактических решений. Первое – исключить метрику из опроса с помощью `metric_relabel_configs`:

```
scrape_configs:
  - job_name: some_application
    static_configs:
      - targets:
        - localhost:1234
  metric_relabel_configs:
    - source_labels: [__name__]
      regex: expensive_metric_name
      action: drop
```

В этом случае метрика, как и прежде, передается по сети и анализируется, а экономия достигается за счет отказа от ее передачи на уровень хранения¹.

Если проблемы вызывают конкретные приложения, то соответствующие цели можно исключить с помощью механизма изменения меток.

Последнее решение – увеличить `scrape_interval` и `evaluation_interval`. Это может дать вам немного времени на передышку, но имейте в виду, что эти параметры нецелесообразно увеличивать более чем на 2 мин. Кроме того, изменение интервала опроса (`scrape_interval`) может сделать неработоспособными некоторые выражения PromQL, зависящие от конкретного значения в этом параметре.

В конфигурации опроса есть еще один полезный параметр – `sample_limit`. Если количество образцов, оставшихся после изменения меток², превысит `sample_limit`, то опрос завершится ошибкой и образцы не будут загружены. По умолчанию этот параметр отключен, но может действовать как аварийный предохранительный клапан, если кардинальность одной из ваших целей внезапно увеличится, например из-за добавления метрики с идентификатором клиента в качестве метки. Это параметр не предназначен для тонкой настройки или построения какой-либо системы квот; если вы собираетесь его использовать, то установите достаточно большое значение, которое редко придется менять.

Мы советуем предусматривать достаточный запас мощности на своих серверах Prometheus, чтобы благополучно справляться с умеренными всплесками кардинальности и целей.

Горизонтальное сегментирование

Если проблема масштабирования обусловлена кардинальностью метки `instance`, а не инструментированных меток, то можно попробовать организовать горизонтальное сегментирование серверов Prometheus с помощью действия `hashmod` механизма изменения меток, как было показано в разделе «Hashmod» выше. Этот подход обычно необходим только при наличии многих тысяч целевых приложений одного типа, поскольку иначе предпочтительнее использовать вертикальное сегментирование как более простой метод масштабирования Prometheus (см. раздел «Расширение Prometheus» выше).

Идея горизонтального сегментирования состоит в том, чтобы запустить ведущий сервер Prometheus и несколько подчиненных серверов, каждый из которых осуществляет опрос подмножества целей:

¹ Клиенты на Java и Python поддерживают выборку определенных временных рядов с использованием параметров URL, таких как `/metrics?metric[]=process.cpu_seconds_total`. Эта возможность может не поддерживаться некоторыми нестандартными сборщиками, но в общем случае способна сэкономить много ресурсов по обе стороны, если количество действительно необходимых вам метрик невелико.

² То есть значение `scrape_samples_post_metric_relabeling`.

```
global:
  external_labels:
    env: prod
    scraper: 2
scrape_configs:
  - job_name: my_job

# Здесь находятся настройки обнаружения служб и т. д.

relabel_configs:
  - source_labels: [__address__]
    modulus: 4
    target_label: __tmp_hash
    action: hashmod
  - source_labels: [__tmp_hash]
    regex: 2 # Этой 3-й сервер, выполняющий опрос.
    action: keep
```

Здесь, как можно видеть по значению параметра `modulus`, действуют четыре сервера опроса. Каждый из них должен иметь уникальную внешнюю метку плюс внешние метки ведущего сервера Prometheus. В такой конфигурации ведущий сервер Prometheus может использовать конечную точку удаленного чтения для прозрачного извлечения данных из серверов опроса:

```
global:
  external_labels:
    env: prod
remote_read:
  - url: http://scraper0:9090/api/v1/read
    read_recent: true
  - url: http://scraper1:9090/api/v1/read
    read_recent: true
  - url: http://scraper2:9090/api/v1/read
    read_recent: true
  - url: http://scraper3:9090/api/v1/read
    read_recent: true
```

Механизм удаленного чтения поддерживает оптимизацию, которая предотвращает попытки чтения данных из удаленного источника, если они доступны локально. Использовать такую оптимизацию имеет смысл для работы с системой долговременного хранения. `read_recent: true` отключает ее. Благодаря добавлению внешних меток метрики из каждого сервера опроса будут получать метку `scraper` со значением, определяющим источник данных.

Здесь актуальны все те же предостережения, что и в отношении федерации, описанные в разделе «Выход на глобальный уровень с помощью механизма федерации» выше. Этот прием не предназначен для организации общего сервера Prometheus, позволяющего получить прозрачный доступ ко всем другим вашим серверам Prometheus. На самом деле это был бы отличный способ выключить всю инфраструктуру мониторинга, запустив единственный дорогостоящий запрос. При использовании этого метода все, что может потребоваться, лучше агрегировать на стороне серверов опроса (как описано

в разделе «Уменьшение кардинальности» главы 17), чтобы уменьшить объем данных, извлекаемых ведущим сервером.

Используя горизонтальное масштабирование, важно не скучиться на количество серверов опроса и увеличивать его каждые несколько лет. При этом увеличивать желательно не менее чем в два раза, чтобы избежать необходимости повторного увеличения в ближайшее время.

Управление изменениями

Со временем вы неизбежно столкнетесь с необходимостью изменить структуру целевых меток, чтобы она соответствовала архитектуре ваших систем. Приложения, содержащие метрики, которые предназначены для планирования емкости, со временем будут меняться, разделяясь и объединяясь в ходе естественного развития, а метрики будут появляться и исчезать.

Управлять изменением целевых меток с целью их адаптации под требования новой иерархии можно с помощью настроек в разделе `metric_label_configs`. Но со временем эти настройки будут накапливаться и в конечном итоге начнут вызывать больше путаницы, чем пользы.

Мы советуем признать, что такие изменения являются естественной частью эволюции системы, и, так же как в случае с пропусками из-за неудачных попыток опроса, вы обнаружите, что старые имена вас мало будут волновать постфактум.

С другой стороны, в долгосрочных процессах, таких как планирование мощностей, требуется иметь данные за довольно протяженный период. Для их поддержки желательно хотя бы записывать названия метрик, меняющиеся с течением времени, и подумать о возможности использования подхода, описанного в «Правилах для API» главы 17, при настройке глобального сервера Prometheus, если изменения происходят слишком часто, чтобы управлять ими вручную.

В этой главе вы узнали, как развернуть систему мониторинга Prometheus, в каком направлении ее развивать, как спроектировать архитектуру Prometheus и запустить ее и как решать возникающие проблемы производительности.

