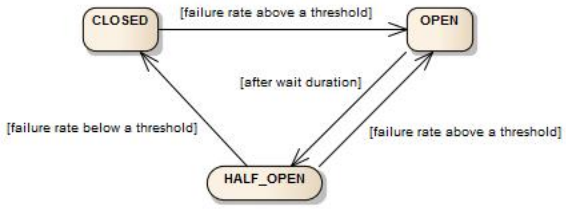
Circuit Breaker

O CircuitBreaker é implementado através de uma máquina de estados finitos com três estados normais: CLOSED, OPEN e HALF_OPEN e dois estados especiais DISABLED e FORCED_OPEN.



O CircuitBreaker usa uma janela deslizante para armazenar e agregar o resultado das chamadas. Você pode escolher entre uma janela deslizante baseada em contagem e uma janela deslizante baseada em tempo. A janela deslizante baseada em contagem agrega o resultado das últimas N chamadas. A janela deslizante baseada em tempo agrega o resultado das chamadas dos últimos N segundos.

Janela deslizante baseada em contagem

A janela deslizante baseada em contagem é implementada com uma matriz circular de N medições.

Se o tamanho da janela de contagem for 10, a matriz circular terá sempre 10 medições.

A janela deslizante atualiza incrementalmente uma agregação total. A agregação total é atualizada quando um novo resultado de chamada é registrado. Quando a medição mais antiga é removida, a medição é subtraída da agregação total e o depósito é redefinido. (Subtrair ao Despejar)

O tempo para recuperar um Snapshot é constante O(1), pois o Snapshot é pré-agregado e independe do tamanho da janela.

O requisito de espaço (consumo de memória) desta implementação deve ser O(n).

Janela deslizante baseada em tempo

A janela deslizante baseada em tempo é implementada com uma matriz circular de N agregações parciais (baldes).

Se o tamanho da janela de tempo for de 10 segundos, a matriz circular terá sempre 10 agregações parciais (buckets). Cada balde agrega o resultado de todas as chamadas que acontecem em um determinado período segundo. (Agregação parcial). O balde (bucket) principal da matriz circular armazena os resultados da chamada do segundo da época atual. As outras agregações parciais armazenam os resultados das chamadas dos segundos anteriores.

A janela deslizante não armazena resultados de chamada (tuplas) individualmente, mas atualiza agregações parciais (bucket) e uma agregação total de forma incremental.

A agregação total é atualizada incrementalmente quando um novo resultado de chamada é registrado. Quando o depósito mais antigo é removido, a agregação total parcial desse depósito é subtraída da agregação total e o depósito é redefinido. (Subtrair ao Despejar)

O tempo para recuperar um Snapshot é constante O(1), pois o Snapshot é pré-agregado e independe do tamanho da janela de tempo.

O requisito de espaço (consumo de memória) desta implementação deve ser quase constante O(n), uma vez que os resultados da chamada (tuplas) não são armazenados individualmente. Apenas N agregações parciais e 1 agregação total total são criadas.

Uma agregação parcial consiste em 3 números inteiros para contar o número de chamadas com falha, o número de chamadas lentas e o número total de chamadas. E um long que armazena a duração total de todas as chamadas.

Taxa de falha e limites de taxa de chamada lenta

O estado do disjuntor muda de FECHADO para ABERTO quando a taxa de falha é igual ou maior que um limite configurável. Por exemplo, quando mais de 50% das chamadas gravadas falharam.

Por padrão, todas as exceções contam como uma falha. Você pode definir uma lista de exceções que devem contar como uma falha. Todas as outras exceções são contadas como um sucesso, a menos que sejam ignoradas. As exceções também podem ser ignoradas para que não contem como falha nem como sucesso.

O CircuitBreaker também muda de FECHADO para ABERTO quando a porcentagem de chamadas lentas é igual ou maior que um limite configurável. Por exemplo, quando mais de 50% das chamadas gravadas duraram mais de 5 segundos. Isso ajuda a reduzir a carga em um sistema externo antes que ele realmente pare de responder.

A taxa de falha e a taxa de chamadas lentas só podem ser calculadas se um número mínimo de chamadas for registrado. Por exemplo, se o número mínimo de chamadas necessárias for 10, pelo menos 10 chamadas deverão ser registradas antes que a taxa de falha possa ser calculada. Se apenas 9 chamadas tiverem sido avaliadas, o CircuitBreaker não abrirá, mesmo que todas as 9 chamadas tenham falhado.

