Java 開発 2.0: Redis を実際に使用する

読み取り処理が大量に行われるアプリケーションで、Redis は memcached に比べてどのように優れているのか

Andrew Glover
Author and developer
Beacon50

2012年 1月 27日

Redis には memcached と多くの共通点がありますが、機能の数という点では Redis のほうが充実しています。今月の連載「Java 開発 2.0」では、Andrew が自作の位置情報ベースのモバイル・アプリケーションに、試しに Redis を追加して (Java ベースの Redis クライアント Jedis で) 使用してみます。Redis が単純なデータ・ストアとして機能する仕組みを学んだ後、今度は Redis を超高速の軽量キャッシュとして使用してみます。

このシリーズの他の記事を見る

この連載について

Java 技術が初めて登場してから現在に至るまでに、Java 開発の様相は劇的に変化しました。成熟したオープンソースのフレームワーク、そしてサービスとして提供される信頼性の高いデプロイメント・インフラストラクチャーを利用できる (借りられる) おかげで、今では Java アプリケーションを短時間かつ低コストでアセンブルし、テスト、実行、保守することが可能になっています。この連載では Andrew Glover が、この新たな Java 開発パラダイムを可能にする多種多様な技術とツールを詳しく探ります。

この連載では以前、NoSQL の概念について説明し、Google の Bigtable や Amazon の SimpleDB など、Java プラットフォームに対応するさまざまな NoSQL データ·ストアを紹介しました。また、MongoDB や CouchDB などといった従来型に近いサーバー・ベースのデータ・ストアも取り上げました。いずれのデータ・ストアにも長所と短所がありますが、特定のドメインでのシナリオに当てはめると、それは一層顕著になります。

今月の連載「Java 開発 2.0」でスポットライトを当てるのは、軽量のキー・バリュー型データ・ストア、Redis です。ほとんどの NoSQL 実装は基本的にキー・バリュー型ですが、サポートする値の型という点では Redis は群を抜いて充実しており、文字列型、リスト型、セット型、ハッシュ型までサポートします。このことから、Redis はデータ構造サーバーに分類されることがよくあります。また、Redis には並外れて高速なデータ・ストアとしての評判もあり、ある特定のタイプの状況に有効な最適な選択肢となります。

新しいことを理解するには、すでによく理解しているものと比較すると役立つことがあります。そこで、この記事では Redis について詳しく探るにあたり、まず Redis と memcached の類似点を検討するところから始めます。その後、特定のアプリケーション・シナリオで memcached よりも優れた能力を発揮する Redis の主な機能を紹介し、最後に Redis をモデル・オブジェクトの従来のデータ・ストアとして使用する方法を説明します。

Redis と memcached

memcached は、ターゲットのキーと値をメモリー・キャッシュに取り込むことによって機能する、よく知られたインメモリー・オブジェクト・キャッシュ・システムです。オブジェクトをメモリー内にキャッシュする memcached は、読み取り処理によってディスクへアクセスする際に発生する I/O コストを回避します。Web アプリケーションとデータベースの間で memcached を使用すると、読み取り処理のパフォーマンスが向上することになるため、memcached はデータの高速検索が必要なアプリケーションに適しています。そのようなアプリケーションの一例は、株価検索サービスです。株価検索サービスでは、memcached を使用しないとしたら、証券コードと銘柄との対応情報のような極めて静的なデータから株価情報のような動的データまでを取得するために、データベースにアクセスすることになります。

MemcacheDB

Redis を memcached と比べるのは、必ずしも公平とは言えません。それよりも遥かに妥当な比較対象は、データ永続化を目的に設計された分散キー・バリュー型ストレージ・システム、MemcacheDB です。MemcacheDB は Redis とかなり似ているのみならず、memcached のクライアント実装と簡単に通信できるというメリットもあります。

一方、memcached にはいくつかの限界があります。その 1 つは、値のすべてが単純な文字列であることです。memcached の代替手段としての Redisは、それよりも充実したデータ型をサポートしているだけでなく、一部のベンチマークでは、Redis の速度は memcached を大きく上回ることも示されています。Redis は充実したデータ型をサポートするため、memcached で保管できるデータよりも極めて高度なデータをメモリーに保管することができます。しかも、memcached とは異なり、Redis ではそのデータを永続化することもできます。

