EVCache

EVCache介绍

EVCache是一个开源、快速的分布式缓存

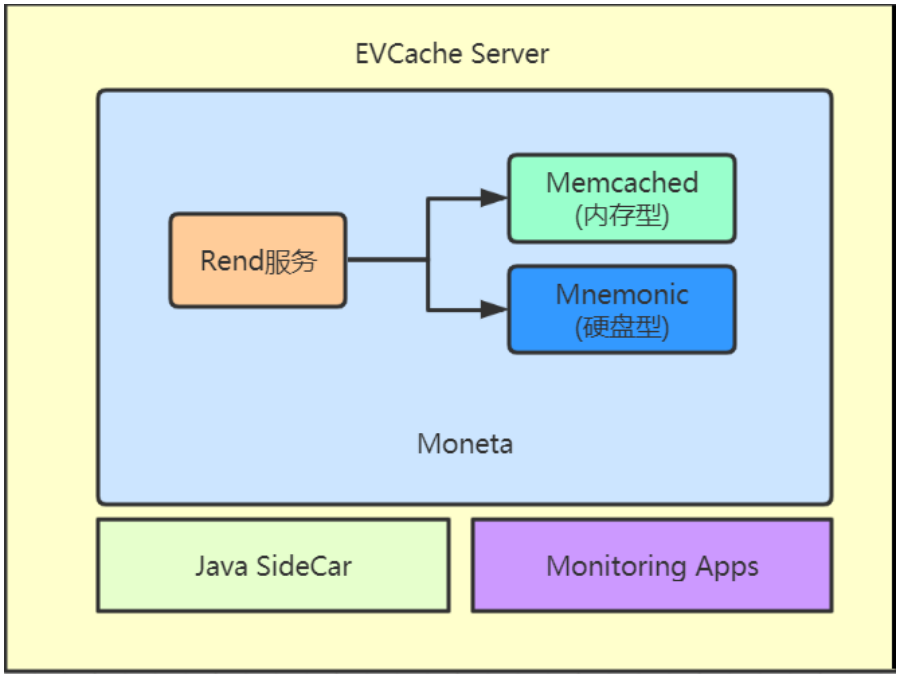
是基于Memcached的内存存储和Spymemcached客户端实现的

是Netflix（网飞）公司开发的

E：Ephemeral：数据存储是短暂的，有自身的存活时间

V：Volatile：数据可以在任何时候消失

Cache：内存级键值对存储



Rend服务：是一个代理服务，用GO语言编写，能够高性能的处理并发。

Memcached：基于内存的键值对缓存服务器

Mnemonic：基于硬盘（SSD）的嵌入式键值对存储服务器，封装了RocksDB(是一种SSD的技术)