O CircuitBreaker rejeita chamadas com quando está ABERTO. Depois de decorrido um tempo de espera, o estado do CircuitBreaker muda de OPEN para HALF_OPEN e permite um número configurável de chamadas para ver se o back-end ainda está indisponível ou se tornou disponível novamente. Outras chamadas são rejeitadas com um `CallNotPermittedException`, até que todas as chamadas permitidas sejam concluídas. Se a taxa de falha ou taxa de chamada lenta for igual ou maior que o limite configurado, o estado volta a ABRIR. Se a taxa de falha e a taxa de chamada lenta estiverem abaixo do limite, o estado volta a ser FECHADO.

O **CircuitBreaker** suporta mais dois estados especiais, DISABLED (sempre permitir o acesso) e FORCED_OPEN (sempre negar o acesso). Nesses dois estados, nenhum evento de disjuntor (além da transição de estado) é gerado e nenhuma métrica é registrada. A única maneira de sair desses estados é acionar uma transição de estado ou reiniciar o disjuntor.

Variáveis atômicas:

Uma variável atômica, também conhecida como variável atomicamente atualizável ou variável atomicamente acessível, é um tipo especial de variável utilizada em programação concorrente para garantir operações atômicas. Uma operação atômica é uma operação que é executada completamente, sem ser interrompida, sem que outros processos ou threads possam observar seu estado intermediário.

Uma variável atômica permite que várias threads ou processos acessem e atualizem seu valor de forma concorrente, ao mesmo tempo que garantem a consistência dos dados. Isso significa que as operações realizadas em uma variável atômica ocorrem em um único passo indivisível, sem sofrer interferência de outras operações concorrentes.

As variáveis atômicas são usadas para evitar condições de corrida, onde múltiplas threads ou processos podem tentar modificar a mesma variável simultaneamente, resultando em comportamento indefinido ou inconsistente. Com o uso de variáveis atômicas, é possível garantir que as operações de leitura e escrita ocorram de forma segura, sem conflitos ou erros de concorrência.

Em linguagens de programação, existem diferentes implementações de variáveis atômicas, geralmente fornecidas por bibliotecas ou frameworks específicos. Essas implementações podem variar de acordo com a linguagem, mas o objetivo é sempre fornecer mecanismos para operações atômicas e garantir a consistência dos dados em ambientes concorrentes. Exemplos de linguagens que possuem suporte para variáveis atômicas incluem Java, C++, Python e C#.

O **CircuitBreaker** é thread-safe da seguinte forma:

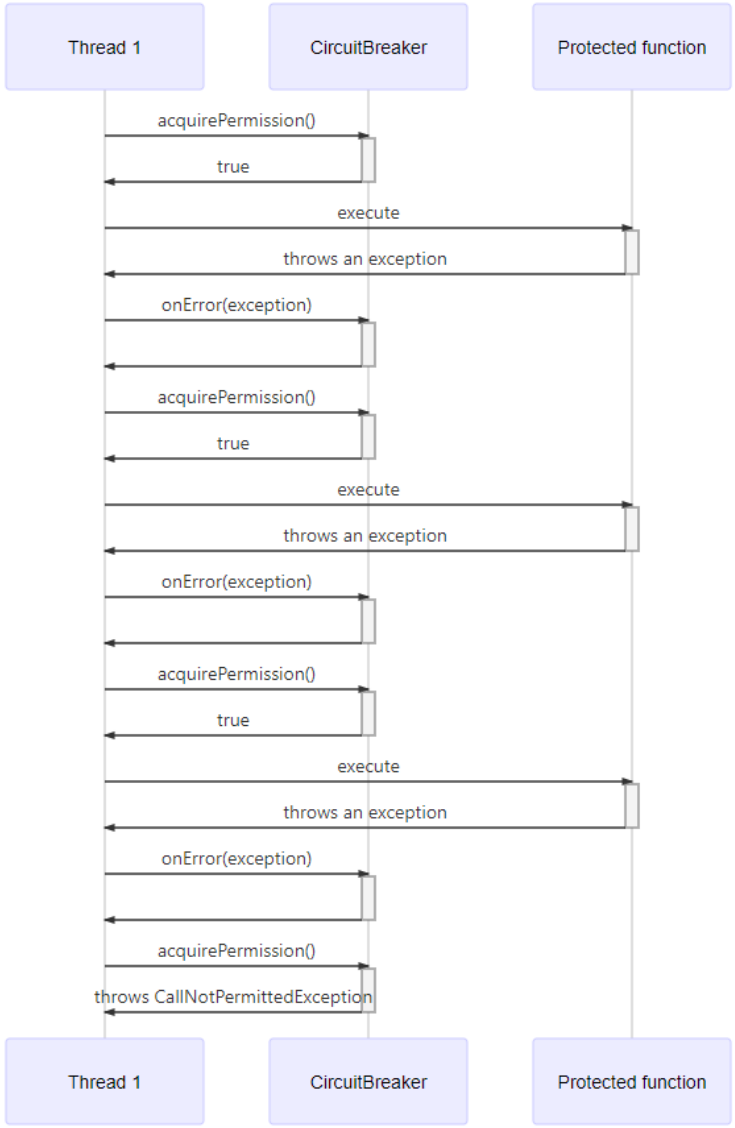
- O estado de um **CircuitBreaker** é armazenado em um **AtomicReference.**
- O **CircuitBreaker** usa operações atômicas para atualizar o estado com funções livres de efeitos colaterais.
- A gravação de chamadas e a leitura de snapshots da janela deslizante são sincronizadas.

