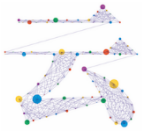

eMem

弹性云缓存系统（The cloud cache system based on redis/memcached）

2014 全国高校云计算应用创新大赛

项目总结文档

eMem——弹性云缓存系统

上海交通大学 eMem 团队

2014-12

文档目录

一、项目介绍

缓存系统广泛应用于构建大规模 Web 的应用中，其允许 Web 应用从快速的内存缓存系统中检索信息，而无需完全依赖于速度较慢的基于磁盘的数据库，从而提高了 Web 应用的性能。目前常见的两种开源缓存引擎包括 Memcached 和 Redis。

Memcached 是一种被广泛采用的内存对象缓存系统。Memcached 基于一个存储键/值对的 hashmap。其守护进程（daemon ）是采用 C 编写，但是客户端可以用任何语言来编写，并通过 memcached 协议与守护进程通信。

Redis 是一种常用的开源内存键值存储系统，可支持有序集合和列表等数据结构。和 Memcached 类似，它支持存储的 value 类型相对更多，包括 string(字符串)、list(链表)、set(集合)、zset(sorted set --有序集合)和 hash（哈希类型）。这些数据类型都支持 push/pop、add/remove 及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。redis 的出现，很大程度补偿了 memcached 单一 key/value 存储的不足，在部分场合可以对关系数据库起到很好的补充作用。

由于能极大的提高业务系统的存取性能，缓存系统受到业界的越来越广泛的关注。

但是，使用缓存系统也是有一定的代价的，由于其对内存资源的硬性要求较高，使得单独配备缓存服务器的成本偏高；并且由于不用缓存引擎使用了不同的协议规范，使得在同一个业务系统向前向后的兼容性比较差。

eMem 弹性云缓存系统就是针对这一问题而设计开发的,eMem 系统向用户提供提供统一的缓存服务接口，提供按需分配的缓存资源，用户可以即时申请即时使用。用户在享用缓存高效性能的同时，不再会为缓存服务器部署和维护所拖累，能忽略不同缓存引擎的差异性，把精力更多的投入在业务系统的开发当中。

同时,eMem 系统提供了良好的运维保护,支持缓存引擎(Redis)的统一管理，能够根据实际使用情况进行在线动态扩展，并能够实时对缓存服务监控和预警，提供缓存使用情况的分析报告。使得运维人员能一键式操控整个分布式缓存系统。

总结 eMem 弹性云缓存系统的特点就是方便、快捷、高效。

二、需求整合

目前我们已经完美实现了所有必选需求和大部分可选需求，并在此基础上完成了额外的一些功能：

需求名称	是否必选	是否实现	备注
缓存服务子系统	是	√	HTTP、RMI 协议
缓存管理子系统	是	√	提供 Web 界面
缓存配置	是	√	一键式操作
缓存监控	否	√	缓存监控的 4 个模块中，第 4 个模块还未调试完全.
缓存统计分析	否	√	统计分析的算法和策略在进一步优化
客户端	是	√	发布 JAR 包、Javadoc.
缓存资源自动调整	否	√	使用并优化 ArrayList 伸缩算法
缓存接口支持 HTTP 协议	是	√	
缓存接口支持 RMI 协议	否	√	
支持 Redis 引擎	是	√	
支持 Memcached 引擎	否	×	
平均读写时间	是	√	平均读写时间： HTTP 协议<5ms， RMI 协议<3ms 局域网内测试