EVCache集群在峰值每秒可以处理200kb的请求，

Netflix生产系统中部署的EVCache经常要处理超过每秒3000万个请求，存储数十亿个对象，

跨数千台memcached服务器。整个EVCache集群每天处理近2万亿个请求。

EVCache集群响应平均延时大约是1-5毫秒，最多不会超过20毫秒。

EVCache集群的缓存命中率在99%左右。

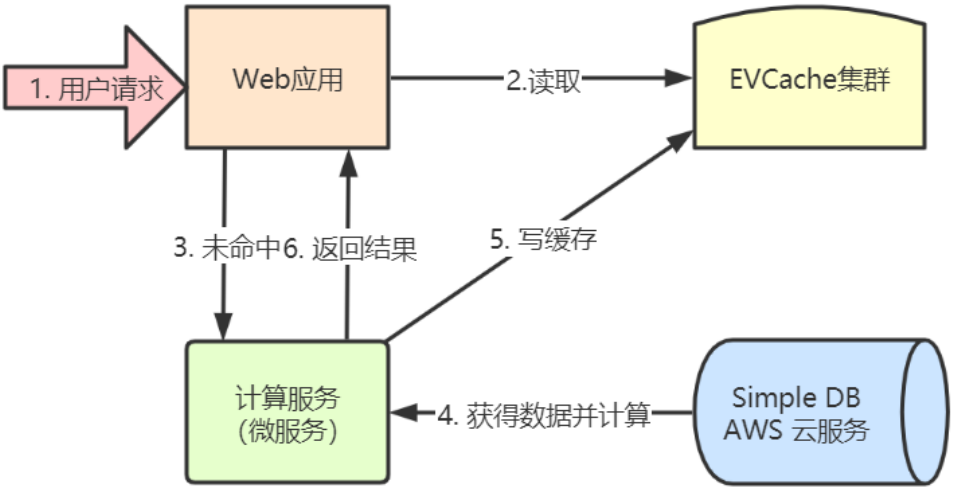
EVCache的应用

典型用例

Netflix用来构建超大容量、高性能、低延时、跨区域的全球可用的缓存数据层

EVCache典型地适合对强一致性没有必须要求的场合

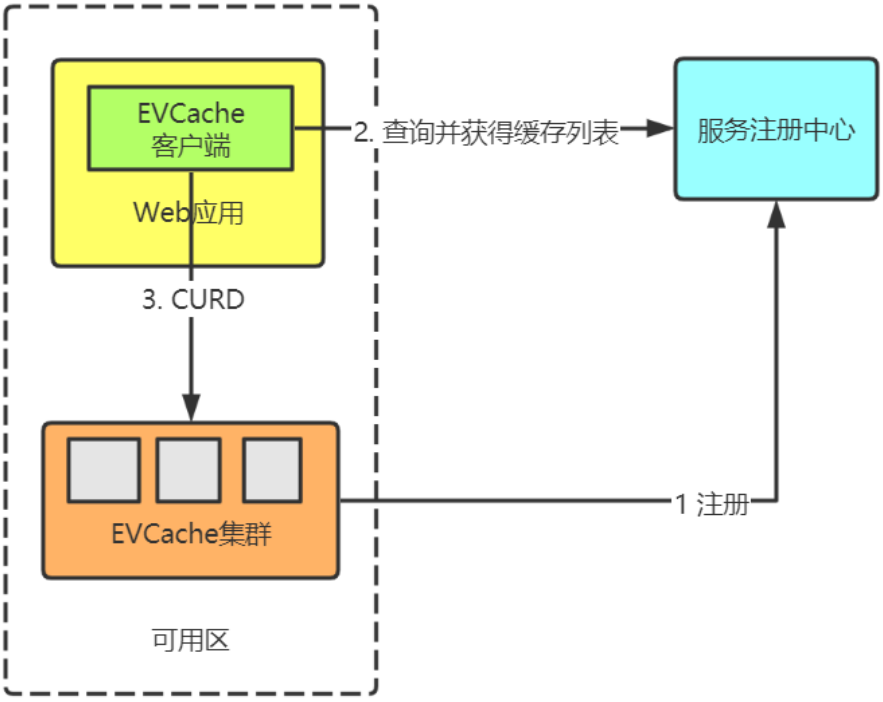
典型用例：Netflix向用户推荐用户感兴趣的电影



典型部署

EVCache 是线性扩展的，可以在一分钟之内完成扩容，在几分钟之内完成负载均衡和缓存预热。

单节点部署

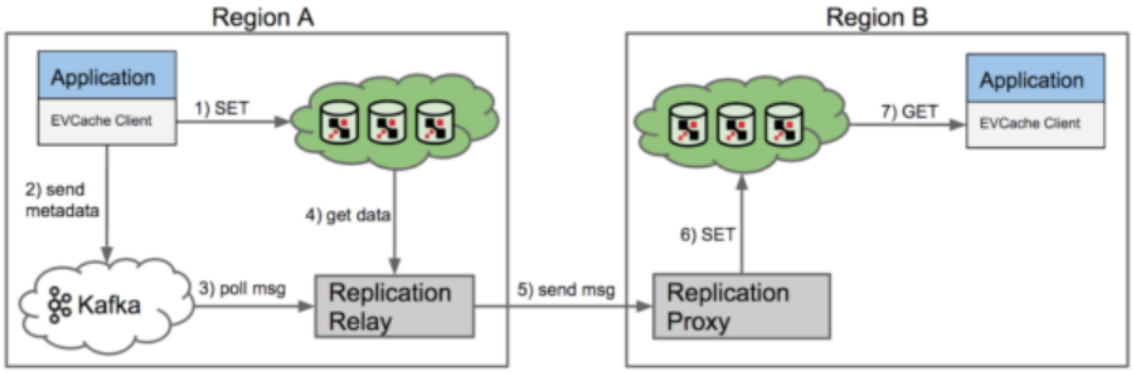


1、集群启动时，EVCache向服务注册中心（Zookeeper、Eureka）注册各个实例

2、在web应用启动时，通过EVCache的客户端(EVCache Client)查询命名服务中的EVCache服务器列表，并建立连接。

3、web应用通过EVCache的客户端操作数据，客户端通过key使用一致性hash算法，将数据分片到集群上。

多可用区部署



EVCache的跨可用区复制

1.EVCache客户端库包发送SET到缓存系统的本地地区的一个实例服务器中。

2.EVCache客户端库包同时也将写入元数据(包括key，但是不包括要缓存的数据本身)到复制消息队列(Kafka)