Redis は素晴らしいキャッシング・ソリューションになりますが、Redis の充実した機能群からはキャッシュ以外の用途も考えられます。Redis はデータをディスクに保管し、複数のノードにデータを複製できることから、Redis を従来のデータ・モデルのデータ・リポジトリーとして利用することができます (つまり、RDBMS とほとんど同じように Redis を使用できるということです)。また、Redis はクエリー・システムとして採用されることもよくあります。その場合、Redis は作業キューのバックアップ永続ストアのベースとなり、作業キューが Redis のリスト型を使用するという仕組みです。このような方法で Redis を使用している大規模インフラストラクチャーの例としては、GitHub が挙げられます。

Redis を使用してみましょう！

Redis を使い始めるには、Redis にアクセスできなければなりませんが、その方法としては、ローカルに Redis をインストールする方法、あるいはホスト・プロバイダー経由でアクセスするという方法があります。Mac を使っている場合、インストール・プロセスはこの上なく簡単です。Windows を使用しているとしたら、Cygwin をインストールする必要があります。ホスト・プロバイダーを探している場合は、Redis4You に無料のプランが用意されています。Redis にどのよ

うにアクセスするかに関わらず、この記事で後ほど説明する例に従うことはできます。ただし指摘しておく点として、Redis のホスト・プロバイダーを使用してキャッシュするという方法は、あまり優れたキャッシング・ソリューションにはならないかもしれません。なぜなら、ネットワーク遅延によって、パフォーマンスの向上が帳消しになってしまう可能性があるからです。

Redis を操作するには、コマンドを使用します。つまり Redis には、SQL のような問い合わせ言語はないのです。Redis の操作は、従来の `map` データ構造を操作するのと非常によく似ていて、あらゆるデータにはキーと値があり、それぞれの値には豊富なデータ型が関連付けられています。さらに、すべてのデータ型にはそれぞれ固有のコマンド一式があります。例えば、何らかのキャッシング・スキームで単純なデータ型を使用する場合には、`set` および `get` コマンドを使用することができます。

Redis のインスタンスを操作するには、コマンドライン・シェルを使用することができます。また、Redis をプログラムによって操作するためのクライアント実装も複数あります。リスト 1 は、基本的なコマンドを使用したコマンドライン・シェルでの単純な操作の例です。

リスト 1. 基本的な Redis コマンドを使用する

```
redis 127.0.0.1:6379> set page registration
OK
redis 127.0.0.1:6379> keys *
1) "foo"
2) "page"
redis 127.0.0.1:6379> get page
"registration"
```

上記の例では、`set` コマンドによって page というキーを registration という値に関連付けています。次に、`keys` コマンドを呼び出します (末尾の `*` は、有効なインスタンス・キーのすべてを表示することを意味します)。`keys` コマンドによって、`page` キーだけでなく、`foo` というキーもあることが示されます。キーに関連付けられた値は、`get` コマンドによって取得することができます。ただし、`get` コマンドで取得できる値は、文字列のみであることに注意してください。例えば、キーの値がリストの場合には、リスト型に固有のコマンドを使ってリストの要素を取得する必要があります (値の型を問い合わせるためのコマンドもあります)。

Jedis を使用することによる Java アプリケーションへの Redis の統合

Redis を Java アプリケーションに統合したいというプログラマーに Redis チームが推奨しているのは Jedis というプログラムです。Jedis は軽量のライブラリーで、ネイティブ Redis コマンドを単純な Java メソッドへとマッピングします。Jedis によって、例えばリスト 2 のように単純な値を取得したり、設定したりすることができます。

リスト 2. Java コードでの基本的な Redis コマンド

```
JedisPool pool = new JedisPool(new JedisPoolConfig(), "localhost");
Jedis jedis = pool.getResource();

jedis.set("foo", "bar");
String foobar = jedis.get("foo");
assert foobar.equals("bar");

pool.returnResource(jedis);
pool.destroy();
```

リスト 2 ではまず、接続プールを構成して、接続を取得します (通常の JDBC シナリオで行う方法とほとんど同じです)。そしてリストの最後でこの接続を破棄しています。この接続プール・ロジックに挟まれたコードでは、値 `“bar”` をキー `“foo”` に関連付けて設定します。このキーは、`get` コマンドで取得します。

memcached と同じように、Redis では値に有効期限を関連付けることができます。したがって、値 (例えば、ある時点での株の取引価格) を設定して、その値を最終的には Redis キャッシュからパージすることができます。Jedis で有効期限を設定するとしたら、`set` を呼び出した後で、値に有効期限を関連付けることになります (リスト 3 を参照)。