Isso significa que a atomicidade deve ser garantida e apenas um thread é capaz de atualizar o estado ou a Sliding Window em um determinado momento.

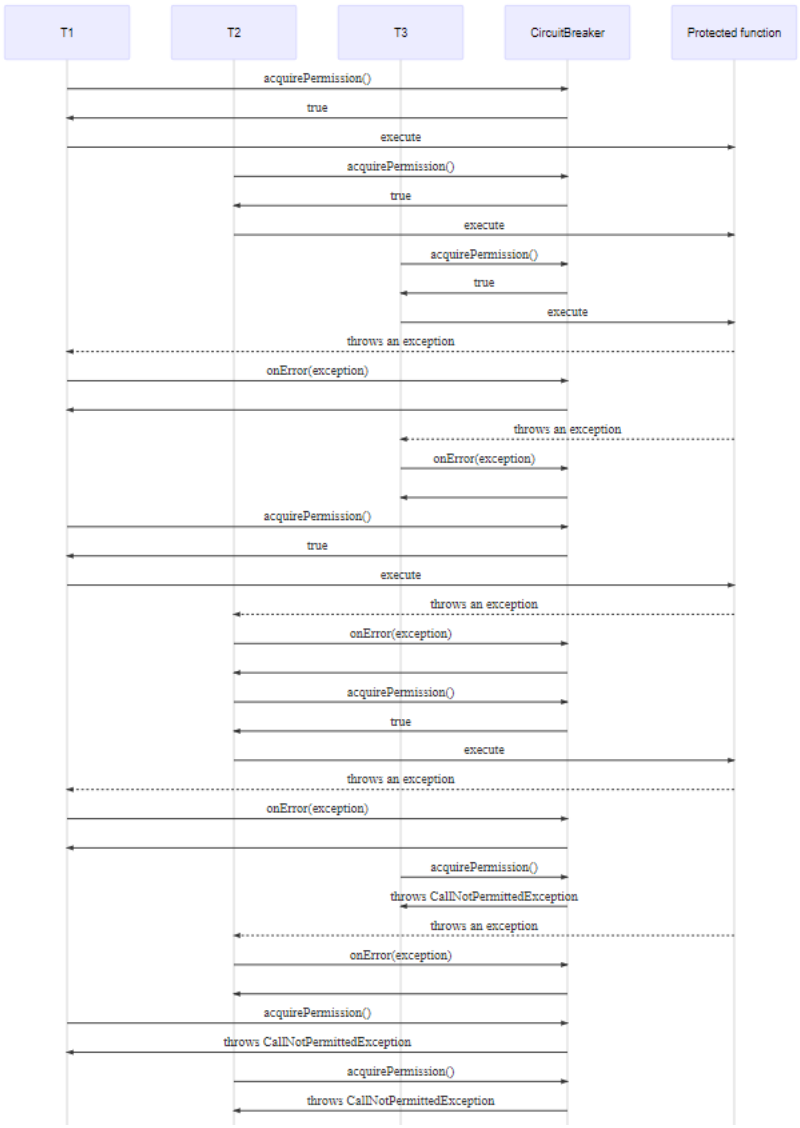
Mas o CircuitBreaker não sincroniza a chamada de função. Isso significa que a própria chamada de função não faz parte da seção crítica. Caso contrário, um CircuitBreaker introduziria uma enorme penalidade de desempenho e gargalo. Uma chamada de função lenta teria um enorme impacto negativo no desempenho/taxa de transferência geral.