三、设计思路

依照系统的需求文档，系统可大致分解为三个部分：客户端、缓存服务子系统、缓存管理子系统。整个系统的大致思路可以简化为图 1：

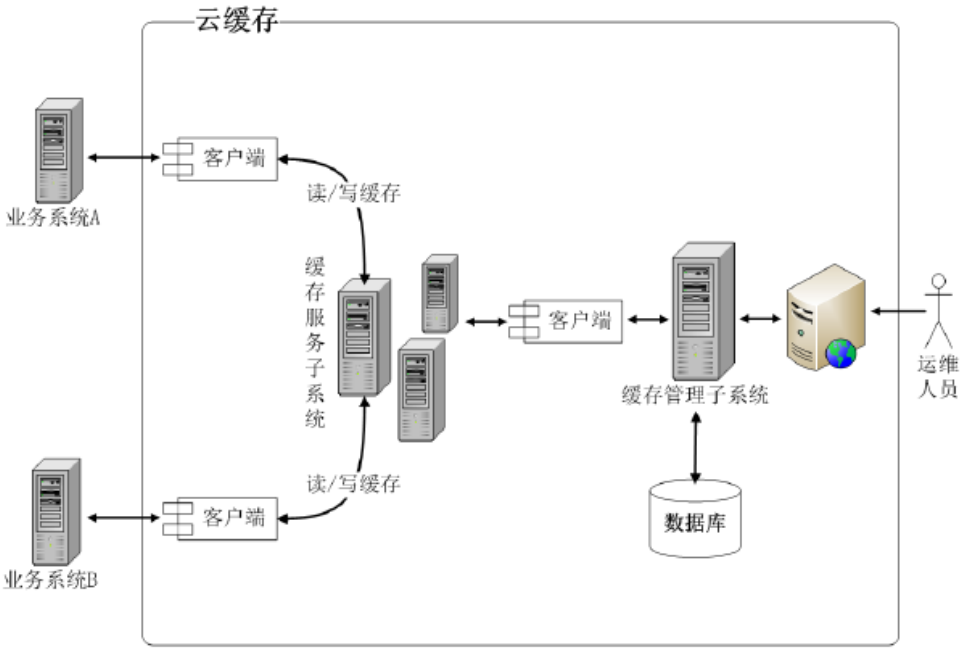

图 1　弹性云缓存系统架构（摘自需求文档）

1. 客户端：

客户端主要负责完成各业务系统和云缓存系统的集成，提供了一组标准的缓存数据的操作接口(JAR 包)。业务系统只需在项目内引入客户端 JAR 包，再稍作配置，就可以轻松使用云缓存提供的缓存服务。

2. 缓存服务子系统：

缓存服务子系统直接依赖 Redis/Memcached 实现，通过指定的通讯协议将一组标准的缓存数据操作接口暴露给用户，使用户能够方便的对缓存数据进行操作。

3. 缓存管理子系统：

缓存管理子系统是运维人员/开发人员用以配置、管理、监控和分析缓存服务的一个管控平台，是“弹性云缓存”的核心系统。系统主要包括缓存配置、缓存监控和缓存统计分析三大核心功能模块。

在 eMem 项目设计之时，我们考虑到各个模块之间的耦合程度比较低，因此在设计阶段就预先定义好各个模块之间的接口，再对各个模块并行开发，来提高整个项目的开发效率，事实证明这个思路非常正确切有效,是我们能在 1 个月的时间内完成大部分需求的主要原因。

我们把整个系统进行了进一步细化，把整个系统的内部需要连接的模块进一步概括为：

- 客户端 - 缓存服务子系统
- 缓存服务子系统 - 缓存引擎
- 缓存引擎 - 缓存服务器
- 缓存服务器 - 缓存管理子系统
- 缓存管理子系统 - 运维 Web 界面

