Программирование: как кэширование в Redis ускоряет проект и когда оно вредит

13.11.2026 · PDF-версия статьи с сайта cms.dolinais.ru

Зачем кэш, если и так работает

Кэширование — это сохранение результатов дорогих операций в быстром хранилище, чтобы не выполнять их повторно. Redis — популярное in-memory хранилище структур данных, которое часто используют как кэш. Типичный сценарий: вы добавили Redis, замерили p95 — стало быстрее. Прошла неделя, карточка товара снова грузится секундами, а база получает в разы больше запросов, чем до кэширования. Проблема не в Redis, а в том, как вы его используете. В этой статье разберём типичные антипаттерны cache-aside (подход, при котором приложение само читает из кэша и при промахе идёт в базу) и покажем, как их диагностировать.

Антипаттерн 1: горячий ключ

Горячий ключ — это ключ, который запрашивается непропорционально часто. Например, данные главной страницы или популярного товара. Если такой ключ хранится на одном шарде Redis Cluster, нагрузка на этот шард растёт, и задержки увеличиваются для всех запросов, попадающих на него.

Как обнаружить: команда INFO keyspace покажет количество ключей, но не частоту обращений. Используйте MONITOR осторожно (только в тестовой среде) или включите логирование медленных запросов через SLOWLOG. Также помогает анализ со стороны приложения: счётчики попаданий и промахов по каждому ключу.

Что делать: разбить горячий ключ на несколько копий с суффиксом (например, product:123:1, product:123:2) и выбирать копию случайно. Или использовать локальный кэш в приложении (например, Caffeine для Java или аналоги) с коротким TTL.

Антипаттерн 2: гонка при инвалидации

Инвалидация — удаление или обновление данных в кэше при изменении источника истины. Классическая проблема: два процесса одновременно обновляют ключ. Первый прочитал старое значение из базы и собирается записать его в кэш, второй в это время уже обновил базу и удалил ключ. В результате в кэше оказывается устаревшее значение, которое может жить долго.

Как обнаружить: сравните данные в кэше и в базе для одних и тех же сущностей. Если расхождения повторяются, скорее всего, есть гонка. Также поможет логирование: при каждом обновлении базы фиксируйте, был ли удалён ключ.

Что делать: использовать паттерн «обновление через блокировку» (например, распределённая блокировка на Redis) или применять версионирование значений. Другой вариант — стратегия write-through (запись сначала в кэш, потом в базу) с осторожностью. И помните о TTL: даже если гонка произошла, короткий TTL ограничит время жизни плохих данных.

Антипаттерн 3: отсутствие TTL или слишком большой TTL

TTL (time to live) — время жизни ключа. Если TTL нет, ключ живёт, пока его не удалят или не вытеснят по политике maxmemory. Это приводит к устареванию данных и переполнению памяти. Слишком большой TTL даёт тот же эффект.

Как обнаружить: команда INFO stats покажет evicted_keys — количество вытесненных ключей. Если оно растёт, памяти не хватает. Также проверьте средний TTL по ключам (например, через сканирование в тестовой среде).

Что делать: задавать TTL для каждого ключа исходя из допустимой задержки устаревания. Для справочников — часы, для динамических данных — минуты или секунды. Использовать политику вытеснения allkeys-lru или volatile-lru в зависимости от того, все ли ключи имеют TTL.

Антипаттерн 4: кэширование ошибок и пустых значений

Если при промахе вы идёте в базу и получаете ошибку, а затем кэшируете эту ошибку как значение, все последующие запросы будут получать ошибку, пока не истечёт TTL. Аналогично с пустыми результатами: если кэшировать «не найдено» на долго, можно пропустить появление данных.

Как обнаружить: поищите в коде места, где результат запроса к базе безусловно сохраняется в кэш. Проверьте, обрабатываются ли исключения.

Что делать: кэшировать только успешные ответы. Для пустых значений использовать короткий TTL (например, 30 секунд) или отдельный ключ-маркер. Ошибки не кэшировать вовсе.

Как диагностировать проблемы с помощью Redis

Встроенные команды дают много информации. SLOWLOG GET покажет последние медленные команды — это помогает найти тяжёлые операции (например, KEYS * или большие HGETALL). INFO предоставляет секции: memory (использование памяти), stats (попадания, промахи, вытеснения), clients (подключения), keyspace (количество ключей и TTL). Регулярно снимайте эти метрики и настраивайте алерты на рост evicted_keys или падение keyspace_hits.

> Совет: не используйте MONITOR в продакшене — он снижает производительность и может раскрыть конфиденциальные данные в логах.

Чек-лист внедрения кэширования

1. Определите, какие данные действительно стоит кэшировать: частые чтения, редкие изменения.
2. Выберите стратегию: cache-aside, write-through или write-behind. Для большинства случаев подходит cache-aside.
3. Установите TTL для каждого ключа. Не оставляйте ключи без TTL, если не уверены.
4. Настройте политику вытеснения и лимит памяти в Redis (maxmemory).
5. Обрабатывайте промахи и ошибки: не кэшируйте ошибки, для пустых значений используйте короткий TTL.

6. Внедрите метрики: попадания/промахи, задержки, количество вытеснений. Настройте алерты.
7. Проведите нагрузочное тестирование с реалистичным профилем запросов, включая горячие ключи.

Кэширование — мощный инструмент, но требует дисциплины. Начните с малого: закэшируйте один тип данных, измерьте эффект и постепенно расширяйте. И помните: кэш не заменяет оптимизацию запросов к базе, а дополняет её.

Источник: https://cms.dolinais.ru/programmirovanie-kak-keshirovanie-v-redis-uskoryaet-proekt-i-kogda-ono-vredit