Se 20 threads simultâneos solicitarem permissão para executar uma função e o estado do CircuitBreaker for fechado, todos os threads poderão invocar a função. Mesmo que o tamanho da janela deslizante seja 15. A janela deslizante não significa que apenas 15 chamadas podem ser executadas simultaneamente. Se você quiser restringir o número de threads simultâneos, use um Bulkhead. Você pode combinar um Bulkhead e um CircuitBreaker.

Exemplo com 1 Thread:



Exemplo com 3 Threads:



Criar um CircuitBreakerRegistry

Resilience4j vem com um `CircuitBreakerRegistry` in-memory baseado em um **ConcurrentHashMap** que fornece segurança de encadeamento e garantias de atomicidade. Você pode usar o **CircuitBreakerRegistry** para gerenciar (criar e recuperar) instâncias do **CircuitBreaker**. Você pode criar um **CircuitBreakerRegistry** com um padrão global `CircuitBreakerConfig` para todas as instâncias do **CircuitBreaker** da seguinte maneira.

```
CircuitBreakerRegistry circuitBreakerRegistry =
  CircuitBreakerRegistry.ofDefaults();
```

Create and configure a CircuitBreaker

Você pode fornecer seu próprio CircuitBreakerConfig global personalizado. Para criar um CircuitBreakerConfig global personalizado, você pode usar o construtor CircuitBreakerConfig. Você pode usar o construtor para configurar as propriedades a seguir.

Propriedade de configuração	Valor padrão	Descrição
FailureRateThreshold	50	Configura o limite da taxa de falha em porcentagem. Quando a taxa de falha é igual ou

