Паттерн Circuit Breaker (Предохранитель)

В распределенных системах сбой одного сервиса может вызвать эффект домино, положив всё приложение. Паттерн **Circuit Breaker** создан для предотвращения таких **каскадных сбоев** (Cascading Failures).

Он работает как прокси-прослойка между вашим сервисом и внешней зависимостью (другим микросервисом, базой данных или API).

- **Принцип действия:** Circuit Breaker мониторит статистику ответов. Если частота ошибок (Error Rate) превышает заданный порог, он **«размыкает цепь»**.
- **Результат:** Дальнейшие запросы к проблемному сервису блокируются мгновенно, без ожидания таймаута. Приложение сразу получает ошибку или заглушку (Fallback).

Аналогия из реального мира Название заимствовано из электротехники. Электрический автомат в щитке отключает питание, если в сети возникает перегрузка, чтобы предотвратить пожар. Программный Circuit Breaker делает то же самое: он «обесточивает» неисправную зависимость, чтобы спасти всю систему от истощения ресурсов (потоков, памяти, соединений).

Проблема: Эффект домино (Cascading Failures)

В микросервисной архитектуре сбои заразны. Если один компонент начинает тормозить, он может утащить за собой всю систему. Circuit Breaker предотвращает этот сценарий, решая три фундаментальные проблемы:

1. Исчерпание ресурсов (Resource Exhaustion)

Это самый коварный сценарий. Сервис, который **медленно отвечает**, опаснее сервиса, который **упал мгновенно**. Когда зависимость «тупит», вызывающий сервис держит соединение открытым, ожидая ответа. Это блокирует потоки (threads) и соединения в пуле (connection pool). В итоге у вызывающего сервиса заканчиваются ресурсы, и он перестает отвечать даже на те запросы, которые не связаны с проблемной зависимостью.

- **Пример:** Сервис заказов вызывает Платежный API. Из-за проблем с базой данных API отвечает 60 секунд вместо 200 мс. Сервис заказов быстро исчерпывает свой пул потоков, ожидая ответов. В результате пользователи не могут даже просто посмотреть список своих старых заказов (операция `read-only`), хотя платежи им сейчас не нужны.
- **Решение:** Circuit Breaker обнаруживает тайм-ауты и размыкает цепь. Заказы мгновенно получают ошибку или ставятся в очередь, а потоки освобождаются для обслуживания других запросов.

2. Защита критического пути (Partial Failure)

Сбой второстепенной функции не должен блокировать основной бизнес-процесс.

- **Пример:** Страница товара подгружает данные из трех сервисов: «Товары», «Цены» и «Отзывы». Если сервис «Отзывы» упал, а магазин ждет успешного ответа от всех трех, страница не загрузится вовсе. Вы теряете продажи из-за недоступности комментариев.
- **Решение:** Circuit Breaker изолирует сбой. Если «Отзывы» не отвечают, магазин просто рендерит страницу без них (возвращает пустой список или кэшированные данные), сохраняя кнопку «Купить» активной. Это называется **Graceful Degradation** (Плавная деградация).

3. Предотвращение восстановления (Prevention of Recovery)

Когда упавший сервис пытается подняться, его часто «добивают» клиенты. Если база данных перегружена, а 10 микросервисов продолжают долбить её ретраями (повторными запросами), она никогда не восстановится. Это превращает временный лаг в длительный простой.

- **Пример:** База данных начинает тормозить под нагрузкой. Клиенты, не получая ответа, начинают агрессивно слать повторные запросы, увеличивая нагрузку в разы.
- **Решение:** Circuit Breaker во всех клиентах одновременно переходит в состояние **Open** (Разомкнут). Нагрузка на базу падает до нуля. Это дает администраторам БД драгоценное время, чтобы разгрести очередь запросов или перезагрузить сервер.

Почему try-catch недостаточно

Классическая обработка ошибок (блок `try-catch`) отлично работает в монолитах: операция либо выполняется, либо мгновенно выбрасывает исключение. Однако в распределенных системах мы сталкиваемся с третьим, самым опасным состоянием — **зависанием** (Latent Failure).

Когда внешний сервис просто «молчит», ваш код продолжает ждать ответа, потребляя память и потоки. Это не приносит ни успеха, ни ошибки — только истощение ресурсов. Главная задача Circuit Breaker — превратить эти медленные, мучительные зависания в **быстрые сбои (Fail Fast)**. Лучше мгновенно вернуть ошибку пользователю, чем заставлять его ждать 30 секунд перед белым экраном.