3.本地区的“复制转播”的服务将会从这个消息队列中读取消息。

4.转播服务会从本地缓存中抓取符合key的数据

5.转播会发送一个SET请求到远地区的“复制代理”服务。

6.在远地区，复制代理服务会接受到请求，然后执行一个SET到它的本地缓存，完成复制。

7.在接受地区的本地应用当通过GET操作以后会在本地缓存看到这个已经更新的数据值。

EVCache的安装与使用

由于Netflix没有开源EVCache的服务器部分，这里采用Memcached作为服务器。

安装Memcached

```
#安装libevent库
yum install libevent libevent-devel gcc-c++
#下载最新的memcached
wget http://memcached.org/latest
#解压
tar -zxvf latest
#进入目录
cd memcached-1.6.6
#配置
./configure --prefix=/usr/memcached
#编译
make
#安装
make install
#启动
memcached  -d -m 1000 -u root -l 192.168.127.131 -p 11211 -c 256 -P
/tmp/memcached.pid
-d         选项是启动一个守护进程
-m        是分配给Memcache使用的内存数量，单位是MB，我这里是10MB
```

```
-u         是运行Memcache的用户，我这里是root
-l          是监听的服务器IP地址，我这里指定了服务器的IP地址192.168.127.131
-p         是设置Memcache监听的端口，我这里设置了11211（默认），最好是1024以上的端口
-c         选项是最大运行的并发连接数，默认是1024，我这里设置了256，按照你服务器的负载量来设定
-P         是设置保存Memcache的pid文件，我这里是保存在 /tmp/memcached.pid
```

使用EVCache Client

1、pom

```
<dependency>
            <groupId>com.netflix.evcache</groupId>
            <artifactId>evcache-client</artifactId>
            <version>4.139.0</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/net.spy/spymemcached -->
        <dependency>
            <groupId>net.spy</groupId>
            <artifactId>spymemcached</artifactId>
            <version>2.12.3</version>
        </dependency>


        <dependency>
            <groupId>com.netflix.eureka</groupId>
            <artifactId>eureka-client</artifactId>
            <version>1.5.6</version>
            <scope>runtime</scope>
        </dependency>

        <dependency>
            <groupId>com.netflix.spectator</groupId>
            <artifactId>spectator-nflx-plugin</artifactId>
            <version>0.80.1</version>
            <scope>runtime</scope>
        </dependency>

        <dependency>
            <groupId>com.netflix.spectator</groupId>
            <artifactId>spectator-api</artifactId>
            <version>0.80.1</version>
        </dependency>


        <dependency>
            <groupId>com.netflix.rxjava</groupId>
            <artifactId>rxjava-core</artifactId>
            <version>0.20.7</version>
        </dependency>

        <dependency>
            <groupId>com.netflix.servo</groupId>
            <artifactId>servo-core</artifactId>
            <version>0.12.25</version>
        </dependency>

        <dependency>
```

```
        <groupId>com.google.code.findbugs</groupId>
        <artifactId>annotations</artifactId>
        <version>3.0.1</version>
    </dependency>


    <dependency>
        <groupId>com.netflix.nebula</groupId>
        <artifactId>nebula-core</artifactId>
        <version>4.0.1</version>
        <scope>runtime</scope>
    </dependency>

    <dependency>
        <groupId>com.netflix.archaius</groupId>
        <artifactId>archaius2-core</artifactId>
        <version>2.3.13</version>
        <scope>runtime</scope>
    </dependency>

    <dependency>
        <groupId>com.netflix.archaius</groupId>
        <artifactId>archaius-aws</artifactId>
        <version>0.6.0</version>
    </dependency>

    <dependency>
        <groupId>javax.inject</groupId>
        <artifactId>javax.inject</artifactId>
        <version>1</version>
    </dependency>

    <dependency>
        <groupId>io.reactivex</groupId>
        <artifactId>rxjava</artifactId>
        <version>1.3.8</version>
    </dependency>
```