		maior que o limite, o CircuitBreaker faz a transição para abrir e inicia chamadas de curto-circuito.
slowCallRateThreshold	100	Configura um limite em porcentagem. O CircuitBreaker considera uma chamada lenta quando a duração da chamada é maior que `slowCallDurationThreshold` Quando a porcentagem de chamadas lentas é igual ou superior ao limite, o **CircuitBreaker** faz a transição para abrir e inicia chamadas de curto-circuito.
slowCallDurationThreshold	60000 [ms]	Configura o limite de duração acima do qual as chamadas são consideradas lentas e aumenta a taxa de chamadas lentas.
permittedNumberOfCalls InHalfOpenState	10	Configura o número de chamadas permitidas quando o CircuitBreaker está meio aberto (HALF_OPEN).
maxWaitDurationInHalfOpenState	0 [ms]	Configura uma duração máxima de espera que controla a maior quantidade de tempo que um disjuntor pode permanecer no estado meio aberto, antes de mudar para aberto. O valor 0 significa que o disjuntor esperaria infinitamente no estado HalfOpen até que todas as chamadas permitidas fossem concluídas.
slidingWindowType	COUNT_BASED	Configura o tipo de janela deslizante que é usada para registrar o resultado das chamadas quando o CircuitBreaker é fechado. A janela deslizante pode ser baseada em contagem ou baseada em tempo. Se a janela deslizante for **COUNT_BASED**, as últimas chamadas de **slidingWindowSize** serão registradas e agregadas. Se a janela deslizante for **TIME_BASED**, as chamadas dos últimos segundos **slidingWindowSize** serão registradas e agregadas.
slidingWindowSize	100	Configura o tamanho da janela deslizante que é usada para registrar o resultado das chamadas quando o CircuitBreaker é fechado.
minimumNumberOfCalls	100	Configura o número mínimo de chamadas necessárias (por período de janela deslizante) antes que o **CircuitBreaker** possa calcular a taxa de erro ou a taxa de chamada lenta. Por exemplo, se **MinimumNumberOfCalls** for 10, pelo menos 10 chamadas devem ser registradas antes que a taxa de falha possa ser calculada. Se apenas 9 chamadas tiverem sido gravadas, o **CircuitBreaker** não fará a transição para abrir, mesmo que todas as 9 chamadas tenham falhado.
waitDurationInOpenState	60000 [ms]	O tempo que o CircuitBreaker deve esperar antes de passar de aberto para HALF-OPEN.
automaticTransition FromOpenToHalfOpenEnabled	false	Se definido como verdadeiro, significa que o CircuitBreaker fará a transição automática do estado aberto para o estado semi-aberto (HALF_OPEN) e nenhuma chamada será necessária para acionar a transição. Um thread é criado para monitorar todas as instâncias de CircuitBreakers para fazer a transição para HALF_OPEN assim que waitDurationInOpenState passar. Considerando que, se definido como falso, a transição para HALF_OPEN só acontece se uma chamada for feita, mesmo depois que waitDurationInOpenState for passado. A vantagem aqui é que nenhum thread monitora o estado de todos os CircuitBreakers.
recordExceptions	empty	Uma lista de exceções que são registradas como uma falha e, portanto, aumentam a taxa de falha. Qualquer exceção correspondente ou herdada de uma das listas conta como uma falha, a menos que seja explicitamente ignorada por meio de `ignoreExceptions`. Se você especificar uma lista de exceções, todas as outras exceções serão consideradas bem-sucedidas, a menos que sejam explicitamente ignoradas por `ignoreExceptions`.
ignoreExceptions	empty	Uma lista de exceções que são ignoradas e não contam como falha nem como sucesso. Qualquer exceção correspondente ou herdada de uma lista não contará como falha ou sucesso, mesmo que as exceções façam parte de `recordExceptions`.
recordFailurePredicate	throwable -> true por padrão, todas as exceções são registradas como falhas.	Um predicado personalizado que avalia se uma exceção deve ser registrada como uma falha. O Predicado deve retornar true se a exceção for considerada uma falha. O Predicado deve retornar false, se a exceção contar como um sucesso, a menos que a exceção seja explicitamente ignorada por `ignoreExceptions`.
ignoreExceptionPredicate	throwable -> false por padrão, nenhuma exceção é ignorada.	Um predicado personalizado que avalia se uma exceção deve ser ignorada e não conta como falha nem como sucesso. O Predicado deve retornar true se a exceção deve ser ignorada. O Predicado deve retornar false, se a exceção for considerada uma falha.

```
// Create a custom configuration for a CircuitBreaker
CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()
  .failureRateThreshold(50)
  .slowCallRateThreshold(50)
  .waitDurationInOpenState(Duration.ofMillis(1000))
  .slowCallDurationThreshold(Duration.ofSeconds(2))
  .permittedNumberOfCallsInHalfOpenState(3)
  .minimumNumberOfCalls(10)
  .slidingWindowType(SlidingWindowType.TIME_BASED)
  .slidingWindowSize(5)
  .recordException(e -> INTERNAL_SERVER_ERROR
                .equals(getResponse().getStatus()))
  .recordExceptions(IOException.class, TimeoutException.class)
  .ignoreExceptions(BusinessException.class, OtherBusinessException.class)
  .build();

// Create a CircuitBreakerRegistry with a custom global configuration
CircuitBreakerRegistry circuitBreakerRegistry =
  CircuitBreakerRegistry.of(circuitBreakerConfig);

// Get or create a CircuitBreaker from the CircuitBreakerRegistry
// with the global default configuration
CircuitBreaker circuitBreakerWithDefaultConfig =
  circuitBreakerRegistry.circuitBreaker("name1");

// Get or create a CircuitBreaker from the CircuitBreakerRegistry
// with a custom configuration
CircuitBreaker circuitBreakerWithCustomConfig = circuitBreakerRegistry
  .circuitBreaker("name2", circuitBreakerConfig);
```

Você pode adicionar configurações que podem ser compartilhadas por várias instâncias do CircuitBreaker.

```
CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()
  .failureRateThreshold(70)
  .build();

circuitBreakerRegistry.addConfiguration("someSharedConfig", config);

CircuitBreaker circuitBreaker = circuitBreakerRegistry
  .circuitBreaker("name", "someSharedConfig");
```

Você pode sobrescrever as configurações.

```
CircuitBreakerConfig defaultConfig = circuitBreakerRegistry
   .getDefaultConfig();

CircuitBreakerConfig overwrittenConfig = CircuitBreakerConfig
  .from(defaultConfig)
  .waitDurationInOpenState(Duration.ofSeconds(20))
  .build();
```

Se você não quiser usar o CircuitBreakerRegistry para gerenciar as instâncias do CircuitBreaker, também poderá criar instâncias diretamente.

```
// Create a custom configuration for a CircuitBreaker
CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()
  .recordExceptions(IOException.class, TimeoutException.class)
  .ignoreExceptions(BusinessException.class, OtherBusinessException.class)
  .build();

CircuitBreaker customCircuitBreaker = CircuitBreaker
  .of("testName", circuitBreakerConfig);
```

