**Лучшие практики в области производительности**

Руководство пользователя по общим и специфическим для каждого языка рекомендациям по повышению производительности.

Общая информация

Всегда **по возможности повторно используйте заглушки и каналы**.

**Используйте контрольные запросы** для поддержания активности HTTP/2-соединений в периоды бездействия, чтобы можно было быстро и без задержек выполнять первоначальные RPC-запросы (например, аргумент канала C++ GRPC\_ARG\_KEEPALIVE\_TIME\_MS).

**Используйте потоковые RPC-вызовы** при обработке длительного логического потока данных от клиента к серверу, от сервера к клиенту или в обоих направлениях. Потоки позволяют избежать непрерывного инициирования RPC-вызовов, что включает балансировку нагрузки на стороне клиента, запуск нового HTTP/2-запроса на транспортном уровне и вызов пользовательского обработчика методов на стороне сервера.

Однако после запуска потоков их нельзя сбалансировать по нагрузке, и их сложно отлаживать в случае сбоев. Они также могут повысить производительность в небольших масштабах, но могут снизить масштабируемость из-за балансировки нагрузки и сложности, поэтому их следует использовать только в том случае, если они обеспечивают существенную производительность или упрощают логику приложения. Используйте потоки для оптимизации приложения, а не gRPC.

***Примечание:*** *это не относится к Python (подробности см. в разделе о Python).*

*(Специальная тема)* Каждый канал gRPC использует 0 или более подключений HTTP/2, и каждое подключение обычно имеет ограничение на количество одновременных потоков. Когда количество активных RPC-запросов в соединении достигает этого предела, дополнительные RPC-запросы ставятся в очередь на стороне клиента и должны дождаться завершения активных RPC-запросов, прежде чем будут отправлены. Из-за этой очереди в приложениях с высокой нагрузкой или длительными потоковыми RPC-запросами могут возникать проблемы с производительностью. Есть два возможных решения:

**Создайте отдельный канал для каждой области с высокой нагрузкой** в приложении.

**Используйте пул каналов gRPC** для распределения RPC-запросов по нескольким соединениям (каналы должны иметь разные аргументы, чтобы предотвратить повторное использование, поэтому определите аргумент канала для конкретного использования, например номер канала).

***Примечание:*** *команда gRPC планирует добавить функцию для устранения этих проблем с производительностью (см. grpc/grpc#21386 для получения дополнительной информации), поэтому любое решение, предполагающее создание нескольких каналов, является временным обходным решением, которое в конечном итоге не потребуется.*

C++

**Не используйте Sync API на серверах, чувствительных к производительности.** Если производительность и/или потребление ресурсов не являются проблемой, используйте Sync API, так как его проще всего реализовать для сервисов с низкой скоростью обработки запросов.

**Предпочитайте API обратного вызова другим API для большинства RPC-запросов**, учитывая, что приложение может избежать всех блокирующих операций или перенести их в отдельный поток. API обратного вызова проще в использовании, чем асинхронный API с очередью завершения, но в настоящее время он медленнее для рабочих нагрузок с действительно высокой скоростью передачи данных.

Если вам нужно использовать API асинхронных очередей завершения, **лучшим компромиссом с точки зрения масштабируемости будет наличие numcpuпотоков.** Идеальное количество очередей завершения в зависимости от количества потоков может меняться со временем (по мере развития gRPC C++), но в версии gRPC 1.41 (сентябрь 2021 г.) использование 2 потоков на каждую очередь завершения обеспечивает наилучшую производительность.

Для API асинхронной очереди завершения убедитесь, что **зарегистрировано достаточно серверных запросов для желаемого уровня параллелизма**, чтобы сервер не застревал на медленном этапе, который приводит к последовательной обработке запросов.

*gRPC::GenericStub* (специальная тема) может быть полезен в некоторых случаях, когда на сериализацию протокола тратится много ресурсов / процессорного времени. Этот класс позволяет приложению напрямую отправлять **необработанный gRPC::ByteBuffer в качестве данных**, а не сериализировать его из какого-либо протокола. Это также может быть полезно, если одни и те же данные отправляются несколько раз: сначала выполняется явная сериализация протокола в ByteBuffer, а затем несколько раз отправляется ByteBuffer.

Java

**Используйте неблокирующие заглушки** для распараллеливания RPC.

**Предоставьте пользовательский исполнитель, который ограничивает количество потоков в зависимости от вашей рабочей нагрузки** (кэшированный (по умолчанию), фиксированный, forkjoin и т. д.).

Питон

Потоковые RPC создают дополнительные потоки для получения и, возможно, отправки сообщений, что делает **потоковые RPC намного медленнее, чем унарные RPC** в gRPC Python, в отличие от других языков, поддерживаемых gRPC.

**Использование asyncio** могло бы повысить производительность.

Использование future API в стеке синхронизации приводит к созданию дополнительного потока. **По возможности избегайте использования future API**.

*(Экспериментальная версия)* Экспериментальная **однопоточная реализация унарного потока** доступна через опцию канала SingleThreadedUnaryStream, которая может сократить задержку на 7% для каждого сообщения.

Последнее изменение: 12 ноября 2024 г.: встраивание видео с YouTube на разные веб-страницы (#1380) (196f408)