Получение помощи

После прочтения этой книги у вас могут возникнуть вопросы. Есть несколько мест, где вы сможете получить ответы на них. На странице сообщества веб-сайта Prometheus (<https://oreil.ly/lqZT8>) перечислены официальные методы связи с проектами Prometheus, такие как список рассылки `prometheus-users` (<https://oreil.ly/CTPNp>), где пользователи могут задавать вопросы и получать ответы. Есть также неофициальные дискуссионные площадки, в том числе StackOverflow (тег Prometheus; <https://oreil.ly/MGxtQ>), канал `#prometheus` на

CNCF Slack (<https://slack.cncf.io/>) и PrometheusMonitoring subreddit (<https://oreil.ly/nDPxu>). Наконец, на странице «Поддержка и обучение» (https://oreil.ly/XEg_F) приводится список компаний и частных лиц, предлагающих коммерческую поддержку, в том числе компания Брайана Robust Perception (<https://oreil.ly/X90Yd>) и компания Жюльена O11y (<https://o11y.eu/>).

Мы надеемся, что эта и все предыдущие главы были вам полезны и мониторинг с использованием Prometheus поможет облегчить вам жизнь.

Предметный указатель

Символы

`^`, оператор возведения в степень, 277
`==`, оператор, 48
`|`, оператор чередования, 155
`$labels`, 315
`$value`, 315
`@`, модификатор, 240
`_address_`, метка, 161, 162
`--collector.filesystem.fs-types-exclude`, флаг, 133
`--collector.filesystem.mount-points-exclude`, флаг, 133
`--collector.textfile.directory`, параметр командной строки, 141
`/metrics`, путь, 78
`_metrics_path_`, 166, 182
`_name_`, 169
`_name_`, метка, 235
`_param_<name>`, 166
`/probe`, путь зондирования Blackbox, 196
`/proc/meminfo`, 136
`/proc/stat`, 137
`_scheme_`, 166, 182
`--timeout-offset`, параметр командной строки (Blackbox Exporter), 208
`--web.enable-lifecycle`, флаг, 295, 361
`--web.route-prefix`, флаг, 363

А

`abs`, функция, 275
`absent`, функция, 283
`absent_over_time`, функция, 284
`acos`, функция, 278
`acosh`, функция, 278
`alert`, поле, 308
`alerting`, поле, 319

`Alertmanager`, 321
веб-интерфейс, 341
дерево маршрутизации, 323
загрузка и установка, 50
конвейер уведомлений, 321
конфигурационный файл, 323
дерево маршрутизации, 330
минимальный пример, 323
приемники, 330
настройка, 50
`alertmanagers`, поле, 319
`alertmanager.yml`, файл, 50
`alertname`, метка, 309
`alert_relabel_configs`, 319
`ALERTS`, метрика, 309
`Alerts`, список, 335
`ALERTS_FOR_STATE`, метрика, 309
`Amazon EC2`, 144
`AmazonEC2ReadOnlyAccess`, 152
`and`, оператор, 271, 309
`annotations`, поле, 315
`Ansible`, 144, 146, 360
`asin`, функция, 278
`asinh`, функция, 279
`atan2`, оператор, 260
`atan`, функция, 279
`atanh`, функция, 279
`authorization`, 166, 176
`avg`, 251
`avg_over_time`, 105, 237, 292, 312
принимает векторы диапазонов, 273
`avg without`, выражение, 133, 229

В

`backend-ticket`, приемник, 325
`basic_auth`, 166

Blackbox Exporter, экспортер, 194
настройка в Prometheus, 205
тайм-аут, 208
blackbox.yml, 196
Blakbox Exporter и реестр по умолчанию, 222
bool, модификатор, 261
bottomk, 254
by, модификатор, 248
by, оператор, 104

C

cAdvisor, 170, 172
метки, 172
метрики потребления памяти, 172
процессора, 171
CAP_NET_RAW, 195
Cassandra, 141
ceil, функция, 277
cert_file, 165
cgroups, 170
changes, функция, 289
check-config (amtool), 323
check config (promtool), 295
check rules (promtool), 295
Chef, 144
clamp, функция, 278
clamp_max, функция, 278
clamp_min, функция, 278
CloudWatch Exporter, экспортер, 210
cluster, метка, 247
Collect, метод, 220
collectd, система мониторинга, 209
совместимость с Prometheus, 209
Collectd Exporter, экспортер, 209
CollectorRegistry.collect, 90
CollectorRegistry.
metricFamilySamples, 90
configMap, объект Kubernetes, 174
connection refused (отказано в соединении), ошибка, 47
ConstMetrics, 29
Consul, 144, 166, 187
настройка агента, 151
экспортер Consul Telemetry, 215
consul_catalog_service_node_healthy, 188
Consul Exporter, 187
метрики, 188

consul_exporter_build_info, 188
consul_serf_lan_members, 188
consul_up, 188, 220
container_, префикс, 171
container_cpu_system_seconds_total, 171
container_cpu_usage_seconds_total, 171
container_cpu_user_seconds_total, 171
container_label_, префикс, 172
container_memory_cache, 172
container_memory_rss, 172
container_memory_usage_bytes, 172
container_memory_working_set_bytes, 172
container_spec_memory_limit_bytes, 172
Content-Type, 150
context deadline exceeded (превышен крайний срок контекста) ошибка, 47
continue, параметр (маршрутизация), 326
cos, функция, 279
cosh, функция, 279
count, 250
CounterValue, 220
count_over_time, 292
count_values, 255
cpu, метка, 132
CPython, 79
credentials_file, 166, 176
cron, задания, 140
curl, утилита, 82

D

dashbord, аннотация, 337
dec, метод, 61
DefaultExports.initialize, 84
deg, функция, 279
delete_from_gateway, 88
delta, функция, 291
Deployment, объект Kubernetes, 174
deriv, функция, 290
Desc, тип, 219
Describe, метод, 219
device, метка, 133, 228, 247
удаление с помощью sum without, 229
df, команда, 133
dns, зонд, 203
Docker, 170
запуск Node Exporter, 132
установка Grafana, 113

dont_fragment, 196
 DOWN, уведомление о состоянии, 47
 drop (действие изменения меток), 156
 Dropwizard, 211
 Dropwizard (метрики), счетчики, 216

E

EC2, 144
 Elasticsearch, 91
 email_to, метка, 338
 end(), функция, 240
 endpointslice, 178
 env, метка, 247, 327
 evaluation_interval, 296, 371
 exp, функция, 276
 external_labels, 325
 external_labels, поле, 319

F

files, поле, 295
 file_sd_configs, 148, 295
 float64, 221
 floor, функция, 277
 follow_redirects, 203
 for, поле, 310
 fork, системный вызов, 137
 frontend-pager, приемник, 331
 fstype, метка, 133, 228, 247

G

Gather, метод, 90
 GaugeHistograms (метрики), 95
 GaugeValue, 220
 get_sample_value, функция, 69
 Go
 запуск экспортера Consul Telemetry, 218
 разработка экспортёров, 215
 сборщики, 219
 система шаблонов, 315
 экспортирование из клиентских библиотек, 82
 язык шаблонов, 113
 Registry.Gather, 90
 WithValueLabels, 99
 go_, префикс, 188