Alternativamente, você pode criar CircuitBreakerRegistry usando seus métodos de construção.

```
Map <String, String> circuitBreakerTags = Map.of("key1", "value1", "key2", "value2");

CircuitBreakerRegistry circuitBreakerRegistry = CircuitBreakerRegistry.custom()
```

```
    .withCircuitBreakerConfig(CircuitBreakerConfig.ofDefaults())
    .addRegistryEventConsumer(new RegistryEventConsumer() {
        @Override
        public void onEntryAddedEvent(EntryAddedEvent entryAddedEvent) {
            // implementation
        }
        @Override
        public void onEntryRemovedEvent(EntryRemovedEvent entryRemoveEvent) {
            // implementation
        }
        @Override
        public void onEntryReplacedEvent(EntryReplacedEvent entryReplacedEvent) {
            // implementation
        }
    })
    .withTags(circuitBreakerTags)
    .build();

CircuitBreaker circuitBreaker = circuitBreakerRegistry.circuitBreaker("testName");
```

Se você deseja conectar sua própria implementação do Registry, pode fornecer uma implementação customizada da Interface RegistryStore e conectar usando o método builder.

```
CircuitBreakerRegistry registry = CircuitBreakerRegistry.custom()
    .withRegistryStore(new YourRegistryStoreImplementation())
    .withCircuitBreakerConfig(CircuitBreakerConfig.ofDefaults())
    .build();
```

Decore e execute uma interface funcional

Você pode decorar qualquer **Callable**, **Supplier**, **Runnable**, **Consumer**, **CheckedRunnable**, **CheckedSupplier**, **CheckedConsumer** ou **CompletionStage** com um **CircuitBreaker**.

Você pode invocar a função decorada com Try.of(…) ou Try.run(…) do Vavr. Isso permite encadear outras funções com mapa, mapa plano, filtro, recuperação ou andThen. As funções encadeadas são invocadas apenas se o CircuitBreaker estiver FECHADO ou HALF_OPEN.

No exemplo a seguir, Try.of(…) retorna uma Mônada Success<String>, se a invocação da função for bem-sucedida. Se a função lançar uma exceção, uma Mônada Failure<Throwable> será retornada e o mapa não será invocado.

```
// Given
CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("testName");

// When I decorate my function
CheckedFunction0<String> decoratedSupplier = CircuitBreaker
        .decorateCheckedSupplier(circuitBreaker, () -> "This can be any method which returns: 'Hello");

// and chain an other function with map
Try<String> result = Try.of(decoratedSupplier)
                .map(value -> value + " world'");

// Then the Try Monad returns a Success<String>, if all functions ran successfully.
assertThat(result.isSuccess()).isTrue();
assertThat(result.get()).isEqualTo("This can be any method which returns: 'Hello world'");
```

Consumir RegistryEvents emitidos

Você pode registrar o consumidor de evento em um CircuitBreakerRegistry e executar ações sempre que um CircuitBreaker for criado, substituído ou excluído.

```
CircuitBreakerRegistry circuitBreakerRegistry = CircuitBreakerRegistry.ofDefaults();
circuitBreakerRegistry.getEventPublisher()
  .onEntryAdded(entryAddedEvent -> {
    CircuitBreaker addedCircuitBreaker = entryAddedEvent.getAddedEntry();
    LOG.info("CircuitBreaker {} added", addedCircuitBreaker.getName());
  })
  .onEntryRemoved(entryRemovedEvent -> {
```

```
    CircuitBreaker removedCircuitBreaker = entryRemovedEvent.getRemovedEntry();
    LOG.info("CircuitBreaker {} removed", removedCircuitBreaker.getName());
  });
```

Consumir CircuitBreakerEvents emitidos

Um CircuitBreakerEvent pode ser uma transição de estado, uma reinicialização do disjuntor, uma chamada bem-sucedida, um erro registrado ou um erro ignorado. Todos os eventos contêm informações adicionais, como tempo de criação do evento e duração do processamento da chamada. Se você deseja consumir eventos, deve registrar um consumidor de eventos.

```
circuitBreaker.getEventPublisher()
    .onSuccess(event -> logger.info(...))
    .onError(event -> logger.info(...))
    .onIgnoredError(event -> logger.info(...))
    .onReset(event -> logger.info(...))
    .onStateTransition(event -> logger.info(...));
// Or if you want to register a consumer listening
// to all events, you can do:
circuitBreaker.getEventPublisher()
    .onEvent(event -> logger.info(...));
```

Você pode usar o CircularEventConsumer para armazenar eventos em um buffer circular com uma capacidade fixa.

```
CircularEventConsumer<CircuitBreakerEvent> ringBuffer =
  new CircularEventConsumer<>(10);
circuitBreaker.getEventPublisher().onEvent(ringBuffer);
List<CircuitBreakerEvent> bufferedEvents = ringBuffer.getBufferedEvents()
```

Você pode usar adaptadores RxJava ou RxJava2 para converter o EventPublisher em um fluxo reativo.