Почему это критично в облаке:

1. **Ловушка Автомасштабирования (Autoscaling):** Если ваши серверы зависли в ожидании ответа от базы данных, Autoscaler увидит рост потребления ресурсов и добавит новые инстансы. Но новые серверы точно так же зависнут. Вы просто увеличите счет за облако, не решив проблему.
2. **Слепота Балансировщика (Load Balancer):** Балансировщики обычно проверяют доступность сервера через простой `Health Check` (ping). Зависший сервис может отвечать на пинг «Я жив», но при этом не обрабатывать реальные транзакции. Балансировщик продолжит слать трафик на «зомби-сервис».
3. **Иллюзия нормальности (Monitoring Delays):** Дашборды могут показывать «зеленый свет» (сервис работает, CPU в норме), потому что потоки просто спят в ожидании ввода-вывода (IO Wait). Реальная проблема обнаружится только когда начнут приходить жалобы от пользователей.

Анатомия Circuit Breaker: Машина состояний

Логика работы Circuit Breaker описывается тремя состояниями. Важно запомнить контринтуитивную терминологию: **Closed (Закрыт)** означает, что система работает нормально (ток идет), а **Open (Разомкнут)** — защита активирована (ток прерван).

1. Closed (Закрыт — Нормальная работа)

- Запросы беспрепятственно проходят к внешнему сервису.
- Система фоном считает статистику: количество успешных и неуспешных вызовов.
- Если уровень ошибок (Error Rate) превышает порог (например, 50% за последние 10 секунд), автомат переходит в состояние **Open**.

2. Open (Разомкнут — Сбой/Защита)

- Цепь разорвана. Все попытки вызвать внешний сервис **мгновенно** блокируются без отправки запроса по сети.
- Клиент получает ошибку или заглушку (Fallback).
- Это состояние длится заданное время (например, 60 секунд), давая упавшему сервису «остыть» и восстановиться. По истечении таймера система переходит в **Half-Open**.

3. Half-Open (Полуоткрыт — Проверка)

- Система пропускает ограниченное количество тестовых запросов («канарейку») к внешнему сервису.
- **Успех:** Если запросы прошли успешно, сервис считается восстановленным. Цепь **замыкается (Closed)**, счетчики сбрасываются.
- **Провал:** Если ошибка повторяется, цепь снова **размыкается (Open)**, и таймер ожидания запускается заново.

Принципы проектирования

Чтобы Circuit Breaker приносил пользу, а не головную боль, нужно правильно настроить три компонента:

1. Детекция сбоев (Failure Detection) Как мы понимаем, что «всё плохо»?

- **Критерии:** Не только код 500, но и тайм-ауты (Timeouts). Часто 5 медленных запросов хуже, чем 5 быстрых ошибок.
- **Скользящее окно (Sliding Window):** Не считайте ошибки «вообще», считайте их во времени. Используйте кольцевой буфер (например, храним состояние последних 100 запросов) или временное окно (последние N секунд), чтобы старые ошибки не влияли на текущее решение.

2. Стратегия восстановления (Recovery Strategy) Как мы возвращаемся в строй?

- Не пытайтесь восстановиться мгновенно. Дайте системе **Sleep Window** (период тишины).
- В режиме Half-Open не пускайте сразу весь трафик. Достаточно 1-2 запросов, чтобы проверить гипотезу «сервис ожил».

3. Fallback (План «Б») Что увидит пользователь, когда цепь разомкнута? Просто вернуть ошибку — это скучно.

- **Graceful Degradation:** Если упал сервис рекомендаций, покажите «Популярные товары» из кэша.
- **Default Values:** Верните пустой список или заглушку.
- **Queueing:** Если это запись данных, сохраните запрос в локальную очередь (Dead Letter Queue) для повторной отправки позже.

Стратегии внедрения Circuit Breaker

Паттерн можно реализовать на разных уровнях стека. Выбор зависит от того, кто должен контролировать надежность: разработчик (в коде) или DevOps (в инфраструктуре).

1. Уровень приложения (Application Libraries)

Самый старый и проверенный способ. Логика встраивается прямо в код микросервиса с помощью SDK.