2、编写EVCache代码

```
    String deploymentDescriptor = System.getenv("EVCACHE_SERVER");
    if ( deploymentDescriptor == null ) {
        deploymentDescriptor = "SERVERGROUP1=192.168.127.131:11211";
    }
    System.setProperty("EVCACHE_1.use.simple.node.list.provider", "true");
    System.setProperty("EVCACHE_1-NODES", deploymentDescriptor);
    EVCache evCache = new EVCache.Builder().setAppName("EVCACHE_1").build();
    // s:key  t :value i:ttl
    evCache.set("name","zhangfei",10);
    String v = evCache.get("name");
    System.out.println(v);
```

EVCache原理

EVCache的内存存储是基于Memcached实现的

EVCache的客户端是基于Spymemcached实现的

Memcached内存存储

Memcached简介

Memcached是danga（丹加）开发的一套分布式内存对象缓存系统，用于在动态系统中减少数据库负载，提升性能

Memcached是C/S模式的

基于libevent的事件处理

Libevent 是一个用C语言开发的，高性能;轻量级，专注于网络，不如 ACE 那么臃肿庞大；源代码相当精炼、易读；跨平台，支持 Windows、 Linux、 BSD 和 Mac OS；支持多种 I/O 多路复用技术，epoll、 poll、 select 和 kqueue 等；支持 I/O，定时器和信号等事件；注册事件优先级。

Memcached是多线程的

	Memcached	**Redis**
多线程	是	否
数据结构	简单kv，V为一个值	V多种，List、Set、Hash....
持久化	否	是
集群	客户端或代理端（twemproxy）	客户端、代理端(codis)、服务器端(RedisCluster)
性能	读多写少,大数据 大于100K	读写都行
通信协议	文本（telnet）、二进制	RESP
集群内通信	无	Gossip、订阅
数据安全	无	有
集群failover	无	有

Slab Allocation机制

传统的内存分配是通过对所有记录简单地进行malloc（动态内存分配）和free（释放） 来进行的。是以Page（M）为存储单位的(BuddySystem)。

这种方式会导致内存碎片，加重操作系统内存管理器的负担。

memcached采用Slab（块） Allocation的方式分配和管理内存

slab是Linux操作系统的一种内存分配机制

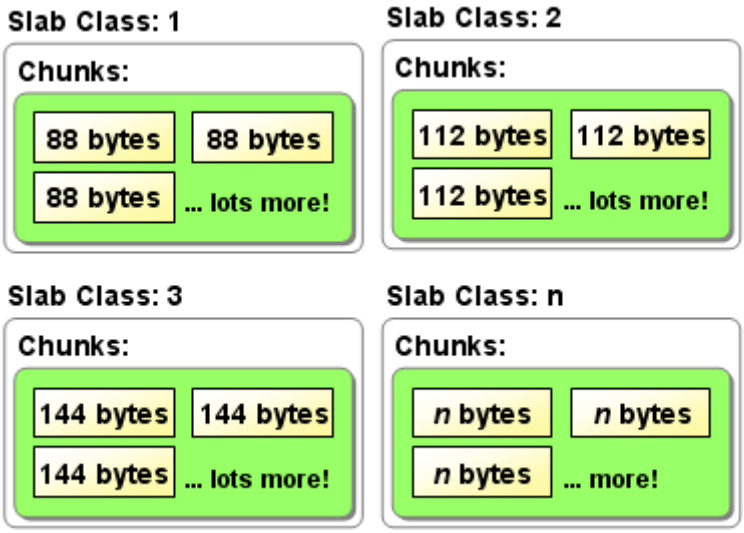
slab分配器分配内存以Byte为单位，专为小内存分配而生

Slab Allocation的原理：

根据预先设定的大小(Page=1M)， memcached -m 分配的内存 默认64M

将分配的内存分割成各种大小的块(Chunk)，

并把尺寸相同的块分成组(Slab Class)，Memcached根据收到的数据大小，选择最合适的slabClass进行存储



注：块越小存的数据越多，块越来越大，是由growth factor决定的(1.25)