Grafana, 112, 356
 возможность смещения панели в другой временной диапазон, 239
 дашборды и панели, 116
 источники данных, 115
 панель
 временной шкалы состояний, 125
 временных рядов, 118
 статистики, 121
 таблицы, 123
 параметр настройки reporting_enabled, 113
 переменные шаблона, 126
 установка, 113
 элементы управления временем, 119
 эффект искажения, 121
 Graphite, 90, 209
 мост, 90
 Graphite Exporter, 351
 Graphite Exporter, экспортёр, 210
 Grok Exporter, экспортёр, 191
 group, 252
 group_by, 326, 339
 group_interval, 328
 настройка, 328
 group_left, 107, 137
 group_left, модификатор, 265
 group_wait, 328
 Gunicorn, 79

H

handler, метка, 230
 hashmod, 370
 HELP (метрики), 91
 histogram_quantile, 255, 285
 histogram_quantile, функция, 66, 232
 holt_winters, функция, 291
 honor_labels, 169
 htpasswd, утилита, 348
 http_2xx, модуль, 201
 HTTP-запросы, разбивка по путям, 97
 http, зонд, 202
 HTTP-сервер на Python (пример), 54
 HTTP API, 240
 конечные точки query и query_range, 240
 HTTP Basic Authentication, 166
 HTTP Bearer Token Authentication, 166
 http_config, поле, 333

HTTP (HyperText Transfer Protocol – протокол передачи гипертекста), 201
`http_request_size_bytes`, 167
`http_requests_total`, 240
`http_response_size_bytes`, 230
`http_response_size_bytes_count`, 230
`http_response_size_bytes_sum`, 230
`HTTPServer`, класс, 83
`HttpServlet`, класс, 85

I

`icmp`, модуль, 196
ICMP (Internet Control Message Protocol – протокол управляющих сообщений интернета), 194
`id`, метка, 171
`id`, метка (`cAdvisor`), 172
`idefault`, функция, 291
`ignoring`, модификатор, 263
`image`, метка (`cAdvisor`), 172
`inc`, метод, 61
`increase`, функция, 229, 287
InfluxDB, 209
 совместимость с Prometheus, 209
InfluxDB Exporter, экспортер, 209, 211
Info (метрики), 95
ingress, роль, 184
`insecure_skip_verify`, 166, 176
instance, 157, 161
instance, метка, 38, 87, 247
 и группировка оповещений, 327
 удаление с помощью `sum without`, 228
InstanceDown, оповещение, 49
iptables, команда, 140
IPv4, 197
IPv6, 197
`irate`, функция, 229, 288

J

Java
экспортирование из клиентских библиотек, 83
 сервлеты, 85
 `HTTPServer`, класс, 83
`labels`, метод, 99
Jinja2, механизм шаблонов (Ansible), 146
JMX Exporter, экспортер, 210

JMX (Java Management eXtensions), стандарт, 210
job, 157
job, метка, 38, 162, 166, 247
`job_name`, 147, 149, 162
`job\process_cpu_seconds\rate5m`, 297
JSON, 147, 148, 215
JVM (Java Virtual Machine – виртуальная машина Java), 83

K

`keep` (действие изменения меток), 154
`key_file`, 165
`kill -HUP`, команда, 295
kubectl, 174
`kube_deployment_spec_replicas`, 185
`kube-dns`, 183
Kubelet, 175
`kube_node_status_condition`, 185
`kube_pod_container_status_restarts_total`, 185
`kube_pod_info`, 185
Kubernetes, 144, 170, 173
 и Prometheus, 173
 обнаружение служб, 175
 `endpointslice`, 178
 `ingress`, 184
 `node`, 175
 `pod`, 183
 `service`, 178
 управление конфигурацией, 360
kubernetes, служба, 178
`kube-state-metrics`, 188
`kube-state-metrics`, экспортер, 185

L

`labeleddrop` (действие изменения меток), 168, 172
`label_join`, функция, 283
`labelkeep` (действие изменения меток), 168
`labelmap` (действие изменения меток), 162
`label_replace`, функция, 282
`labels`, метод, 99, 103
`label vtnrf`, 137
`last_over_time`, 292, 312
`latency_too_high_threshold`, 338

le, 169
 le, метка, 231
 Linux, предлагаемые метрики, 131
 ln, функция, 276
 log2, функция, 276
 log10, функция, 276
 log_http_requests_total, 192
 LTS (Long Term Support – долгосрочная поддержка), 35

M

ManyInstancesDown, оповещение, 308
 max, 253

использование с датчиками, 228
 max_over_time, 238, 292, 312
 metric_relabel_configs, 167, 283
 metric_relabel_configs (действие изменения меток), 167
 metrics_app, 77
 metrics_path, 165, 166
 metrics.Samples, 221
 MetricsServlet, класс, 85
 min, 253
 Minikube, 173
 minikube dashboard --url, 174
 min_over_time, 292
 mmap, утилита, 81
 mode, метка, 132, 250, 265
 mountpoint, метка, 133, 228, 247
 MultiProcessCollector, 80
 multiprocess_mode, конфигурационный параметр (датчики), 80
 MustNewConstMetric, функция, 220, 223
 MustRegister, функция, 83
 MySQLD, экспортер, 189
 mysql_global_status_, префикс, 190
 mysql_global_variables_, префикс, 190

N

name, метка (cAdvisor), 172
 NaN (Not a Number – не число), 59
 Native Histograms, экспериментальная функция, 67
 NetBox, 144
 NewCounter, 83
 NewDesc, 219
 New Relic Exporter, экспортер, 210
 nice, режим, 133

node_boot_time_seconds, 137
 node_context_switches_total, 137
 node_cpu_guest_seconds_total, 133
 node_cpu_seconds_total, 132
 node_disk_io_now, 134
 node_disk_io_time_seconds_total, 134
 node_disk_read_bytes_total, 134, 282
 node_disk_reads_completed_total, 134
 node_disk_read_time_seconds_total, 134
 node_disk_writes_completed_total, 135
 node_disk_write_time_seconds_total, 135
 node_disk_written_bytes_total, 135
 Node Exporter, 42, 132, 350
 загрузка и установка, 43
 запуск в Docker, 132
 настройка для мониторинга Prometheus, 43
 сборщик
 дисковой статистики, 134
 информации
 об аппаратной части, 136
 об имени узла, 138
 об операционной системе, 138
 о давлении, 139
 о потреблении памяти, 136
 о процессоре, 132
 о сетевых устройствах, 135
 о средней нагрузке, 139
 о файловой системе, 133
 статистики, 137
 текстовых файлов, 140
 использование, 141
 отметки времени, 143
 node_filesystem_size_bytes, 227
 node_filesystem_, префикс, 133
 node_filesystem_avail_bytes, 134
 node_filesystem_files, 134
 node_filesystem_files_free, 134
 node_filesystem_free_bytes, 134
 node_filesystem_size_bytes, 248
 node_forks_total, 137
 node_hwmon_, префикс, 136
 node_hwmon_sensor_label, 137
 node_hwmon_temp_celsius, 137
 node_interrupts_total, 137
 node_intr_total, 137
 node_load1, 139
 node_load5, 139
 node_load15, 139

