第 4-1 课：Spring Boot 操作 Memcache

> 《精通 Spring Boot 42 讲》共分五大部分，第四部分主要讲解 Spring Boot 和中间件的使用，共 10 课，中间件是互联网公司支撑高并发业务的必备组件，常用的组件有缓存、消息中间件、NoSQL 数据库、定时任务等。常用的缓存中间件有 Memcache 和 Redis，缓存主要支撑业务架构中高速读写；常用的消息中间件有 ActiveMQ 、RabbitMQ，使用消息中间件的意义是，尽快地完成主线交易，其他非实时业务异步或者解耦完成；最主流的 NoSQL 有 MongoDB、ElasticSearch，前者主要是解决分布式存储和检索的问题，后者主要解决分布式文档检索的解决方案；定时任务常常使用开源框架 Quartz。以上的这些内容我们都会在第四部分进行学习。

在常见的企业架构中，随着公司业务高速发展，最先出现瓶颈的是数据库，这个时候很多企业就会考虑使用缓存来缓解数据库的压力，这是缓存使用最多的场景之一；另外在高并发抢购、分布式 Session 等场景下，也会使用缓存来提高系统的高可用性。常用的缓存中间件有 Memcache 和 Redis，今天我们先来学习 Memcache 的使用。

Memcache 介绍

Memcache 是一个自由和开放源代码、高性能、分配的内存对象缓存系统。简单来说，Memcache 是一个高性能的分布式内存对象的 key-value 缓存系统，用于加速动态 Web 应用程序，减轻数据库负载，现在也有很多人将它作为内存式数据库在使用。

它可以应对任意多个连接，使用非阻塞的网络 IO，由于它的工作机制是在内存中开辟一块空间，然后建立一个 Hash 表，Memcached 自动管理这些 Hash 表。

Memcache 由国外社区网站 LiveJournal 开发团队开发，设计理念就是小而强大，它简单的设计促进了快速部署、易于开发并解决面对大规模的数据缓存的许多难题，而所开放的 API 使得 Memcache 能用于 Java、C/C++/C#、Perl、Python、PHP、Ruby 等大部分流行的程序语言。

> Memcache 和 Memcached 的区别：Memcache 是这个项目的名称，而 Memcached 是服务器端的主程序名称。

Memcache 特点

协议简单

Memcache 的服务端客户端通信使用简单的文本协议，通过 Telnet 即可在 Memcached 上存取数据。

基于 Libevent 的事件处理

Libevent 是一套跨平台的事件处理接口的封装，能够兼容包括这些操作系统：Windows/Linux/BSD/Solaris 等操作系统的的事件处理，包装的接口包括：poll、select（Windows）、epoll（Linux）、

kqueue（BSD）/dev/pool（Solaris）。

Memcache 使用 Libevent 来进行网络并发连接的处理，能够保持在很大并发情况下，仍旧能够保持快速的响应能力。

内置内存存储方式

Memcache 中保存的数据都存储在 Memcache 内置的内存存储空间中。由于数据仅存在于内存中，因此重启 Memcache、重启操作系统会导致数据全部丢失。Memcache LRU（Least Recently Used）算法自动删除不使用的缓存，不过这个功能是可以配置的，Memcache 启动时通过“-M”参数可以禁止 LRU。不过，Memcache 本身是为缓存而设计的，建议开启 LRU。

不适应场景

- 缓存对象不能大于 1 MB
- key 的长度大于 250 字符
- Memcache 未提供任何安全策略
- 不支持久化

Memcache 安装

在 Centos 下安装使用 yum 命令安装 Memcache 非常简单：

```
yum install -y memcached
```

启动：

```
/usr/bin/memcached -b  -p 11211 -m 150 -u root >> /tmp/memcached.log  &
```

启动参数可以配置，常用的命令选项如下：

- m 内存
- c 最大链接数
- p 端口
- u 用户
- t 线程数

查看 memcached 是否在运行：

```
ps -ef | grep memcached
```

Memcache 客户端

Memcached Client 目前有 3 种：

- Memcached Client for Java（已经停止更新）
- SpyMemcached（已经停止更新）
- XMemcached（主流使用）

Memcached Client for Java 比 SpyMemcached 更稳定、更早、更广泛；SpyMemcached 比 Memcached Client for Java 更高效；XMemcached 比 SpyMemcache 并发效果更好。