数据Item

Item就是我们要存储的数据。是以双向链表的形式存储的。

```
/**
 * Structure for storing items within memcached.
 */
typedef struct _stritem {
    struct _stritem *next;  //next即后向指针
    struct _stritem *prev;  //prev为前向指针
    struct _stritem *h_next;    /* hash chain next */
    rel_time_t      time;       /* least recent access */
    rel_time_t      exptime;    /* expire time */
    int             nbytes;     /* size of data */
    unsigned short  refcount;
    uint8_t         nsuffix;    /* length of flags-and-length string */
    uint8_t         it_flags;   /* ITEM_* above */
    uint8_t         slabs_clsid;/* which slab class we're in */
    uint8_t         nkey;       /* key length, w/terminating null and padding */

    union {
        uint64_t cas;
        char end;
    } data[];

} item;
```

item 的结构分两部分:

item结构定义next、prev、time(最近访问时间)、exptime（过期的时间）、nkey（key的长度）、refcount（引用次数）、nbytes（数据大小）、slabs_clsid（从哪个 slabclass 分配而来）

item数据： CAS, key, suffix, value 组成

缓存过期机制

Memcached有两种过期机制：Lazy Expiration（惰性过期）和LRU

Lazy Expiration

Memcached在get数据时，会查看exptime，根据当前时间计算是否过期(now-exptime>0)，如果过期则删除该数据

LRU

当Memcached使用内存大于设置的最大内存(-m 启动指定 默认64M)使用时，Memcached会启动LRU算法淘汰旧的数据项。

使用slabs_alloc函数申请内存失败时，就开始淘汰数据了。

淘汰规则是，从数据项列表尾部开始遍历，在列表中查找一个引用计数器(refcount)为0的item，把此item释放掉。

如果在item列表找不到计数器为0的item，就查找一个3小时没有访问过的item(now-time>3H)。把他释放，如果还是找不到，就返回NULL（申请内存失败）。

当内存不足时，memcached会把访问比较少或者一段时间没有访问的item淘汰，以便腾出内存空间存放新的item。

Spymemcached设计思想

spymemcached 是一个 memcached的客户端， 使用NIO实现。

memcachedclient　danga

spymemcached　 spy

xmemcached

主要有以下特性：

Memcached协议支持Text和Binary(二进制)

异步通信：使用NIO，采用callback

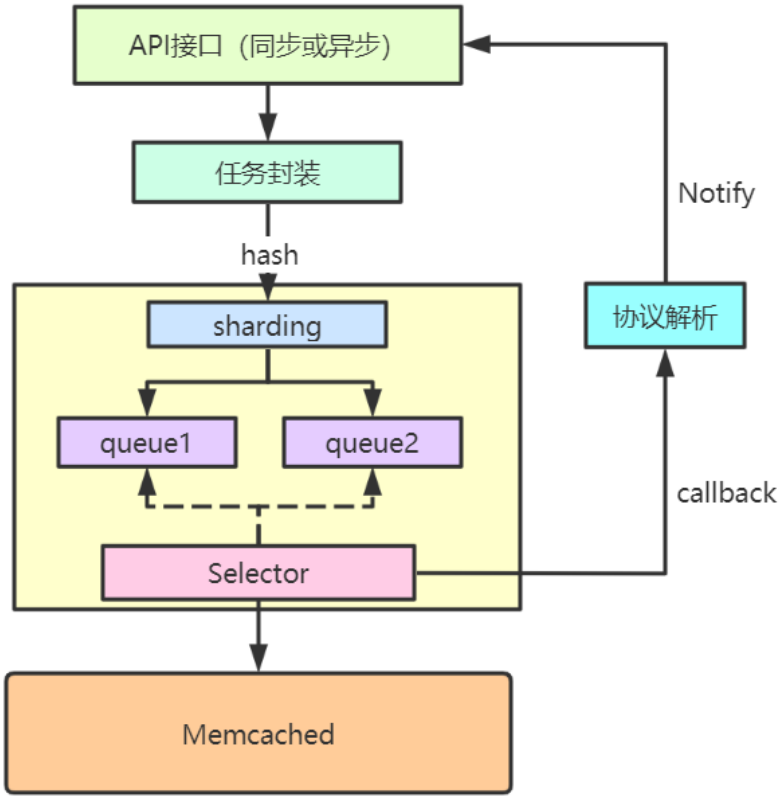
集群：支持sharding机制

自动恢复：断网重连

failover：可扩展容错，支持故障转移

支持批量get、支持jdk序列化

整体设计



1. API接口：提供同步或异步接口调用，异步接口返回Future
2. 任务封装：将访问的操作及callback封装成Task
3. 路由分区：通过Sharding策略（后面讲），选择Key对应的连接（connection）
4. 将Task放到对应连接的队列中
5. Selector异步获取队列中的Task，进行协议的封装和发送相应Memcached
6. 收到Memcached返回的包，会找到对应的Task，回调callback，进行协议解析并通知业务线程Future.get