Substituir o RegistryStore

Você pode substituir o RegistryStore na memória por uma implementação personalizada. Por exemplo, se você deseja usar um Cache que remove as instâncias não utilizadas após um determinado período de tempo.

```
CircuitBreakerRegistry circuitBreakerRegistry = CircuitBreakerRegistry.custom()
  .withRegistryStore(new CacheCircuitBreakerRegistryStore())
  .build();
```

Decorate a functional interface

Decore sua chamada para `BackendService.doSomething()` com um CircuitBreaker e execute o fornecedor decorado e recupere-se de qualquer exceção.

```
Supplier<String> decoratedSupplier = CircuitBreaker
    .decorateSupplier(circuitBreaker, backendService::doSomething);

String result = Try.ofSupplier(decoratedSupplier)
    .recover(throwable -> "Hello from Recovery").get();


/* Esse código implementa o padrão de projeto Circuit Breaker (interrupção de circuito) em Java usando a biblioteca Vavr (anteriormente con

Vou explicar cada linha do código:

Supplier<String> decoratedSupplier = CircuitBreaker.decorateSupplier(circuitBreaker, backendService::doSomething);
Nesta linha, um decoratedSupplier é criado. Ele é um Supplier (fornecedor) que encapsula a chamada de um método doSomething do objeto backe

String result = Try.ofSupplier(decoratedSupplier).recover(throwable -> "Hello from Recovery").get();
Nesta linha, o método Try.ofSupplier é usado para executar o decoratedSupplier encapsulado em uma tentativa (try). O método recover especif

Resumindo, esse código cria um fornecedor decorado com um circuit breaker e, em seguida, executa o fornecedor dentro de uma tentativa. Se o
```

Execute a decorated functional interface

Quando você não quer decorar sua expressão lambda, mas apenas executá-la e proteger a chamada por um CircuitBreaker.

```
String result = circuitBreaker
  .executeSupplier(backendService::doSomething);
```

Recuperar de uma exceção

Se você quiser se recuperar de uma exceção depois que o CircuitBreaker registrou como uma falha, você pode encadear o método `Try.recover()` do Vavr. O método de recuperação só é invocado se `Try.ofSupplier()` retornar uma Mônada `Failure<Throwable>`.

```
// Given
CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("testName");

// When I decorate my function and invoke the decorated function
Supplier<String> checkedSupplier =
  CircuitBreaker.decorateSupplier(circuitBreaker, () -> {
    throw new RuntimeException("BAM!");
  });
Try<String> result = Try.ofSupplier(checkedSupplier)
  .recover(throwable -> "Hello Recovery");

// Then the function should be a success,
// because the exception could be recovered
assertThat(result.isSuccess()).isTrue();
// and the result must match the result of the recovery function.
assertThat(result.get()).isEqualTo("Hello Recovery");


/* Esse código mostra um exemplo de uso do circuit breaker em Java usando a biblioteca Vavr (anteriormente conhecida como Javaslang).

Vou explicar cada parte do código:

CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("testName");
Nesta linha, é criado um objeto CircuitBreaker com o nome "testName". O método ofDefaults é usado para criar um circuit breaker com configu

Supplier<String> checkedSupplier = CircuitBreaker.decorateSupplier(circuitBreaker, () -> { throw new RuntimeException("BAM!"); });
Nesta linha, um checkedSupplier é criado. Ele é um Supplier (fornecedor) que encapsula uma função anônima que lança uma exceção RuntimeExce

Try<String> result = Try.ofSupplier(checkedSupplier).recover(throwable -> "Hello Recovery");
Nesta linha, o método Try.ofSupplier é usado para executar o checkedSupplier encapsulado em uma tentativa (try). O método recover especific

assertThat(result.isSuccess()).isTrue();
Nesta linha, é feita uma asserção para verificar se a tentativa foi bem-sucedida. O método isSuccess retorna true se a tentativa foi conclu

assertThat(result.get()).isEqualTo("Hello Recovery");
Nesta linha, é feita outra asserção para verificar se o resultado da tentativa corresponde à string "Hello Recovery". O método get é usado

Em resumo, esse código cria um circuit breaker com configurações padrão, decora um fornecedor com o circuit breaker e realiza uma tentativa
```