node_memory_, префикс, 136
node_memory_Buffers_bytes, 136
node_memory_Cached_bytes, 136
node_memory_MemFree_bytes, 136
node_memory_MemTotal_bytes, 136
nodename, метка, 138
node_network_receive_bytes_total, 135
node_network_receive_packets_total, 135
node_network_transmit_bytes_total, 135
node_network_transmit_packets_total, 135
node_os_info, 138
node_os_version, 138
node_pressure_cpu_waiting_seconds_total, 139
node_pressure_io_stalled_seconds_total, 139
node_pressure_io_waiting_seconds_total, 139
node_pressure_memory_stalled_seconds_total, 140
node_pressure_memory_waiting_seconds_total, 140
node_procs_blocked, 137
node_procs_running, 137
node_textfile_mtime_seconds, 143
node_uname_info, 138
NRPE Exporter, экспортер, 211

O

OAuth2, 166
observe, метод, 64
offset, модификатор, 239
on, модификатор, 264
on, оператор, 309
OpenMetrics, 28
поддержка в клиентских библиотеках на Python, 82
суффиксы в именах метрик, 62
or, оператор, 268

P

PagerDuty, приемник, 331
params, 166
path, метка, 97, 192
payload_size, 196
pi, функция, 279
ping, 195

Pingdom Exporter, экспортер, 210
pod, 183
predict_linear, функция, 290
present_over_time, 292
printf, функция, 315
probe_duration_seconds, 196
probe_failed_due_to_regex, 200
probe_ip_addr_hash, 196
probe_ip_protocol, 196
probe_ssl_last_chain_expiry_timestamp_seconds, 200
probe_success, 196
process_, префикс, 188
process_cpu_seconds_total, 233
process_open_fds, 222
process_resident_memory_bytes, 38, 45, 172
график в браузере выражений, 40
promauto, 83
Promdash, 113
Prometheus
 и Kubernetes, 173
 настройка
 диспетчеров уведомлений, 318
 для опроса Blackbox Exporter, 205
prometheus.Collector, интерфейс, 219
Prometheus Community Kubernetes Helm Charts, 186
prometheus-deployment.yaml, файл, 174
prometheus_http_response_size_bytes, 167
prometheus_multiproc_dir, переменная окружения, 81
prometheus.MustNewConstMetric, 220
prometheus_notifications_dropped_total, 369
prometheus_notifications_queue_capacity, 369
prometheus_notifications_queue_length, 369
Prometheus Operator, 186
prometheus_rule_evaluation_failures_total, 369
prometheus_rule_group_interval_seconds, 369
prometheus_rule_group_iterations_missed_total, 369
prometheus_rule_group_last_duration_seconds, 369

- `prometheus_sd_ec2_refresh_failures_total`, 369
`prometheus_sd_file_read_errors_total`, 369
`prometheus_sd_http_failures_total`, 150
`prometheus_tsdb_compaction_duration_seconds`, 168
`prometheus_tsdb_compactions_failed_total`, 369
`prometheus_tsdb_wal_corruptions_total`, 369
Prometheus vCloud Director, 144
`prom-http`, 183
`promhttp.Handler`, 82
`prom-https`, 183
PromQL
 введение, 227
 двуместные операторы, 258
 логические операторы, 268
 операторы сравнения, 260
 приоритет операторов, 272
 работа со скалярами, 258
 сопоставление векторов, 262
 математические функции, 275
 `abs`, 275
 `ceil`, 277
 `clamp`, 278
 `clamp_max`, 278
 `clamp_min`, 278
 `exp`, 276
 `floor`, 277
 `ln`, 276
 `log2`, 276
 `log10`, 276
 `round`, 277
 `sgn`, 278
 `sqrт`, 276
 операторы агрегирования, 246
 группировка, 246
 `avg`, 251
 `bottomk`, 254
 `count`, 250
 `count_values`, 255
 `group`, 252
 `max`, 253
 `min`, 253
 `quantile`, 254
 `stddev`, 252
 `stdvar`, 252
 `sum`, 249
 `topk`, 254
 основы агрегирования, 227
 гистограммы, 231
 датчики, 227
 сводные метрики, 230
 счетчики, 229
 правила записи
 антипаттерны, 300
 именование, 302
 как не следует использовать, 300
 когда использовать, 297
 правила для API, 300
 составление функций
 для обработки векторов
 диапазонов, 299
 тригонометрические функции, 278
 `acos`, 278
 `acosh`, 278
 `asin`, 278
 `asinh`, 279
 `atan`, 279
 `atanh`, 279
 `cos`, 279
 `cosh`, 279
 `deg`, 279
 `pi`, 279
 `rad`, 279
 `sin`, 279
 `sinh`, 279
 `tan`, 279
 `tanh`, 279
 функции, 273
 для работы со счетчиками, 286
 `increase`, 287
 `irate`, 288
 `rate`, 286
 `resets`, 289
 изменения датчиков, 288
 `changes`, 289
 `delta`, 291
 `deriv`, 290
 `holt_winters`, 291
 `idelta`, 291
 `predict_linear`, 290
 изменения типа, 273
 `histogram_quantile`, 285
 функции для работы с временем и датой, 279
 `minute`, `hour`, `day_of_week`,
 `day_of_month`, `day_of_year`,
 `days_in_month`, `month` и `year`, 280

time, 279
 timestamp, 281
 функции для работы с метками, 282
 label_join, 283
 label_replace, 282
 HTTP API, 240
 query, 240, 241
 query_range, 240, 243
 promtool, утилита, 94
 proxy_url, 165
 psi=1, параметр ядра, 139
 PSI (Pressure Stall Information – подсистема сбора информации об остановке под давлением), 139
 pushadd_to_gateway, 88
 Pushgateway, неправильное использование, 362
 Pushgateway, 86, 146
 целевые метки, 98
 push_to_gateway, 88
 Python

 клиентские библиотеки в Python 3, 54
 экспортер метрик Consul, 222
 экспортирование в клиентских библиотеках, 77
 Gunicorn , 79
 Twisted, 78
 WSGI, 77
 экспортирование с помощью моста Graphite, 90
 python_info, 107
 python_info, выражение, 55

Q

quantile, 169, 254, 255
 quantile_over_time, 255, 292
 query, конечная точка, 240, 241
 query_range, 279, 297
 query_range, конечная точка, 240, 243