曾经有一段时间 SpyMemcached 使用比较广泛，我简单介绍一下。

Spymemcached 介绍

Spymemcached 是一个采用 Java 开发的异步、单线程的 Memcached 客户端，使用 NIO 实现。Spymemcached 是 Memcached 的一个流行的 Java Client 库，性能表现出色，广泛应用于 Java + Memcached 项目中。

Spymemcached 最早由 Dustin Sallings 开发，Dustin 后来和别人一起创办了 Couchbase（原 NorthScale），职位为首席架构师，2014 年加入 Google。

XMemcached 简介

现在使用最广泛的 Memcache Java 客户端是 XMemcached，它是一个新的 Java Memcache Client。Memcached 通过它的自定义协议与客户端交互，而 XMemcached 就是它的一个 Java 客户端实现。相比其他客户端，XMemcached 有什么优点呢?

XMemcached 的主要特性

XMemcached 支持设置连接池、宕机报警、使用二进制文件、一致性哈希算法、进行数据压缩等操作，总结如下：

- 高性能，由 Nio 支持；
- 协议完整，Xmemcached 支持所有的 Memcached 协议，包括 1.4.0 正式开始使用的二进制协议；
- 支持客户端分布，提供了一致性哈希（Consistent Hash）算法的实现；
- 允许设置节点权重，XMemcached 允许通过设置节点的权重来调节 Memcached 的负载，设置的权重越高，该 Memcached 节点存储的数据将越多，所承受的负载越大；
- 动态增删节点，Memcached 允许通过 JMX 或者代码编程实现节点的动态添加或者移除，方便用户扩展和替换节点等；
- XMemcached 通过 JMX 暴露的一些接口，支持 Client 本身的监控和调整，允许动态设置调优参数、查看统计数据、动态增删节点等；
- 支持客户端连接池，对同一个 Memcached 可以创建 N 个连接组成连接池来提高客户端在高并发环境下的表现，而这一切对使用者来说却是透明的；
- 可扩展性，XMemcached 是基于 Java Nio 框架 Yanf4j 实现的，因此在实现上结构相对清楚，分层比较明晰。

快速上手

上面介绍了这么多，最需要关注的是 XMemcached 是最佳的选择，下面我们先用一个示例，来感受一下 Spring Boot 使用 Xmemcached 集成 Memcache。

添加配置

添加依赖包：

```
<dependency>
    <groupId>com.googlecode.xmemcached</groupId>
    <artifactId>xmemcached</artifactId>
    <version>2.4.5</version>
</dependency>
```

添加配置文件：

```
# 单个 Memcached 配置
memcached.servers=192.168.0.161:11211
# 连接池
memcached.poolSize=10
#操作超时时间
memcached.opTimeout=6000
```

配置 Memcached 的地址和端口号、连接池和操作超时时间，使用集群时可以拼接多个地址：**"host1:port1 host2:port2 ..."**。

创建 XMemcachedProperties 类，读配置信息：

```
@Component
@ConfigurationProperties(prefix = "memcached")
public class XMemcachedProperties {
    private String servers;
    private int poolSize;
    private long opTimeout;
    //省略 getter/setter
}
```

启动加载

利用 @Configuration 注解，在启动时对 Memcached 进行初始化。

```
@Configuration
public class MemcachedBuilder {
    protected static Logger logger =  LoggerFactory.getLogger(MemcachedBuilder.cla
ss);
    @Resource
    private XMemcachedProperties xMemcachedProperties;

    @Bean
    public  MemcachedClient getMemcachedClient() {
        MemcachedClient memcachedClient = null;
        try {
            MemcachedClientBuilder builder = new XMemcachedClientBuilder(AddrUtil.
getAddresses(xMemcachedProperties.getServers()));
            builder.setConnectionPoolSize(xMemcachedProperties.getPoolSize());
            builder.setOpTimeout(xMemcachedProperties.getOpTimeout());
            memcachedClient = builder.build();
        } catch (IOException e) {
            logger.error("inint MemcachedClient failed ",e);
        }
        return memcachedClient;
    }
}
```

因为 XMemcachedClient 的创建有比较多的可选项，所以提供了一个 XMemcachedClientBuilder 类用于构建 MemcachedClient。MemcachedClient 是主要接口，操作 Memcached 的主要方法都在这个接口，XMemcachedClient 是它的一个实现。