eMem 项目团队总共有 4 个人，根据个人的擅长进行了分工，每个人负责自己对应的模块，同时遵循设计初定义好的规范接口进行编程。

四、项目架构

整个系统分为四层，其架构图见图二：

- 系统层
- 数据层
- 接口层
- 应用层

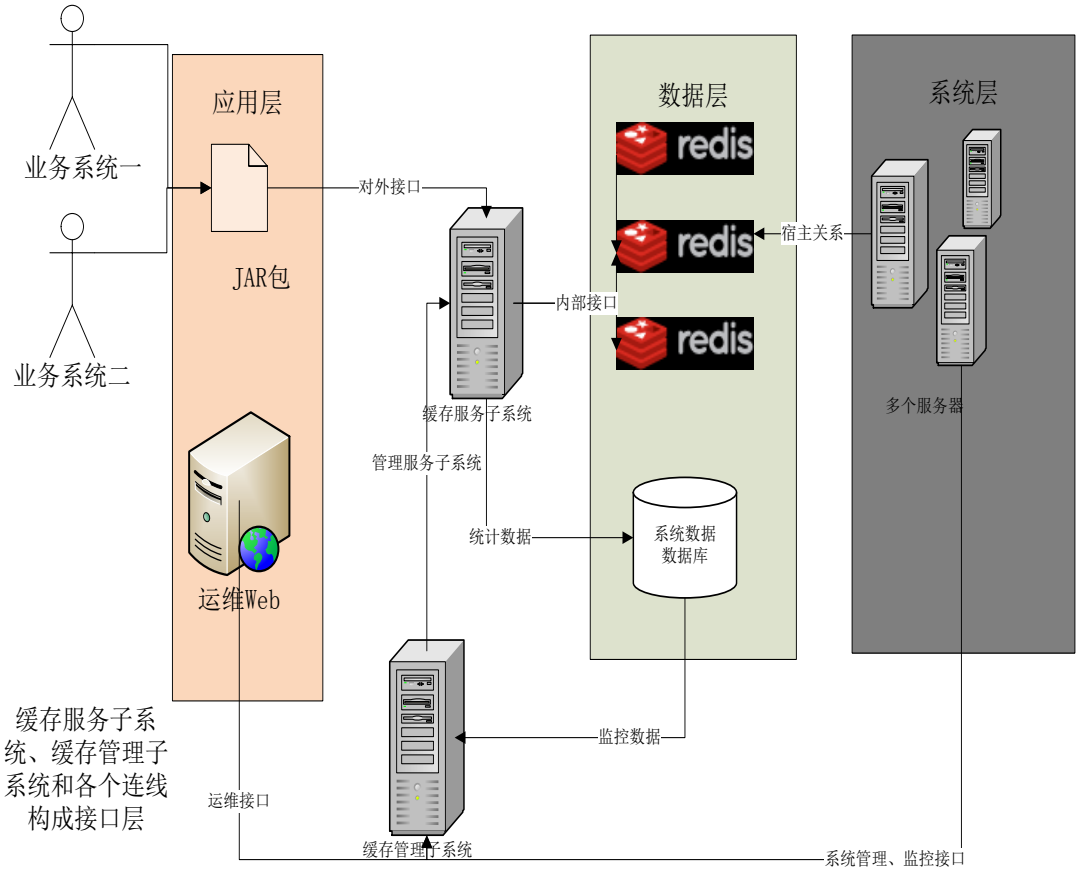

图 2 ： eMem 系统架构图

1. **系统层：**

系统层是整个缓存系统的基础。

不管是否通过虚拟化的技术实现，缓存引擎都是需要运行在硬件平台和操作系统之上的。每个缓存服务器都是系统的的一个缓存节点，我们把缓存节点组成的集群概括为系统层。

由于项目需求和硬件条件的约束，我们通过虚拟化的技术来模拟集群的搭建，创建了众多的虚拟机充当缓存节点。

2. **数据层：**

包括所有 Redis 实例和 eMem 系统自己的数据库。

之所以把这两部分合并为数据层，是因为这两部分在系统中的作用都是充当数据的存取载体，Redis 实例用于存取 eMem 客户数据的载体，系统数据库用于存取 eMem 系统的运维数据。

3. **接口层：**

整个系统中最庞大复杂的一个模块，也是花费时间精力最多的一个模块。

包括各个模块连接的接口，这些接口包括：

- 客户端 - 缓存服务子系统 （HTTP 协议）
- 缓存服务子系统 - 缓存引擎 （Jedis 客户端）
- 缓存管理子系统 - 缓存服务子系统 （HTTP 协议）
- 缓存引擎 - 缓存服务器 （Shell Script）
- 缓存服务器 - 缓存管理子系统 (Socket 编程)
- 缓存管理子系统 - 运维 Web 界面 (HTTP 协议)

4. **应用层：**

应用层是在已经搭建好的分布式系统上提供的对外服务。包括提供给业务系统的 API 和提供给公司运维人员的管控界面。这两个模块属于 eMem 系统的上层应用，比较注重用户体验，他们的是 eMem 系统对外的应用产品。