R

rad, функция, 279
 range, функция, 315
 rate, 171
 rate, функция, 41, 45, 57, 64, 66, 140, 229, 230, 232, 286
 использование с селекторами векторов диапазонов, 236

подсчет исключений, 60
 readall\true, 192
 RED (Rate, Errors, Duration – частота, ошибки и продолжительность), метод, 70
 refresh_interval, 150
 region, метка, 320, 327
 relabel_configs, 155
 rename, системный вызов, 142
 repeat_interval, 329
 replace, 282
 replace (действие изменения меток), 158
 reporting_enabled, параметр настройки, 113
 resets, функция, 289
 round, функция, 277
 route, поле, 323
 routes, поле, 323
 rule_files, раздел, 294

S

scalar, функция, 274
 scheme, 166, 176
 scrape_configs, 147
 scrape_interval, 166, 210, 296, 371
 scrape_samples_scraped, 369
 scrape_timeout, 166
 selected_ip_protocol\ipv4, 198
 sensor, команда, 136
 service, метка, 313
 Service, объект Kubernetes, 174
 set, метод, 61
 set_function, метод, 62
 severity, метка, 312, 324
 sgn, функция, 278
 SHOW STATUS, запрос, 190
 SIGHUP, сигнал, 199, 295, 361
 SIGTERM, сигнал, 361
 simpleclient (Java), 83
 sin, функция, 279
 sinh, функция, 279
 smartctl, команда, 140
 SNMP, 194
 SNMP Exporter, экспортёр, 210
 генератор, 210
 и параметр scrape_interval, 210
 sort, функция, 284
 sort_desc, функция, 284

source_labels, 154
 sqrt, функция, 276
 ssh_banner, модуль, 199
 Stackdriver Exporter, экспортер, 210
 start(), функция, 240
 start_http_server, функция, 77
 StateSet (метрики), 95
 statfs, системный вызов, 133
 static_configs, 284
 StatsD Exporter, 351
 StatsD Exporter, экспортер, 210, 212
 stddev, 252
 stddev_over_time, 292
 stdvar, 252
 stdvar_over_time, 292
 sudo, 195
 sum, 249
 использование с датчиками, 228
 without, модификатор, 247
 SummaryMetricFamily, 222
 systemd, 170

T

tan, функция, 279
 tanh, функция, 279
 Targets (Цели), страница, 36
 tcp_connect, модуль, 199
 tcp_connect_tls, 200
 TCP (Transmission Control Protocol – протокол управления передачей), 199
 team, метка, 154, 313
 time, функция, 279
 timeout, параметр (Blackbox Exporter), 208
 Timer, метрика, 63
 timestamp, функция, 281
 TLS
 включение, 345
 дополнительные параметры, 347
 сертификаты, 346
 topk, 254
 Twisted, 78

U

Unix (Node Exporter для получения метрик), 42
 unless, оператор, 270
 UntypedValue, 220

up, 44, 47, 188, 312
 метрика, 38
 user, режим, 133
 USE (Utilization, Saturation, Errors – потребление, насыщение, ошибки), метод, 70

V

vector, функция, 273
 VirtualBox, 173

W

WebDriver Exporter, экспортер, 203
 Windows Exporter, 131
 without, 158
 модификатор, 247
 оператор, 103
 Write Prometheus, плагин (collectd), 209
 write_relabel_configs, 357
 WSGI (Web Server Gateway Interface – интерфейс шлюза веб-сервера), 77

X

X-Prometheus-Scrape-Timeout-Seconds, HTTP-заголовок, 208

Y

YAML, 36, 147
 year, функция, 274

A

Агрегирование, 103
 основы в PromQL, 227
 гистограммы, 231
 датчики, 227
 сводные метрики, 230
 счетчики, 229
 функции, 292
 Альтернативные развертывания, 185
 Анализ
 неструктурированных журналов, 191
 тенденций, 20
 Аннотации и шаблоны, 315
 Аппаратное обеспечение, 359
 Арифметика с плавающей точкой
 64-битный формат в Prometheus, 59

деление на ноль дает NaN, 59
 Арифметические операторы, 259
 возвведение в степень, 259
 вычитание, 259
 деление, 259
 деление по модулю, 259
 сложение, 259
 умножение, 259
 Архитектура Prometheus, 27
 Аутентификация
 включение базовой аутентификации, 348
 в обнаружении служб через HTTP, 150
 клиента (TLS), 165

Б

Базовые единицы, 39
 квантили и процентиili, 65
 секунды как базовая единица времени, 61
 Безопасность на стороне сервера, 345
 включение TLS, 346
 дополнительные параметры TLS, 347
 функции безопасности в Prometheus, 345
 Библиотеки, инструментирование, 71
 Браузер выражений, 31, 36
 использование, 38
 Брэндан Грегг (Brendan Gregg), 139

В

Ввод-вывод дисковый, 42
 Веб-обработчики, уведомители, 332
 Векторы, 233
 диапазонов, 238
 правила записи и использования, 294
 Версии Prometheus, 35
 Вертикальное сегментирование, 352
 Виртуальная машина Java (Java Virtual Machine, JVM), 83
 Внешние метки, 319
 Временные ряды, 100
 количество, или кардинальность, 110
 Выбор
 кода для инструментирования, 70
 целей для мониторинга, 154
 Выключение, 321

оповещений, 321
 Выставление счетов, 33
 Выход на глобальный уровень с помощью механизма федерации, 353

Г

Гарантии стабильности Prometheus, 35
 Гистограммы, 65, 110, 231, 285
 кумулятивные, 66
 сегменты, 66
 Горизонтальное сегментирование, 371
 Группировка, 246
 оповещений в Alertmanager, 322, 326
 отключение, 327
 повторение оповещений, 328
 регулирование частоты оповещений для, 328
 by, модификатор, 248
 without, модификатор, 247
 Группы, 89
 в файле правил, 295, 298

Д

Датчики, 40, 60
 агрегирование, 227
 обратные вызовы, 62
 Consul, 216
 multiprocessing_mode, конфигурационный параметр, 80
 Дашборды, 31, 112, 351
 и Promdash и шаблоны консоли, 113
 новый дашборд Grafana, 116
 стена графиков, 117
 ускорение работы, 297
 Двойное экспоненциальное слаживание Холта–Винтерса (Holt-Winters), 291
 Двухместные операторы, 258
 операторы сравнения, 260
 приоритет операторов, 272
 работа со скалярами, 258
 арифметические операторы, 259
 сопоставление векторов, 262
 логические операторы, 268
 многие к одному, 265
 многие ко многим, 268
 один к одному, 263
 Дерево маршрутизации, 322

настройка для Alertmanager, 323
Дерево маршрутов
 визуальный редактор, 326
настройка в Alertmanager,
 группировка, 326
Дисковый ввод-вывод, 42, 131
Долгосрочная поддержка (Long Term Support, LTS), 35
Долгосрочное хранение, 32, 356
Домашняя панель управления (Grafana), 113
Дочерние маршруты, 323
Дочерние метрики, 101, 103
Другие системы мониторинга, 209
Дубликаты заданий, 166