在方法 getMemcachedClient() 添加 @Bean 注解，代表启动时候将方法构建好的实例注入到 Spring 容器中，后面在需要使用的类中，直接注入 MemcachedClient 即可。

进行测试

我们创建一个 MemcachedTests 类，来测试 Memcached 配置信息是否配置正确。

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MemcachedTests {
    @Autowired
    private MemcachedClient memcachedClient;
}
```

测试 Memcached 的 get 、set 方法。

```
@Test
public void testGetSet() throws Exception {
    memcachedClient.set("hello", 0, "Hello,xmemcached");
    String value = memcachedClient.get("hello");
    System.out.println("hello=" + value);
    memcachedClient.delete("hello");
}
```

存储数据是通过 set 方法，它有三个参数，第一个是存储的 key 名称，第二个是 expire 时间（单位秒），超过这个时间，memcached 将这个数据替换出去，0 表示永久存储（默认是一个月），第三个参数就是实际存储的数据，可以是任意的 Java 可序列化类型。

获取存储的数据是通过 get 方法，传入 key 名称即可；如果要删除存储的数据，可以通过 delete 方法，它也是接受 key 名称作为参数。

执行 testMemcached() 单元测试之后，控制台会输出：

```
hello=Hello,xmemcached
```

证明 Memcached 配置、设置和获取值成功。

XMemcached 语法介绍

XMemcached 有非常丰富的语法来支持，我们对缓存使用的各种场景，接下来一一介绍。

常用操作

除过上面的 get、set、delete 等方法外，Memcache 还有很多常用的操作。

```
@Test
public void testMore() throws Exception {
    if (!memcachedClient.set("hello", 0, "world")) {
        System.err.println("set error");
    }
    if (!memcachedClient.add("hello", 0, "dennis")) {
        System.err.println("Add error,key is existed");
    }
    if (!memcachedClient.replace("hello", 0, "dennis")) {
        System.err.println("replace error");
    }
    memcachedClient.append("hello", " good");
    memcachedClient.prepend("hello", "hello ");
    String name = memcachedClient.get("hello", new StringTranscoder());
    System.out.println(name);
    memcachedClient.deleteWithNoReply("hello");
}
```

- add 命令，用于将 value（数据值）存储在指定的 key（键）中。如果 add 的 key 已经存在，则不会更新数据（过期的 key 会更新），之前的值将仍然保持相同，并且将获得响应 NOT_STORED。
- replace 命令，用于替换已存在的 key（键）的 value（数据值）。如果 key 不存在，则替换失败，并且将获得响应 NOT_STORED。
- append 命令，用于向已存在 key（键）的 value（数据值）后面追加数据。
- prepend 命令，用于向已存在 key（键）的 value（数据值）前面追加数据。
- deleteWithNoReply 方法，这个方法删除数据并且告诉 Memcached，不用返回应答，因此这个方法不会等待应答直接返回，比较适合于批量处理。

Incr 和 Decr

Incr 和 Decr 类似数据的增和减，两个操作类似 Java 中的原子类如 AtomicIntger，用于原子递增或者递减变量数值，示例如下：

```
@Test
public void testIncrDecr() throws Exception {
    memcachedClient.delete("Incr");
    memcachedClient.delete("Decr");
    System.out.println(memcachedClient.incr("Incr", 6, 12));
    System.out.println(memcachedClient.incr("Incr", 3));
    System.out.println(memcachedClient.incr("Incr", 2));
    System.out.println(memcachedClient.decr("Decr", 1, 6));
    System.out.println(memcachedClient.decr("Decr", 2));
}
```

为了防止数据干扰，在测试开始前前调用 delete() 方法清除两个 key 值。

输出：

```
12
15
17
6
4
```

Incr 和 Decr 都有三个参数的方法，第一个参数指定递增的 key 名称，第二个参数指定递增的幅度大小，第三个参数指定当 key 不存在的情况下的初始值，两个参数的重载方法省略了第三个参数，默认指定为 0。

Counter

Xmemcached 还提供了一个称为计数器的封装，它封装了 incr/decr 方法，使用它就可以类似 AtomicLong 那样去操作计数，示例如下：