五、 产品实现

根据上文中的架构，我们分为四层来介绍我们的产品实现过程。

1. 系统层

利用实验室的便利条件，我们在两台高性能服务器上启动了若干虚拟机作为缓存节点，每个缓存节点上运行了一个我们称之为 NodeManager 的守护进程。

图 3 ：缓存节点的物理机
（VM 配置 1 Core(CPU)/4G(Mem)/Ubuntu 12.04(OS)）

NodeManager 需要进行如下工作：

1) 维护本节点的 redis 引擎的相关信息，提供操作当前节点上 redis 引擎的接口。比如启动一个 redis、停止一个 redis、重启一个 redis、删除一个 redis 等，同时要定期把运行在本节点的 redis 信息向 eMem 缓存管理子系统进行汇报。
2) 对缓存节点的监控，缓存节点需要定期向缓存管理子系统汇报心跳，这个监控信息是缓存节点系统运行信息，包括 CPU、内存、IO、网络的状况。

由于时间有限，NodeManager 的实现是基于 Linux 的 Crontab，使用 Shell Socket 编写了一套节点端的 NodeManager 程序，定期向管理子系统通信汇报实例信息和节点信息。

2. 数据层

数据层分为两层：与 eMem 系统用户相关的 Redis，eMem 自身系统相关的数据库 Mongo。

- 缓存引擎（Redis）

  缓存引擎目前只能支持 Redis，eMem 系统的缓存引擎分配策略是为每个用户分配一个 Redis 引擎。

  Redis 引擎的运行载体是 eMem 系统中的缓存节点，每个缓存节点上可能运行着若干个 Redis，每个 Redis 的配置信息交由该 Redis 所在的 NodeManager 来维护和管理。

  缓存服务子系统通过 Redis 的 Java 客户端 Jedis 进行的存取操作。

  NodeManager 通过 Shell 运行 redis-cli 来配置和操作本节点的 Redis 实例。

- 系统数据库（Mongo）

eMem 系统数据库采用的是 MongoDB，MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。他支持的数据结构非常松散，是类似 json 的 bson 格式，因此可以存储比较复杂的数据类型。Mongo 最大的特点是他支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

之所以采取 Mongo 是因为它灵活，快速，跟 redis 的 key-value 理念相对关系型数据库来说更加契合。

下图是我们整个缓存系统的数据库结构。

```
> show dbs
emem_system   0.078GB
local         0.078GB
node_info     0.078GB
redis_info    0.203GB
stat          0.078GB
top_info      0.078GB
>
```

图 4 ：数据库结构图

Mongo 中包含了整个系统的相关数据。

- 缓存服务子系统会把使用缓存服务的所有操作记录到数据库中。操作记录存在 stat db 中.
- 缓存管理子系统会把 Redis 实例和缓存节点的监控信息记录到数据库中。
- 节点表和实例表存在 emem_system 中，节点的监控历史记录和实例的监控历史记录分别存在 node_info 和 redis_info 中。

通过数据库存储的数据，运维管理系统能获取到整个系统的数据，并通过人性化的方式整合成相应的图表，供运维人员查看。

并且这些数据信息更大的意义在于它们是进行弹性云缓存、负载均衡等策略的条件。

3. 接口层

- 缓存服务子系统 （Groovy + Jetty + Jedis）

缓存服务子系统对外与 eMem 的 JAR 客户端程序交互。每个用户分配一个唯一的标识符定义为 token，当用户调用 JAR 包中的接口时，用户的 token 将会随之发送到

并且缓存服务子系统也需要对每个使用我们缓存服务的用户进行权限的认证，这里的实现方案我们采用的是每个用户分配一个 token，用户在使用 eMem 提供的 JAR 包使用缓存服务时每次请求都会将 token 随请求一起发送到缓存服务子系统，再通过缓存服务子系统进行分发。

缓存服务子系统采用的 Groovy + Jetty + Jedis 的架构，服务的接口协议目前支持 HTTP 和 RMI，利用 Jetty 监听并收取客户端发送来得消息，Groovy 编写的 web 服务器端逻辑，最后使用 Redis 客户端 Jedis 完成业务系统的数据操作（get 或 set）。

缓存服务子系统的接口见**附录 A**。

- 缓存管理子系统 (Java + Shell + PHP)