API接口设计

对外API接口有两种，同步和异步

同步接口：比如set、get等

异步接口：asyncGet、asyncIncr等

```
//获取一个连接到几个服务端的memcached的客户端
MemcachedClient c = new MemcachedClient(AddrUtil.getAddresses("192.168.127.123:1
1211"));
//获取值，如果在5秒内没有返回值，将取消
Object myObj = null;
Future<Object> f = c.asyncGet("name");
try{
    //异步获取 阻塞
    myObj = f.get(5,TimeUnit.SECONDS);
}catch(TimeoutException e){
    //退出
    f.cancel(false);
}
```

线程设计

SpyMemcached有两类线程：业务线程和selector线程

业务线程：

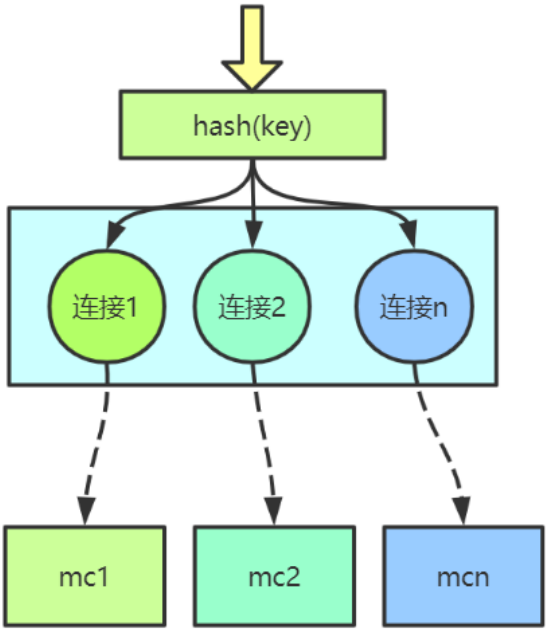
- 封装请求task、对象序列化、封装发送的协议、并将task放到对应连接的队列中
- 对收到的数据进行反序列化为对象

Selector线程

- 读取连接中的队列，将队列中的task的数据发送到memcached
- 读取Memcached返回的数据，解析协议并通知业务线程处理
- 对失败的节点进行自动重连

sharding机制

路由机制



Spymemcached默认的hash算法有：

arrayMod：传统hash 模节点取余

hash(key)%节点数 --->余数　扩容、缩容 做数据迁移

ketama：一致性hash（推荐） 扩容、缩容好些

容错

key路由到服务节点，服务节点宕机，有两种方式处理：

自动重连

失败的节点从正常的队列中摘除

添加到重连队列中

Selector线程定期对该节点进行重连

failover处理

- Redistribute(推荐)

  遇到失败节点后选择下一个节点，直到选到正常的节点

  大量回源

- Retry

  继续访问，一般就失败了

- Cancel

  抛异常退出访问

Memcache的集群处理容错比较差

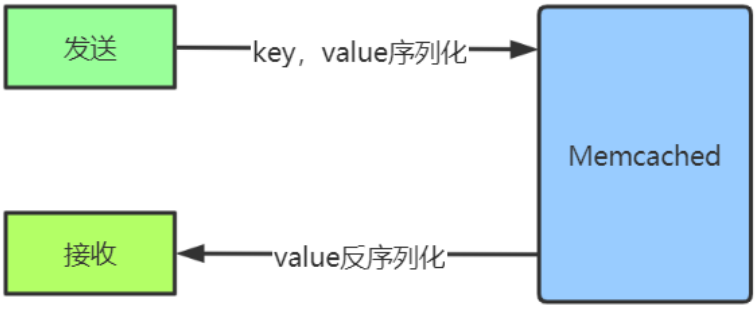
序列化

我们知道在日常存储中数据往往是对象，那么在Memcached中会以二进制的方式存储

所以在存数据时要进行对象的序列化（jdk序列化）

在取数据时要进行对象的反序列化

是业务线程处理



序列化后会判断长度是否大于阈值16384个byte，如果大于将采用Gzip的方式进行数据压缩