```
@Test
public void testCounter() throws Exception {
    MemcachedClient memcachedClient = memcachedUtil.getMemcachedClient();
    Counter counter=memcachedClient.getCounter("counter",10);
    System.out.println("counter="+counter.get());
    long c1 =counter.incrementAndGet();
    System.out.println("counter="+c1);
    long c2 =counter.decrementAndGet();
    System.out.println("counter="+c2);
    long c3 =counter.addAndGet(-10);
    System.out.println("counter="+c3);
}
```

- `memcachedClient.getCounter("counter",10)`，第一个参数为计数器的 key，第二参数当 key 不存在时的默认值；
- `counter.incrementAndGet()`，执行一次给计数器加 1；
- `counter.decrementAndGet()`，执行一次给计数器减 1。

查看 counter.addAndGet(-10) 源码（如下），发现 addAndGet() 会根据传入的值的正负来判断，选择直接给对应的 key 加多少或者减多少，底层也是使用了 incr() 和 decr() 方法。

```
public long addAndGet(long delta) throws MemcachedException, InterruptedException,
 TimeoutException {
    return delta >= 0L ? this.memcachedClient.incr(this.key, delta, this.initialVa
lue) : this.memcachedClient.decr(this.key, -delta, this.initialValue);
}
```

Counter 适合在高并发抢购场景下做并发控制。

CAS 操作

Memcached 是通过 CAS 协议实现原子更新，所谓原子更新就是 Compare and Set，原理类似乐观锁，每次请求存储某个数据同时要附带一个 CAS 值，Memcached 比对这个 CAS 值与当前存储数据的 CAS 值是否相等，如果相等就让新的数据覆盖老的数据，如果不相等就认为更新失败，这在并发环境下特别有用。
XMemcached 提供了对 CAS 协议的支持（无论是文本协议还是二进制协议），CAS 协议其实是分为两个步骤：获取 CAS 值和尝试更新，因此一个典型的使用场景如下：

```
GetsResponse<Integer> result = client.gets("a");
long cas = result.getCas();
//尝试将 a 的值更新为 2
if (!client.cas("a", 0, 2, cas)) {
    System.err.println("cas error");
}
```

首先通过 gets 方法获取一个 GetsResponse，此对象包装了存储的数据和 CAS 值，然后通过 CAS 方法尝试原子更新，如果失败打印“cas error”。显然，这样的方式很繁琐，并且如果你想尝试多少次原子更新就需要一个循环来包装这一段代码，因此 XMemcached 提供了一个 *CASOperation* 接口包装了这部分操作，允许你尝试 N 次去原子更新某个 key 存储的数据，无需显式地调用 gets 获取 CAS 值，上面的代码简化为：

```
client.cas("a", 0, new CASOperation<Integer>() {
            public int getMaxTries() {
            return 1;
        }

        public Integer getNewValue(long currentCAS, Integer currentValue) {
                return 2;
        }
});
```

CASOpertion 接口只有两个方法，一个是设置最大尝试次数的 getMaxTries 方法，这里是尝试一次，如果尝试超过这个次数没有更新成功将抛出一个 TimeoutException，如果你想无限尝试（理论上），可以将返回值设定为 Integer.MAX_VALUE；另一个方法是根据当前获得的 GetsResponse 来决定更新数据的 getNewValue 方法，如果更新成功，这个方法返回的值将存储成功，其两个参数是最新一次 gets 返回的 GetsResponse 结果。

设置超时时间

XMemcached 由于是基于 nio，因此通讯过程本身是异步的，client 发送一个请求给 Memcached，你是无法确定 Memcached 什么时候返回这个应答，客户端此时只有等待，因此还有个等待超时的概念在这里。客户端在发送请求后，开始等待应答，如果超过一定时间就认为操作失败，这个等待时间默认是 5 秒，也可以在获取的时候配置超时时间。

```
value=client.get("hello",3000);
```

就是等待 3 秒超时，如果 3 秒超时就跑出 TimeutException，用户需要自己处理这个异常。因为等待是通过

调用 CountDownLatch.await(timeout) 方法，所以用户还需要处理中断异常 InterruptException，最后的 MemcachedException 表示 Xmemcached 内部发生的异常，如解码编码错误、网络断开等异常情况。

更新缓存过期时间

经常有这样的需求，就是希望更新缓存数据的超时时间（expire time），现在 Memcached 已经支持 touch 协议，只需要传递 key 就更新缓存的超时时间：

```
client.touch(key,new-expire-time);
```

有时候你希望获取缓存数据并更新超时时间，这时候可以用 getAndTouch 方法（仅二进制协议支持）：