リスト 3. 有効期限を設定可能な Redis の値

```
jedis.set("gone", "daddy, gone");
jedis.expire("gone", 10);
String there = jedis.get("gone");
assert there.equals("daddy, gone");

Thread.sleep(4500);

String notThere = jedis.get("gone");
assert notThere == null;
```

リスト 3 では `expire` を呼び出して、”gone” の値が 10 秒で失効するように設定しました。`Thread.sleep` が呼び出された後は、”gone” に対して `get` を実行すると `null` が返されます。

Redis でのデータ型

リストやハッシュなどの Redis のデータ型を操作するには、そのための専用コマンドを使用する必要があります。例えば、リストを作成するには、キーに複数の値を追加するという方法を使用することができます。リスト 4 のコードで実行している `rpush` コマンドは、リストの右側つまり最後に値を追加するためのコマンドです (これに対応する `lpush` コマンドは、リストの先頭に値を追加します)。

リスト 4. Redis のリスト

```
jedis.rpush("people", "Mary");
assert jedis.lindex("people", 0).equals("Mary");

jedis.rpush("people", "Mark");

assert jedis.llen("people") == 2;
assert jedis.lindex("people", 1).equals("Mark");
```

Redis にはデータ型を操作するための多種多様なコマンドがあります。その上、それぞれのデータ型には固有のコマンド一式もありますが、ここでは個々のコマンドについて説明することはしません。代わりに、実際のアプリケーション開発の中でそれらのコマンドのいくつかを使っているところを紹介します。

キャッシング・ソリューションとしての Redis

Redis は簡単にキャッシング・ソリューションとして使用できると言いましたが、ちょうど私もキャッシング・ソリューションを必要としていたところです。ここで取り上げるサンプル・ア

プリケーションでは、私が作成した Magnus という位置情報ベースのモバイル Web サービスに Redis を統合します。

この連載の記事を見逃してしまった読者のために説明しておくと、私はこの Magnus を、最初に Play フレームワークを使って実装しました。それ以降、さまざまな実装で Magnus を開発したり、リファクタリングしたりしています。Magnus は、HTTP `PUT` リクエストによって JSON 文書を取得する単純なサービスです。取得する JSON 文書には、特定のアカウント (すなわち、モバイル機器を使っているユーザー) の位置情報が記されています。

今回は、Magnus にキャッシング機能を統合します。つまり、頻繁に変更されることのないデータをメモリーに保管することによって、検索という形での I/O トラフィックを減らすという狙いです。

Magnus がキャッシュします！

リスト 5 で最初のステップとなるのは、`get` 呼び出しによって受信されたアカウント名 (これが、キーです) が Redis に存在するかどうかを調べることです。`get` を呼び出すと、アカウント ID が値として返されるか、または `null` が返されます。値が返された場合、その値を `acctId` 変数として使用します。`null` が返された場合は (Redis にアカウントの名前がキーとして存在しないことを意味します)、MongoDB でアカウントの値を検索し、`set` コマンドでその値を Redis に追加します。

このコードの長所は実行速度です。次回、リクエストされたアカウントが位置情報を送信するときには、そのアカウントの ID を (インメモリー・キャッシュとして機能する) Redis から取得できるため、MongoDB にアクセスする必要はありません。したがって、読み取り処理による I/O コストが発生しないというわけです。

リスト 5. インメモリー・キャッシュとして Redis を使用する

```
"/location/:account" {
  put {
    def jacksonMapper = new ObjectMapper()
    def json = jacksonMapper.readValue(request.contentText, Map.class)
    def formatter = new SimpleDateFormat("dd-MM-yyyy HH:mm")
    def dt = formatter.parse(json['timestamp'])
    def res = [:]

    try{

      def jedis = pool.getResource()
      def acctId = jedis.get(request.parameters['account'])

      if(!acctId){
        def acct = Account.findByName(request.parameters['account'])
        jedis.set(request.parameters['account'], acct.id.toString())
        acctId = acct.id
      }

      pool.returnResource(jedis)
      new Location(acctId.toString(), dt, json['latitude'].doubleValue(),
      json['longitude'].doubleValue() ).save()
      res['status'] = 'success'
    }catch(exp){
      res['status'] = "error ${exp.message}"
    }
   response.json = jacksonMapper.writeValueAsString(res)
```

```
  }
}
```

リスト 5 の Magnus 実装 (Groovy で作成) では、データ・モデル・ストレージに相変わらず NoSQL 実装を使用していることに注意してください。Redis の用途は、データを検索するためのキャッシュ実装でしかありません。主なアカウント・データは MongoDB に保管されていて、Redis データ・ストアはローカルで動作しています。実際、この MongoDB は MongoHQ.com にあるため、以降にアカウント ID を検索するときには、Magnus の実行速度が飛躍的に向上します。