缓存管理子系统是一个耦合度相对比较高的一个模块。

首先缓存管理子系统需要和所有的 NodeManager 通信，需要解析 NodeManager 发送的节点监控信息和 Redis 实例的监控信息，并且将节点管

理的操作和 Redis 实例的操作发送到相应的 NodeManager 上。

其次，缓存管理子系统还需要提供一套管理配置接口，提供给运维系统调用来。

并且，缓存管理子系统还需要和缓存服务子系统通信，每当新建一个 Redis 实例，需要通知缓存服务子系统来保存这一实例的配置信息，方便缓存服务子系统分发用户的请求。

在缓存管理子系统的实现过程中我们进行了仔细的斟酌。

为了满足高并发的服务器，采用 Java NIO 实现了监听服务 Server 端，监听 8189 和 8190 端口。用于解析 NodeManager 发送的节点监控信息和 Redis 实例信息，提取出有意义的数据存入到系统数据库中。

运维管理的管理监控，是利用 Apache httpd + PHP + Shell 实现的。

在缓存管理子系统中我们还实现了缓存资源负载均衡和缓存资源的动态调整，策略如下：

- **缓存资源负载均衡策略：**

  当运维系统触发新建 Redis 实例的请求时，根据当前各个缓存节点的内存和 CPU 状态，eMem 系统将会优先选取一个最优的节点（可用内存最多）新建一个 Redis 实例，并返回对应的 ip 和端口。

- **缓存资源动态调整策略：**

  为了最大限度的利用系统的缓存资源，eMem 为 Redis 分配的实际内存和用户申请的配额是有差别的。

  比如说用户申请了 2G 的内存空间，在初始阶段，为了最大效率的利用系统资源，系统给用户新建的 Redis 实例的内存大小是小于 2G 的，（初始的内存大小是 256M），只有当用户的使用量上来以后，eMem 系统会动态的为用户的 Redis 更改内存配额。

  实现是通过修改 Redis 实例的参数 maxmemory 实现的。

  扩容或缩容的算法我们参考了 ArrayList 的伸缩算法。即当用户的当前的内存使用率超过一半则扩容（小于等于最大配额），如果当前用户的内存使用率小于 1/4 则缩容（大于等于最小配额）。

  为了避免在临界值附近的波动造成多次重复操作，我们为扩容或缩

容的条件加入了阈值的概念，即只有当连续多次监控中都发现已经满足了扩容或缩容的条件我们才进行相应的操作，这样有效避免了在临界点附近的反复操作的情况。

4. 应用层

- 缓存服务客户端：也就是我们提供给客户的 JAR 包，客户通过一套由我们定义的 API 可以操作数据。
- 运维管理 Web 界面：使用 HTML5+CSS3+JavaScript 进行 web 开发，分别采用了以下前端框架与类库，并通过 Ajax 与管理子系统交互。
  - BootStrap: Bootstrap 是 Twitter 推出的一个用于前端开发的开源工具包。它由 Twitter 的设计师 Mark Otto 和 Jacob Thornton 合作开发,是一个 CSS/HTML 框架
  - Less: LESS 是动态的样式表语言,通过简洁明了的语法定义,使编写 CSS 的工作变得非常简单
  - jQuery: Jquery 是继 prototype 之后又一个优秀的 Javascript 库。它是轻量级的 js 库 ，它兼容 CSS3，还兼容各种浏览器
  - KendoUI: Kendo UI 是一个强大的框架用于快速 HTML5 UI 开发，系统中所有的图表均使用 Kendo 进行开发实现