```
client.getAndTouch(key,new-expire-time);
```

如果在使用过程中报以下错误：

```
Caused by: net.rubyeye.xmemcached.exception.UnknownCommandException: Response erro
r,error message:Unknow command TOUCH,key=Touch
    at net.rubyeye.xmemcached.command.Command.decodeError(Command.java:250)
```

说明安装的 Memcached 服务不支持 touch 命令，建议升级。

测试示例：

```
@Test
public void testTouch() throws Exception {
    memcachedClient.set("Touch", 2, "Touch Value");
    Thread.sleep(1000);
    memcachedClient.touch("Touch",6);
    Thread.sleep(2000);
    String value =memcachedClient.get("Touch",3000);
    System.out.println("Touch=" + value);
}
```

Memcached 集群

Memcached 的分布是通过客户端实现的，客户端根据 key 的哈希值得到将要存储的 Memcached 节点，并将对应的 value 存储到相应的节点。

XMemcached 同样支持客户端的分布策略，默认分布的策略是按照 key 的哈希值模以连接数得到的余数，对应的连接就是将要存储的节点。如果使用默认的分布策略，不需要做任何配置或者编程。

XMemcached 同样支持一致性哈希（Consistent Hash），通过编程设置：

```
MemcachedClientBuilder builder = new XMemcachedClientBuilder(AddrUtil.getAddresses
("server1:11211 server2:11211 server3:11211"));
builder.setSessionLocator(new KetamaMemcachedSessionLocator());
MemcachedClient client=builder.build();
```

XMemcached 还提供了额外的一种哈希算法——选举散列，在某些场景下可以替代一致性哈希：

```
MemcachedClientBuilder builder = new XMemcachedClientBuilder(
                                    AddrUtil.getAddresses("server1:11211 server2:1
1211 server3:11211"));
builder.setSessionLocator(new ElectionMemcachedSessionLocator());
MemcachedClient mc = builder.build();
```

在集群的状态下可以给每个服务设置不同的权重：

```
MemcachedClientBuilder builder = new XMemcachedClientBuilder(AddrUtil.getAddresses
("localhost:12000 localhost:12001"),new int[]{1,3});
MemcachedClient memcachedClient=builder.build();
```

SASL 验证

Memcached 1.4.3 开始支持 SASL 验证客户端，在服务器配置启用 SASL 之后，客户端需要通过授权验证才可以跟 Memcached 继续交互，否则将被拒绝请求，XMemcached 1.2.5 开始支持这个特性。假设 Memcached 设置了 SASL 验证，典型地使用 CRAM-MD 5 或者 PLAIN 的文本用户名和密码的验证机制，假设用户名为 cacheuser，密码为 123456，那么编程的方式如下：

```
MemcachedClientBuilder builder = new XMemcachedClientBuilder(
                AddrUtil.getAddresses("localhost:11211"));
builder.addAuthInfo(AddrUtil.getOneAddress("localhost:11211"), AuthInfo
                .typical("cacheuser", "123456"));
// Must use binary protocol
builder.setCommandFactory(new BinaryCommandFactory());
MemcachedClient client=builder.build();
```

请注意，授权验证仅支持二进制协议。

查看统计信息

Memcached 提供了统计协议用于查看统计信息：

```
Map<InetSocketAddress,Map<String,String>> result=client.getStats();
```

getStats 方法返回一个 map ，其中存储了所有已经连接并且有效的 Memcached 节点返回的统计信息，你也可以统计具体的项目，如统计 items 项目：

```
Map<InetSocketAddress,Map<String,String>> result=client.getStatsByItem("items");
```

只要向 getStatsByItem 传入需要统计的项目名称即可，我们可以利用这个功能，来做 Memcached 状态监控等。

总结

Memcached 是一款非常流行的缓存中间件，被广泛应用在各场景中，使用缓存可以环境数据库压力，某些场景下使用缓存可以大大提高复用的 Tps 。 XMemcached 是 Memcached 的一个高性能 Nio 客户端，支持 Memcached 底层各种操作，并且在 Memcached 协议的基础上进行了封装和完善，提供了连接池、集群、数据压缩、分布式算法等高级功能，不论是完善度和性能各方面来看，XMemcached 都是目前最为推荐的一款 Memcached 客户端。

> 点击这里下载源码。