Ж

Журналирование, 24
Журналы
 запросов, 25
отладки, 25
отладки Prometheus, 47
приложений, 25
транзакций, 25

З

Задания cron, 140
Задержки
 вычисление средней задержки, 68
отслеживание в программе Hello World (пример), 63
SLA и квантили, 67
Запуск Prometheus, 34
 браузер выражений, 36
конфигурация, 35
требования, 34
Запуск Prometheus за обратным прокси, 362

И

Идемпотентность, 71
 пакетных заданий, 71
Извлечение данных, 30
 по запросу и активная передача, 30
Изменение
 меток, 154
|, оператор чередования, 155

изменение регистра значений, 163
регулярные выражения, 154
метрик, 167
регистра значений меток, 163
Именование меток, 100
Импорт, 56
Инструментирование, 27, 54, 76, 351
 библиотек, 71
 гистограммы, 65
 кумулятивные, 66
 как широко использовать
 инструментирование, 72
 пакетных заданий, 71
 пример на Go, 82
 пример программы на Python,
 экспортирующей метрики, 54
 программ на языках JVM, 83
 пулов рабочих потоков, 72
 сводные метрики, 63
 служб, 70
 счетчики, 56
 подсчет исключений, 58
 подсчет размера, 59
 что инструментировать, 70
Инструментированные метки, 98
несколько меток на метрику, 101
Интервалы времени в Prometheus и PromQL, 238
Интерфейс шлюза веб-сервера (Web Server Gateway Interface, WSGI), 77
Информационные метрики, 106
Исключение таблицы, 109
Исключений, подсчет, 58
Источники данных, 115

К

Как широко использовать
инструментирование, 72
Кардинальность, 26, 110
 проблемы
с производительностью, 352
уменьшение с помощью правил
записи, 297
Квантили, 65, 231
 вычисления в гистограммах, 231
и задержки в SLA, 67
и процентили, 65
ограничения, 67

- Кеши, 24
 метрики для общего числа обращений к кешу и для промахов кеша, 71
 отслеживание размеров или количеств элементов, 62
 потребление памяти, 60
- Кластеризация диспетчеров уведомлений, 366
- Кластерные системы хранения, 358
- Клиентские библиотеки, 27, 54, 77, 219
 важность высокой производительности, 81
 официальные и неофициальные, 28
 регистрация метрик, 57
- Ключи группировки, 89
- Контейнеры Kubernetes, 170
- Конфигурационные файлы, формат YAML, 36
- Конфигурация
 перезагрузка в Prometheus, 295
 проверка с помощью `promtool check config`, 295
- Конфликты меток и `honor_labels`, 169
- Критические изменения меток, 108
- Кумулятивные гистограммы, 66
- Л**
- Логические операторы, 268
- М**
- Маркер устаревания, 236
 Маршрутизация, 322
 Маршрут по умолчанию, 323
 Масштабирование Prometheus, 352, 354
 Математические функции, 275
`abs`, 275
`ceil`, 277
`clamp`, 278
`clamp_max`, 278
`clamp_min`, 278
`exp`, 276
`floor`, 277
`ln`, 276
`log2`, 276
`log10`, 276
`round`, 277
`sgn`, 278
- `sqrt`, 276
 Машины роли, 106
 Мгновенные векторы, 241
`{}`, 258
 применение двухместных операторов, 258
- Метамониторинг, 367
- Метки, 97, 352
 агрегирование, 103, 229
 введение, 97
 внешние, 319
 допустимые символы, 154
 изменение, 154
`|`, оператор чередования, 155
 регулярные выражения, 154
- изменение регистра значений, 163
- именование, 100
 инструментированные, 98
 ключи группировки для отправляемых метрик
`Pushgateway`, 89
 когда использовать, 108
 конфликты и `honor_labels`, 169
 критические изменения, 108
 оповещений, 312
 определение в поле `labels` в файле правил, 296
 отражающие структуру приложения, 157
 целевые, 98, 144
 целей, 157
 шаблоны использования, 104
 информационные метрики, 106
 перечисления, 104
`Graphite`, мост, 90
- Метки заданий, 89
- Метрики, 26
 автоматическая регистрация клиентскими библиотеками, 57
 выбор имен, 72
 библиотеки, 74
 единицы измерения в именах, 73
 змеиная нотация, 73
 символы, 73
 суффиксы, 73
 гистограммы, 65
 кумулятивные, 66
 датчики, 60
 из клиентских библиотек для Java, 83

- изменение, 167
 информационные, 106
 как извлекать, 165
 машинного уровня из других систем мониторинга, 350
 модульное тестирование, 68
 определение, 57
 поставляемые экспортером узла, 42
 пример программы на Python, экспортирующей метрики, 54
 проблемы с производительностью из-за высокой кардинальности, 352
 разбивка HTTP-запросов по путям, 97
 сводные, 63
 суффиксы в именах, 62
 счетчики, 56
 подсчет исключений, 58
 подсчет размера, 59
 экспортование в Prometheus, 76
 Consul, 215
 Многозадачность с Gunicorn, 79
 Модульное тестирование метрик, 68
 Мониторинг, выбор целей, 154
 Мосты, 90
- Н**
- Надежность, 363
 Настраиваемые сборщики, 29
 Натуральный логарифм, 276
 Негативные опережающие проверки, 234
 Несколько меток на метрику, 101
 Неудачные попытки извлечь данные, 47
- О**
- Обнаружение проблем, 369
 Обнаружение служб, 29, 144
 механизмы, 146
 на основе файла, 147
 нисходящие и восходящие, 146
 с помощью EC2, 152
 с помощью Consul, 151
 статические, 146
 через HTTP, 150
 на основе файла, 147
 с помощью EC2, 152
 с помощью Consul, 151
- через HTTP, 150
 Kubernetes, 175
 endpointslice, 178
 ingress, 184
 node, 175
 pod, 183
 service, 178
 Обратные вызовы, 62
 Обратный прокси, 362
 Ожидающие оповещения, 309
 ОЗУ, 359
 Операторы агрегирования, 246
 группировка, 246
 avg, 251
 bottomk, 254
 count, 250
 count_values, 255
 group, 252
 max, 253
 min, 253
 quantile, 254
 stddev, 252
 stdvar, 252
 sum, 249
 topk, 254
 Операторы сравнения, 260
 больше, 260
 больше или равно, 260
 меньше, 260
 меньше или равно, 260
 не равно, 260
 равно, 260
 фильтрация, 260
 bool, модификатор, 261
 Оповещения, 20, 47, 307
 аннотации и шаблоны, 315
 архитектура Prometheus и Alertmanager, 307
 должны иметь хозяина, 314
 метки, 312
 правила оповещения, 48
 хорошие оповещения, 317
 Отладка, 20
 Отсутствующие серии, absent и absent_over_time, 283
- П**
- Пакетные задания, 70, 86
 инструментирование, 71