界面示意图：

下图是运维管理界面的登陆界面：

图 5 ：运维管理登录界面

下图是缓存管理的界面：

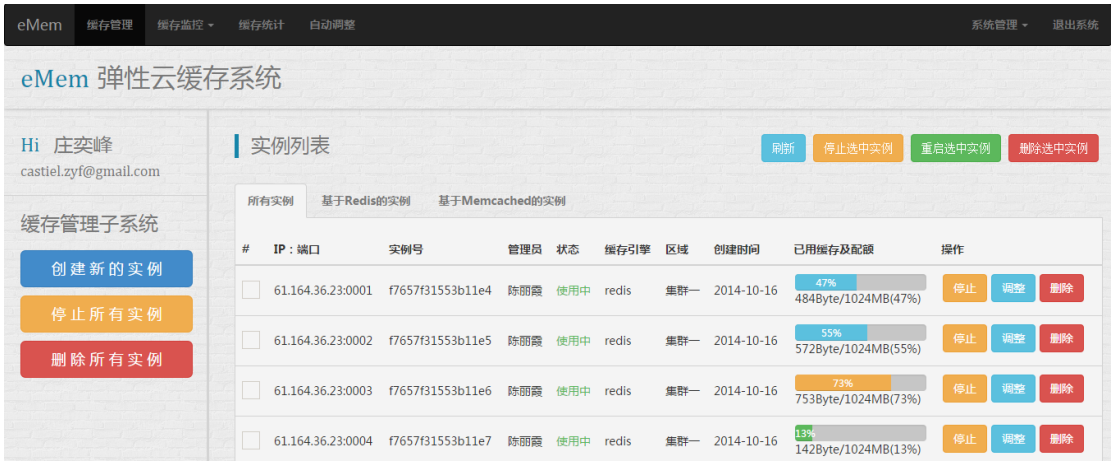

图 6 ： 缓存管理界面

下图是节点监控的界面：

图 7 ： 缓存节点监控界面

下图是 Redis 实例监控界面：

图 8 ： Redis 实例监控界面

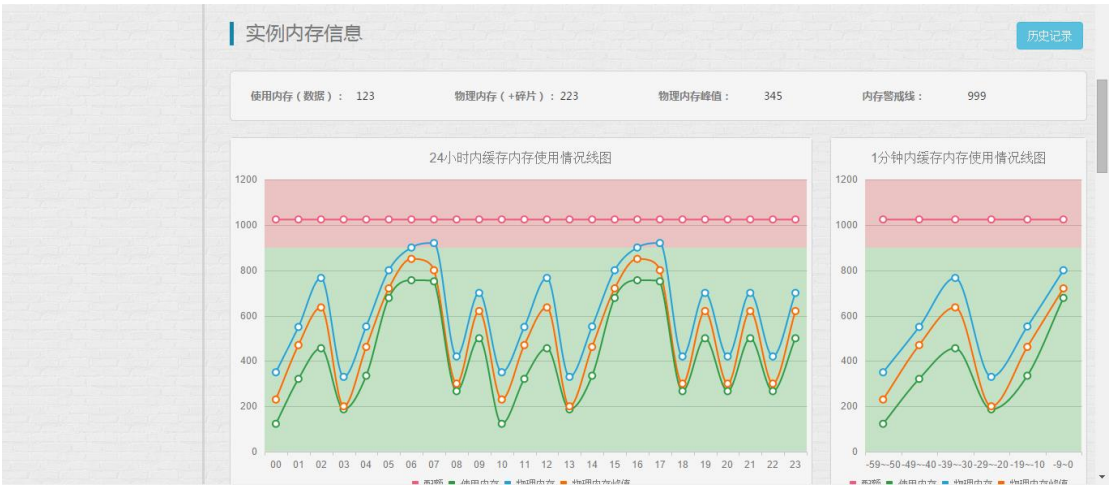

图 9 ： Redis 监控历史信息图

图 10 ：Redis 实例统计信息

六、测试分析

测试分析的过程是在整个系统已经搭建完毕的情况下进行的，此时在系统中已经启动了 4-7 个 Redis 实例，拥有 4 台通过虚拟机启动的缓存节点。