- **Инструменты:**
  - *Resilience4j (Java):* Современный стандарт. Легковесный, модульный, отличная интеграция со Spring Boot.
  - *Polly (.NET):* Аналог для мира C#.
  - *Hystrix (Netflix):* Легендарная библиотека, давшая старт паттерну, но сейчас находится в режиме **Maintenance Mode**. Использовать в новых проектах не рекомендуется.
- **Плюсы:** Максимальный контроль. Разработчик точно знает контекст ошибки и может написать сложную логику Fallback (например, сходить в кэш или другую БД).
- **Минусы:** Привязка к языку (Java-библиотеку не вставишь в Go-сервис). Требует обновления кода при изменении настроек.

2. Уровень инфраструктуры (Service Mesh)

Circuit Breaker выносится из кода в «сайдкар»-прокси (Sidecar Proxy), который стоит рядом с каждым сервисом.

- **Инструменты:** Istio, Linkerd, Envoy.
- **Как работает:** Прокси перехватывает весь трафик. Если сервис Б начинает отдавать 500-ки, прокси сервиса А сам «размыкает цепь», даже не дергая код приложения.
- **Плюсы: Polyglot** (не зависит от языка программирования). Единая точка настройки для Python, Java и Go сервисов.
- **Минусы:** Сложность эксплуатации (Service Mesh — это сложно). Добавляет сетевую задержку (extra network hop). Меньше контекста для бизнес-логики (прокси не знает, что делать при ошибке, кроме как вернуть 503).

3. Уровень API Gateway

Защита периметра. Шлюз защищает бэкенд от внешнего мира, но не защищает микросервисы друг от друга внутри контура.

- **Инструменты:** Kong, AWS API Gateway, Nginx Plus.
- **Плюсы:** Защищает от перегрузки всю систему целиком. Удобно для централизованного управления политиками (Cross-cutting concerns).
- **Минусы: Coarse-grained** (крупнозернистое) управление. Если шлюз заблокировал доступ к сервису, он заблокирован для всех внешних клиентов.

4. Облачные платформы (Serverless)

В мире Serverless (FaaS) провайдер берет управление на себя.

- **Пример:** AWS Lambda. Если асинхронный вызов функции падает, AWS сам делает ретраи (с экспоненциальной задержкой), а после исчерпания попыток отправляет событие в **Dead Letter Queue (DLQ)**.
- **Плюсы:** "It just works". Не нужно писать код.
- **Минусы: Vendor Lock-in**. Вы жестко привязаны к алгоритмам конкретного облака и имеете мало рычагов влияния на них.

5. Уровень драйверов данных (Database Drivers)

Часто забываемый уровень. «Умные» драйверы баз данных уже имеют встроенные механизмы защиты.

- **MongoDB / Redis Sentinel:** Драйверы умеют обнаруживать падение Primary-ноды и автоматически переключаться на реплику, временно блокируя запросы или буферизируя их.
- **Kafka:** Клиенты (Producer/Consumer) имеют встроенные механизмы ретраев и таймаутов, которые де-факто работают как Circuit Breaker.
- **Совет:** Перед тем как писать свой Circuit Breaker поверх вызовов к БД, проверьте настройки драйвера. Возможно, там уже всё есть.

Заключение: Философия стабильности

Circuit Breaker — это не просто «паттерн проектирования», это **аварийный тормоз** для вашей распределенной системы. В мире микросервисов, где сбои неизбежны, он превращает катастрофические каскадные падения в управляемые, локальные инциденты.

Главные выводы (Key Takeaways):

1. **Fail Fast (Отказывай быстро):** Лучше мгновенно вернуть ошибку и освободить поток, чем заставить пользователя ждать 30 секунд и в итоге всё равно показать ошибку.
2. **Защита ресурсов:** Circuit Breaker спасает вашу систему от истощения пулов соединений и CPU, когда внешние зависимости начинают «тормозить».
3. **Graceful Degradation (Плавная деградация):** Разомкнутая цепь — это не конец света, а шанс показать пользователю кэшированные данные или урезанный функционал вместо «белого экрана смерти».
4. **Гибкость внедрения:** Не существует единственно верного способа реализации.
    - Используйте **библиотеки** (Resilience4j), если нужна сложная бизнес-логика при сбое.
    - Используйте **Service Mesh** (Istio), если у вас зоопарк языков и нужна унификация.

Итог: Системы, которые «умеют ломаться» правильно, в конечном итоге оказываются самыми надежными.