- Память**
- график потребления памяти в Grafana, 118
 - потребление сервером Prometheus и экспортером Node Exporter, 45
 - результат process_resident_memory_bytes, 40
- Панель**
- временной шкалы состояний (Grafana), 125
 - временных рядов, 118
 - статистики (Grafana), 121
 - таблицы (Grafana), 123
- Парсеры, 91
- Перекрестный мониторинг, 367
- Переменные шаблона (Grafana), 126
- Перечисления, 104
- Планирование развертывания, 350
- Планирование сбоев, 363
- Повторение
- оповещений, 328
 - уведомлений, 322
- Подавление, 321
- уведомлений, 340
- Подзапросы, 238
- Подсистема сбора информации об остановке под давлением (Pressure Stall Information, PSI), 139
- Подстановочные знаки, в именах файлов, 148
- Подсчет
- исключений, 58
 - размера, 59
- Поиск дорогостоящих метрик и целей, 369
- Получение помощи, 374
- Пользовательские реестры, 57, 80
- Пользовательские сборщики, 105
- Порты экспортёров по умолчанию, 190
- Правила записи, 31, 294
- антипаттерны, 300
 - именование, 302
 - использование, 294
 - как не следует использовать, 300
 - когда использовать, 297
 - правила для API, 300
 - составление функций для обработки векторов диапазонов, 299
- Правила оповещения, 31, 48, 308
- for, поле, 310
- Приемники, 322
- настройка, 330
 - backend-ticket, 325
- Продолжительность уплотнения, базы данных временных рядов, 232
- Производительность, важность для клиентских библиотек, 81
- Пропускная способность сети, 359
- Протокол
- передачи гипертекста (HyperText Transfer Protocol, HTTP), 201
 - управления передачей (Transmission Control Protocol, TCP), 199
 - управляющих сообщений интернета (Internet Control Message Protocol, ICMP), 194
- Профилирование, 23
- Процентили, 65
- и квантили, 65
- Процессы, многозадачность с Gunicorn, 79
- Пулы рабочих потоков, инструментирование, 72
- P**
- Развертывание Prometheus, 350
- долгосрочное хранение, 356
 - планирование развертывания, 350
 - планирование сбоев, 363
 - получение помощи, 374
 - расширение Prometheus, 352
 - управление изменениями, 373
 - управление
 - производительностью, 368
 - горизонтальное
 - сегментирование, 371
 - обнаружение проблем, 369
 - поиск дорогостоящих метрик и целей, 369
 - снижение нагрузки, 370
- Федерация, 353
- Эксплуатация Prometheus, 358
- аппаратное обеспечение, 359
 - сети и аутентификация, 362
 - управление конфигурацией, 360
- Размер**
- дискового сектора, 135
 - подсчет, 59

Расширение Prometheus, 352
 Регрессия методом наименьших квадратов, 290
 Регулирование частоты оповещений, 328
 уведомлений, 322
 Регулярные выражения, 156
 сопоставители, 324
 Реестр по умолчанию, 76
 Реестры, 57, 76
 Режим агента, 358
 Резервный маршрут, 323

C

Самоподписанные сертификаты, 346
 Сбои, планирование, 363
 Сборщики, 132
 дисковой статистики, 134
 информации
 об аппаратной части, 136
 об имени узла, 138
 об операционной системе, 138
 о давлении, 139
 о потреблении памяти, 136
 о процессоре, 132
 о сетевых устройствах, 135
 о средней нагрузке, 139
 о файловой системе, 133
 пользовательские, 105, 219
 метки для метрик, 223
 статистики, 137
 текстовых файлов, 140
 использование, 141
 отметки времени, 143
 Сводные метрики, 63
 агрегирование, 230
 Сегменты (в гистограммах), 66
 кумулятивные гистограммы, 66
 Селекторы, 233
 векторов диапазонов, 236
 мгновенных векторов, 233, 235
 подзапросы, 238
 Семейство метрик, 100
 Сервлеты, 85
 Сертификаты, 176
 самоподписанные, 346
 TLS, 346
 Сети и аутентификация, 362

Системы
 автономного обслуживания, 70
 мониторинга, другие, 209
 онлайн-обслуживания, 70
 управления конфигурациями, 144
 Скаляры, арифметические
 операторы, 258
 Службы, инструментирование, 70
 Смарт-хост SMTP, 50
 Смещение, 239
 Снижение нагрузки, 370
 События, счетчики, 40
 Сопоставители, 234, 324
 неравенства, 234
 несоответствия регулярному
 выражению, 234
 ошибка в маршруте по
 умолчанию, 324
 равенства, 234
 соответствия регулярному
 выражению, 234
 Сопоставитель меток, 45
 Сопоставление векторов, 262
 логические операторы, 268
 многие к одному, 265
 многие ко многим, 268
 один к одному, 263
 Сортировка с помощью sort
 и sort_desc, 284
 Списки, 164
 генераторы, 66
 Средние значения
 вычисление для сводных метрик, 230
 вычисление средних размеров
 событий с использованием
 гистограмм, 232
 Стандартная дисперсия, 253
 Стандартное отклонение, 252
 Статическое обнаружение служб, 146
 Стена графиков, 117
 Суперагент, 131
 Суффиксы метрик, 62
 Счетчики, 40, 56, 110, 285
 агрегирование, 229
 деление на сумму, 230
 и многозадачный режим, 81
 модульное тестирование, 68
 обработка в пользовательских
 сборщиках, 221

подсчет исключений, 58
 подсчет размера, 59
 текстовый формат
 экспортования, 92
 увеличение на отрицательное число
 считается ошибкой, 60
Consul, 216
`increase`, функция, 287
`irate`, функция, 288
`rate`, функция, 286
`resets`, функция, 289

T

Таймеры, 216
 Текстовый формат (*Prometheus*), 91
 парсеры, 91
 Теорема выборки
 Найквиста–Шеннона, 121
 Трассировка, 24
 Тригонометрические функции, 278
 `acos`, 278
 `acosh`, 278
 `asin`, 278
 `asinh`, 279
 `atan`, 279
 `atanh`, 279
 `cos`, 279
 `cosh`, 279
 `deg`, 279
 `pi`, 279
 `rad`, 279
 `sin`, 279
 `sinh`, 279
 `tan`, 279
 `tanh`, 279
 Тригонометрический оператор, 260
 `atan2`, 260

У

Уведомители, 330
 Уведомления, 307, 322
 конвейер уведомлений
 в *Alertmanager*, 321
 отправка только в определенное
 время, 309
 подавление, 340
 устраненные, 338
 шаблоны, 316, 333