测试的方案如下：

编写 groovy 脚本文件调用 jar 包（http 包和 rmi 包）的每种操作方法，每种操作方法并发执行 10000 次。

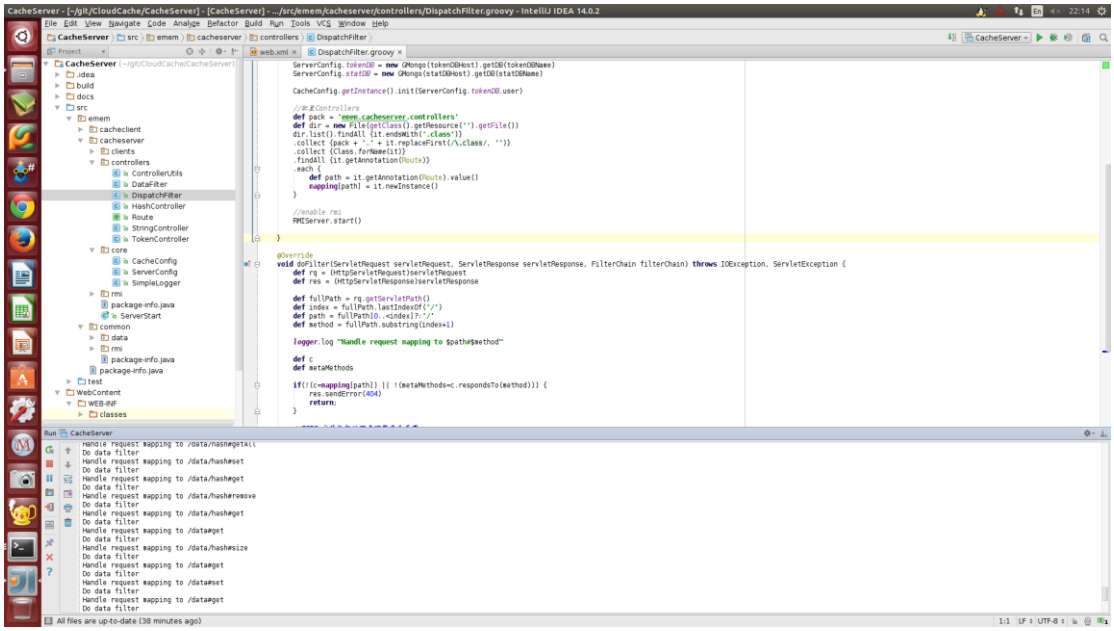

图 11 ： 测试过程中缓存服务子系统的 log 信息

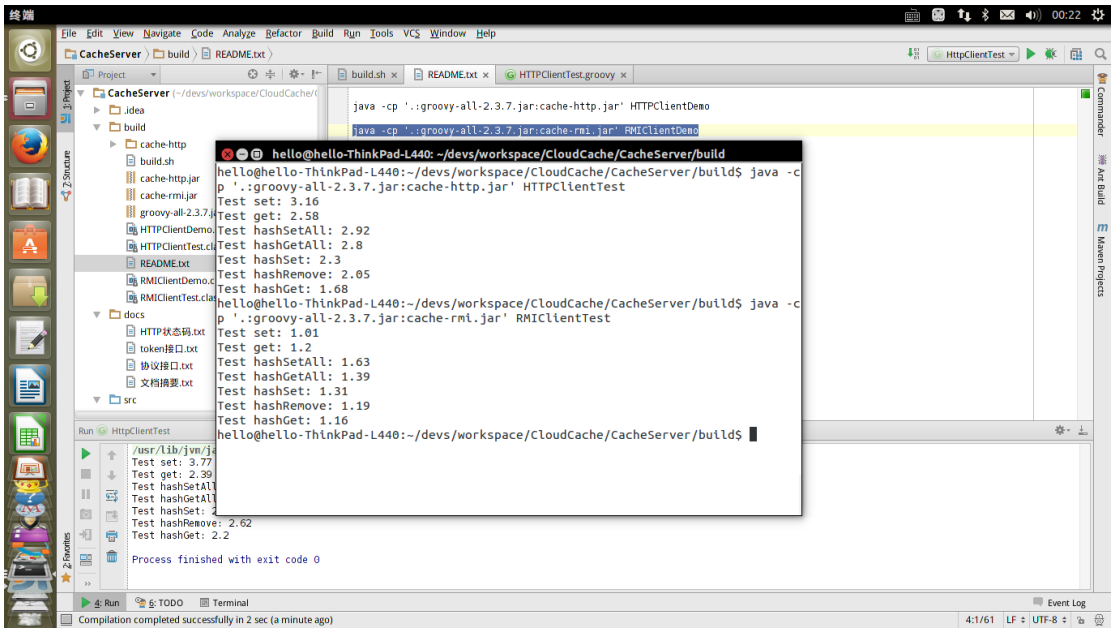

图 12 ： 测试结果截图

附录

附录 A 缓存服务子系统接口

1. 数据类型:

缓存作为 key-value 的快速映射, key 是字符串类型的. value 支持两种数据类型, 分别是 string, hash.