Se você deseja se recuperar de uma exceção antes que o **CircuitBreaker** a registre como uma falha, faça o seguinte:

```
Supplier<String> supplier = () -> {
            throw new RuntimeException("BAM!");
        };

Supplier<String> supplierWithRecovery = SupplierUtils
  .recover(supplier, (exception) -> "Hello Recovery");

String result = circuitBreaker.executeSupplier(supplierWithRecovery);

assertThat(result).isEqualTo("Hello Recovery");


/* Este código mostra o uso da classe SupplierUtils para criar um fornecedor (supplier) com tratamento de exceção e recuperação.

Aqui está uma explicação linha por linha do código:

Supplier<String> supplier = () -> { throw new RuntimeException("BAM!"); };
Nesta linha, é criado um fornecedor supplier que encapsula uma função anônima. A função lança uma exceção RuntimeException com a mensagem "
```

```
Supplier<String> supplierWithRecovery = SupplierUtils.recover(supplier, (exception) -> "Hello Recovery");
Nesta linha, é criado um supplierWithRecovery usando a classe SupplierUtils. O método recover é usado para criar um fornecedor decorado que

String result = circuitBreaker.executeSupplier(supplierWithRecovery);
Nesta linha, o supplierWithRecovery é executado dentro do circuit breaker circuitBreaker usando o método executeSupplier. O resultado é atr

assertThat(result).isEqualTo("Hello Recovery");
Nesta linha, é feita uma asserção para verificar se o valor da variável result é igual a "Hello Recovery".

Resumindo, esse código demonstra como usar a classe SupplierUtils para criar um fornecedor decorado com tratamento de exceção e recuperação
```

`SupplierUtils` e `CallableUtils` contêm outros métodos como `andThen`, que podem ser usados para encadear funções. Por exemplo, para verificar o código de status de uma resposta **HTTP**, para que exceções possam ser lançadas.

```
Supplier<String> supplierWithResultAndExceptionHandler = SupplierUtils
  .andThen(supplier, (result, exception) -> "Hello Recovery");

Supplier<HttpResponse> supplier = () -> httpClient.doRemoteCall();
Supplier<HttpResponse> supplierWithResultHandling = SupplierUtils.andThen(supplier, result -> {
    if (result.getStatusCode() == 400) {
       throw new ClientException();
    } else if (result.getStatusCode() == 500) {
       throw new ServerException();
    }
    return result;
});
HttpResponse httpResponse = circuitBreaker
  .executeSupplier(supplierWithResultHandling);


/* Esse código demonstra o uso da classe SupplierUtils para criar fornecedores (suppliers) com tratamento de resultados e exceções.

Vou explicar cada parte do código:

Supplier<String> supplierWithResultAndExceptionHandler = SupplierUtils.andThen(supplier, (result, exception) -> "Hello Recovery");
Nesta linha, é criado um supplierWithResultAndExceptionHandler usando a classe SupplierUtils. O método andThen é usado para criar um fornec

Supplier<HttpResponse> supplier = () -> httpClient.doRemoteCall();
Nesta linha, é criado um fornecedor supplier que encapsula a chamada de um método doRemoteCall em um objeto httpClient. O tipo de retorno d

Supplier<HttpResponse> supplierWithResultHandling = SupplierUtils.andThen(supplier, result -> { ... });
Nesta linha, é criado um supplierWithResultHandling usando a classe SupplierUtils. O método andThen é usado para criar um fornecedor decora

HttpResponse httpResponse = circuitBreaker.executeSupplier(supplierWithResultHandling);
Nesta linha, o supplierWithResultHandling é executado dentro do circuit breaker circuitBreaker usando o método executeSupplier. O resultado

Em resumo, esse código mostra como usar a classe SupplierUtils para criar fornecedores decorados com tratamento de resultados e exceções. I
```

Reset CircuitBreaker

O CircuitBreaker suporta a redefinição para seu estado original, perdendo todas as métricas e redefinindo efetivamente sua Janela Deslizante.

```
CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("testName");
circuitBreaker.reset();
```

Transição para estados manualmente

```
CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("testName");
circuitBreaker.transitionToDisabledState();
// circuitBreaker.onFailure(...) won't trigger a state change
circuitBreaker.transitionToClosedState(); // will transition to CLOSED state and re-enable normal behaviour, keeping metrics
circuitBreaker.transitionToForcedOpenState();
// circuitBreaker.onSuccess(...) won't trigger a state change
circuitBreaker.reset(); //  will transition to CLOSED state and re-enable normal behaviour, losing metrics
```