Удаленная запись, 357
 Удаленная конечная точка записи, 357
 Удаленное чтение, 357
 Управление
 изменениями, 373
 конфигурацией, 360
 производительностью, 368
 горизонтальное
 сегментирование, 371
 обнаружение проблем, 369
 поиск дорогостоящих метрик
 и целей, 369
 снижение нагрузки, 370
 уведомлениями, 32
 Установка *Grafana*, 113
 Устаревание, 235
 Устраненные оповещения, 309
 Устраненные уведомления, 338

Ф

Файловые дескрипторы, 360
 Файловые системы, 360
 `node_filesystem_size_bytes`,
 метрика, 227
 Файлы правил, 294, 361
 Федерация, выход на глобальный
 уровень, 353
 Фильтрация, 260
 в правилах оповещения, 308
 Форматы экспортования
 парсеры в клиентских библиотеках
 Python, 91
 текстовый формат *OpenMetrics*, 94
 метки, 96
 отметки времени, 96
 типы метрик, 95
 текстовый формат *Prometheus*, 91
 метки, 92
 отметки времени, 93
 проверка метрик, 94
 типы метрик, 92
 экранирование, 93
 Функции, 273
 агрегирования по времени, 292
 для работы с временем и датой, 279
 `minute`, `hour`, `day_of_week`,
 `day_of_month`, `day_of_year`,
 `days_in_month`, `month` и `year`, 280
 `time`, 279

- timestamp, 281
 - для работы с метками, 282
 - label_join, 283
 - label_replace, 282
 - для работы со счетчиками, 286
 - increase, 287
 - irate, 288
 - rate, 286
 - resets, 289
 - изменения датчиков, 288
 - changes, 289
 - delta, 291
 - deriv, 290
 - holt_winters, 291
 - idelta, 291
 - predict_linear, 290
 - изменения типа
 - scalar, 274
 - vector, 273
 - математические, 275
 - тригонометрические, 278
- X**
- Хранение долгосрочное, 356
 - Хранилище, 30
- Ц**
- Целевые метки, 98, 144, 157, 247
 - Центры сертификации, самоподписанные сертификаты, 346
- Ш**
- Шаблоны
 - использования меток, 104
 - информационные метрики, 106
 - перечисления, 104
 - консоли, 113
 - уведомлений, 316, 333
 - Go, 315
 - Шторм уведомлений, 32
- Э**
- Эксплуатация Prometheus, 358
 - аппаратное обеспечение, 359
 - сети и аутентификация, 362
 - управление конфигурацией, 360
 - Экспортер узла, 42
 - Экспортеры, 28, 187
 - безопасность на стороне сервера, 345
 - в развертывании Prometheus, 350
 - и реестр по умолчанию, 222
 - порты по умолчанию, 190
 - разработка, 215
 - методические рекомендации, 224
 - пользовательские сборщики, 219
 - Consul Telemetry, 215
 - экспортер метрик Consul на Python, 222
 - Blackbox Exporter, 194
 - CloudWatch Exporter, 210
 - Collectd Exporter, 209
 - Consul Exporter, 187
 - Graphite Exporter, 210
 - Grok Exporter, 191
 - InfluxDB Exporter, 209, 211
 - JMX Exporter, 210
 - MySQLD, 189
 - New Relic Exporter, 210
 - NRPE Exporter, 211
 - Pingdom Exporter, 210
 - SNMP Exporter, 210
 - Stackdriver Exporter, 210
 - StatsD Exporter, 210, 212
 - WebDriver Exporter, 203
 - Экспортирование, 76
 - из библиотек на Python, 77
 - из клиентских библиотек
 - на Go, 82
 - парсеры, 91
 - из клиентских библиотек на Java, 83
 - из пакетных заданий, Pushgateway, 86
 - поддержка в Prometheus, 77
 - с использованием мостов, 90
 - Элементы управления временем (Grafana), 119
 - Эффект искажения (Grafana), 121
 - Эхо-запросы, 195
 - Эхо-ответы, 195

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38, оф. 10;
тел.: (499) 782-38-89, электронная почта: books@aliants-kniga.ru.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: <http://www.galaktika-dmk.com/>.

Жюльен Пивотто и Брайан Бразил

Запускаем Prometheus

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Зам. главного редактора *Сенченкова Е. А.*
Перевод *Киселев А. Н.*
Корректор *Абросимова Л. А.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура РТ Serif. Печать цифровая.
Усл. печ. л. 31,85. Тираж 100 экз.

Веб-сайт издательства: www.dmkpress.com

Возьмите на вооружение Prometheus – систему мониторинга на основе метрик, используемую тысячами организаций. Это обновленное издание расскажет, что такое Prometheus, и познакомит с наиболее важными аспектами этой системы. Вы узнаете, как настроить Prometheus, Node Exporter и Alertmanager, научитесь использовать эти инструменты в своей работе и поймете, почему эта система с открытым исходным кодом продолжает набирать популярность.

Прочитав эту книгу, вы:

- познакомитесь с приемами мониторинга инфраструктуры с помощью Node Exporter и сбора данных о работе сети, дисков и параметрах нагрузки;
- узнаете, как и когда инструментировать код приложений;
- получите общее представление о Grafana, популярном инструменте для создания дашбордов;
- увидите, как получать доступ для мониторинга к своим машинам и службам с помощью механизмов обнаружения служб, включая новый механизм HTTP SD;
- научитесь использовать Prometheus в Kubernetes и познакомитесь с экспортёрами, которые можно использовать с контейнерами;
- познакомитесь с расширенными возможностями Prometheus, включая тригонометрические функции;
- узнаете, как Prometheus обеспечивает безопасность с помощью различных средств, включая TLS и базовую аутентификацию.

Издание предназначено инженерам по поддержке инфраструктуры, администраторам Kubernetes (и не только), а также разработчикам приложений.

«Авторы – два ведущих разработчика Prometheus, и книга отражает их глубокое знание этой системы. В ней вы найдете бесценные практические советы по развертыванию и эксплуатации Prometheus в реальных ситуациях».

Юлиус Вольц, сооснователь Prometheus и основатель PromLabs

«Эта книга, представляющая опыт ведущих разработчиков по использования Prometheus, поможет вам уверенно контролировать свои службы».

Тиджей Хоплок,
старший инженер
по надежности сайта, NS1

Жюльен Пивотто – ведущий разработчик сервера Prometheus и экосистемы CNCF с 2017 г. В настоящее время работает главным архитектором программного обеспечения в компании Ollu и одновременно является ее соучредителем.

Брайан Бразил – основатель компании Robust Perception и разработчик ядра Prometheus. Хорошо известен в сообществе Prometheus, провел бесчисленное количество презентаций на конференциях и освещает многие аспекты Prometheus и мониторинга в своем блоге.

ISBN 978-6-01810-341-4



9 786018 103414 >