しかし、ちょっと待ってください。MongoDB と Redis は両方とも必要なのでしょうか？どちらか一方だけで済ませることはできないのでしょうか？

ORM のための Node.js

Redis のための ORM (Object Relational Mapping) 的なマッピングを提供しているプロジェクトは多数あります。なかでも極めて影響力が大きいのは、Ruby をベースとした Ohm というプロジェクトです。このプロジェクトから派生した Java ベースのプロジェクト (JOhm という名前のプロジェクト) も調べてみましたが、結局のところ、Node 用に作成された Nohm というプロジェクトを使用することに決めました。Ohm とそこから派生したプロジェクトの長所は、Redis ベースのデータ構造にオブジェクト・モデルをマッピングできるところにあります。したがって、モデル・オブジェクトは永続オブジェクトであると同時に、(大抵は) 極めて高速に読み取ることができます。

Nohm を使用した結果、私は短時間で Magnus アプリケーションを JavaScript で作成し直し、`Location` オブジェクトを簡単に永続化することができました。リスト 6 で定義した `Location` モデルには、3 つのプロパティーが組み込まれています (この例を単純にしておくために、`timestamp` は本物のタイムスタンプではなく、文字列にしていることに注意してください)。

リスト 6. Node.js での Redis による ORM

```
var Location = nohm.model('Location', {
 properties: {
     latitude: {
       type: 'float',
       unique: false,
       validations: [
         ['notEmpty']
       ]
     },
  longitude: {
       type: 'float',
       unique: false,
       validations: [
         ['notEmpty']
       ]
     },
  timestamp: {
       type: 'string',
       unique: false,
       validations: [
         ['notEmpty']
       ]
        }
     }
});
```

Node の Express フレームワークにより、この新しい Nohm `Location` オブジェクトは実に簡単に使用することができます。サンプル・アプリケーションの `PUT` 実装では、受信した JSON 値を取得し、Nohm の `p` を呼び出すことでこれらの値を `Location` のインスタンスに格納します。その後、インスタンスが有効であるかどうかをチェックし、有効であれば、インスタンスを永続化します。

リスト 7. Node の Express.js で Nohm を使用する

```
app.put('/', function(req, res) {
  res.contentType('json');

  var location = new Location;
  location.p("timestamp", req.body.timestamp);
  location.p("latitude", req.body.latitude);
  location.p("longitude", req.body.longitude);

  if(location.valid()){
   location.save(function (err) {
    if (!err) {
      res.send(JSON.stringify({ status: "success" }));
    } else {
     res.send(JSON.stringify({ status: location.errors }));
    }
   });
  }else{
   res.send(JSON.stringify({ status: location.errors }));
  }
});
```

リスト 7 を見るとわかるように、Redis を極めて高速なインメモリーのデータ・ストアにするのは至って簡単です。場合によっては、memcached より優れたキャッシュになることさえあります。

まとめ

Redis はさまざまなデータ・ストレージのシナリオに役立つだけでなく、データをディスクに永続化できることから (そして、豊富なデータ・セットをサポートすることから)、場合によっては memcached の立派なライバルになります。ドメインにとって意味がある場合には、Redis をデータ・モデルおよびキューのバックアップ・ストアとして使用することもできます。Redis クライアント実装は、現在使われているほぼすべてのプログラミング言語に移植されています。

Redis は、RDMBS に完全に置き換わるものでも、MongoDB のようにクエリー機能を満載した大掛かりなストアでもありません。しかし多くの場合、Redis はこれらの技術と共存することができます。この記事で明らかにしたように、Redis は、データ検索に重きを置くアプリケーションや、Redis の高速なアトミック演算によってリアルタイムの統計処理を実現できるアプリケーションには絶好のスタンドアロン・データ・ストレージ・ソリューションとなります。

著者について

Andrew Glover

Andrew Glover は、ビヘイビア駆動開発、継続的インテグレーション、アジャイル・ソフトウェア開発に情熱を持つ開発者であるとともに、著者、講演者、起業家でもあります。また、easyb BDD (Behavior-Driven Development) フレームワークの創始者、そして『継続的インテグレーション入門 開発プロセスを自動化する47の作法』、『Groovy in Action』、『Java Testing Patterns』の 3 冊の本の共著者でもあります。詳細は彼のブログにアクセスするか、Twitter で彼をフォローしてください。