05

Trabalhando com Cache

Trabalhando com Cache

Um ponto muito importante de se preocupar em uma aplicação de alta escalabilidade é com o gerenciamento dos recursos consumidos pela máquina, dentre eles memória e acesso a disco são dois dos mais sensíveis.

Sabemos que o Node.js trabalha muito bem e performa muito bem em operações de I/O e é uma boa prática que nosso código faça mais execuções de I/O do que de armazenamento em memória.

Porém é comum que a aplicação possua dados que não sofram variações com tanta frequência. E se esses dados estão armazenados em um banco de dados relacional, como é o nosso caso com MySQL, passa a ser um bom sinal de alerta para verificarmos se o uso de uma camada de cache que amenize os acessos ao banco seria uma melhoria na performance da aplicação.

Cache de dados com Memcached

Um outro fator que pesa muito positivamente no momento de definir utilizar uma política de cache ou não é o fato que já existem no mercado muitos *frameworks* especialistas nessa função, que abstraem as partes complicadas de infraestrutura, são confiáveis e de fato tendem a melhorar a performance das aplicações em que são utilizados.

O **Memcached** é um ótimo exemplo desses frameworks citados. Ele é definido na verdade como um **sistema de caching de objetos em memória** grátis, *open source*, distribuído e de alta performance, genérico por natureza, mas com uma forte intenção de acelerar o processamento de aplicações web dinâmicas, aliviando a carga de acessos ao banco de dados. Exatamente o objetivo que tínhamos ao pensar em implementar uma política de cache no PayFast.

Ele funciona baseado em um esquema chave-valor que armazena pequenos pedaços de dados de qualquer tipo desejado (string, objetos...) em memória. Podendo esses dados ser oriundos de consultas à banco de dados, à outras APIs ou até mesmo do carregamento de páginas.

Ele é um framework simples, porém bastante poderoso. Fácil de instalar, fazer *deploy* e de desenvolver sobre ele, por ter um *design* simples. Além de prover APIs para diversas linguagens de programação.

A instalação do **Memcached** é muito simples. Uma forma bem padrão de fazê-la é baixar a última versão direto do site oficial, descompactar e instalar:

```
wget http://memcached.org/latest
tar -zxvf memcached-1.x.x.tar.gz
cd memcached-1.x.x
./configure && make && make test && sudo make install
```

Este exemplo mostra a instalação feita diretamente no terminal.

Após instalado, basta executar um comando no próprio terminal para que ele suba e fique pronto para receber conexões:

```
memcached -vv
```

O parâmetro `-vv` indica que queremos que ele rode num modo 'verboso' nível 2.

Após essa execução, ele já exibe no terminal um informativo do seu status atual, que fica sendo atualizado em tempo real conforme o cache for utilizado

```
<18 server listening (auto-negotiate)
<19 send buffer was 9216, now 5592405
<19 server listening (udp)
<20 server listening (udp)
<21 server listening (udp)
<23 send buffer was 9216, now 5592405
```

Implementando um cliente para cache

Além de o próprio **Memcached** já disponibilizar diversas APIs para facilitar a integração de diferentes linguagens de programação com o framework, as linguagens também fazem seu esforço para se manter facilmente acessíveis a recursos importantes como esse.

Com a comunidade Node, isso não é diferente. Existem inúmeros pacotes para integração do Node com o Memcached. Um dos mais importantes, e que será utilizado pelo PayFast, é um que leva o mesmo nome do framework de caching.

Para fazer sua instalação, basta usar o npm mais uma vez:

```
npm install --save memcached
```

O uso dessa lib também é bem simples. A primeira coisa a fazer é carregar o módulo no arquivo em que se deseja ter o cliente e, em seguida, instanciar um novo objeto da lib, que no seu retorno, entrega o cliente que foi criado:

```
var memcached = require('memcached');

var client = new memcached('localhost:11211',
          {
            retries:10,
            retry:10000,
            remove:true
          });
```

Repare que a construção do `memcached` recebe dois parâmetros: a url onde o Memcached está rodando e um json com detalhes sobre as configurações desejadas para criação desse cliente. A url utiliza uma porta que sequer definimos ao subir o Memcached, mas tudo bem pois esta é a porta default do serviço.

Os parâmetros passados no json são configurações específicas para este cliente e não sobrescrevem nenhuma configuração que tenha sido definida no servidor.

- `retries` : *10*, o número de retentativas feitas por request.
- `retry` : *10000*, o tempo entre a falha de um servidor e uma tentativa de colocá-lo de volta em serviço.
- `remove` : *true*, autoriza a remoção automática de servidores mortos do pool.

Existem diversas outras propriedades que podem ser utilizadas. É possível consultar a lista completa de propriedades no próprio GitHub do projeto: https://github.com/3rd-Eden/memcached/blob/master/README.md (https://github.com/3rd-Eden/memcached/blob/master/README.md)

Com o cliente em mãos fica fácil consultar se uma chave está no cache:

```
client.get('pagamento-' + id, function (err, data) {

  if (err || !data){
      console.log('MISS - chave não encontrada no cache');
  } else {
      console.log('HIT - valor:' + data);
  }
});
```

A sintaxe e forma de uso são bem simples. O cliente funciona basicamente como um mapa, a partir do qual é possível fazer um `get` passando a chave que se quer consultar. Como é comum no mundo Node, o segundo parâmetro é uma função de callback, que recebe um possível erro e um objeto de dados.

Caso aconteça um **HIT** no cache, ou seja, a chave de fato está presente, a variável de dados, que definimos como `data` é quem armazena esse valor e ele pode ser utilizado para o que a regra de negócio pedir.

Caso aconteça um **MISS**, que é quando a chave não é encontrada, o objeto `data` é retornado vazio. A outra opção que pode acontecer é que haja algum erro na execução. Nesse caso, essa informação virá na variável **err**.

Para inserir uma nova chave no cache também é bem simples:

```
client.set('pagamento-3', dados, 100000, function (err) {
   console.log('nova chave: pagamento-3');
 });
```

O método invocado agora é o `set` e são passados como parâmetros a chave, o valor, o tempo que ela deve permanecer no cache a uma função de callback.