其中字符串类型就是字符序列, 不支持空字符序列, 空字符序列等效于 null.

其中 hash 是索引-值得集合, 索引和值都是字符串类型的. 例如下面:

{'a': 'hello', 'b': 'value'}可以看成是一个 hash 数据.

2. 协议接口

缓存服务子系统支持客户端通过两种协议来访问, 分别是 HTTP 和 RMI. 针对 HTTP 协议, 提供了客户端 JAR 封装.

注意:

下面的接口描述中, 所给的参数都不能为空, 加上[]的除外

每个接口都有用户隔离的功能, 使用 token 参数进行区别, 就没有单独列出来了

例如对于用户 mingming 来说, 其完整的调用应该是 /set?key=hello&value=world&token=mingming

没有单独提供 remove 数据的接口, 因为意识到用户很少有意识的删除数据, 故前期就不实现了.

但每个接口都添加有过期值的支持, 参数名为 expire, value 值是秒数

例如想要 key 在 10 秒后过期, 完整的调用应该是 /set?key=hello&value=world&token=mingming&expire=10

[过期值有妙用]

- **关于标量 string 的接口:**
  - /set: key, value

    存储字符串的值

    例如: /set?key=hello&value=world

- /get: key => value

  返回字符串的值

  例如: /get?key=hello => world

- **关于 hashtable 的接口:**
  - /hash/setAll: key, value

    存储 hashtable 类型的值,

    其中 hashtable 内部元素的值是由","和":"分割的字符串键值对序列,

    字符串值中的",:\"需要用"\"转义

    例如: /table/setAll?key=hello&value=k1:1,k2:2,k3:3
  - /hash/getAll: key => value

    返回 hashtable 类型的值

    例如: /table/getAll?key=hello => k1:1,k2:2,k3:3
  - /hash/set: key, index, value

    设置 hashtable 集合内指定索引键的值

    例如: /table/set?key=hello&index=k1&value=b
  - /hash/get: key, index

    返回 hashtable 集合内指定索引建的值

    例如: /table/get?key=hello&index=k1
  - /hash/remove:

    删除 hashtable 集合内指定索引建的键值对

    例如: /table/remove?key=hello&index=k1
  - /hash/size: key => size

    返回 hashtable 的大小

    例如: /hash/size?key=hello => 3

3. **HTTP 与 RMI 的缓存服务接口见下表：**以下是对 HTTP 协议的接口描述， RMI 协议的接口相同，具体可以查看 JavaDoc 的 API 文档。

java.lang.String	get(java.lang.String key) 返回绑定到制定键的字符串类型的值
java.lang.String	get(java.lang.String key, int expire)
java.io.Serializable	getObject(java.lang.String key) 返回对象类型的数据，该对象需要实现 Serializable 接口.
java.io.Serializable	getObject(java.lang.String key, int expire)
java.lang.String	hashGet(java.lang.String key, java.lang.String index) 返回 hash 类型的数据，仅仅对其中的某个索引进行操作.
java.lang.String	hashGet(java.lang.String key, java.lang.String index, int expire)
java.util.Map<java.lang.String, java.lang.String>	hashGetAll(java.lang.String key) 返回 hash 类型的完整数据

java.util.Map<java.lang.String,java.lang.String>	hashGetAll(java.lang.String key, int expire) 可以针对 hash 数据中的某个索引进行删除
void	hashRemove(java.lang.String key, java.lang.String index)
void	hashRemove(java.lang.String key, java.lang.String index, int expire)
void	hashSet(java.lang.String key, java.lang.String index, java.lang.String value) 设置 hash 类型的数据，仅仅对其中的某个索引进行操作.
void	hashSet(java.lang.String key, java.lang.String index, java.lang.String value, int expire)
void	hashSetAll(java.lang.String key, java.util.Map<java.lang.String,java.lang.String> map) 设置 hash 类型的完整数据，其中 hash 值用一个字符串到字符串的映射 map 表示

void	hashSetAll(java.lang.String key, java.util.Map<java.lang.String,java.lang.String> map, int expire)
long	hashSize(java.lang.String key) 返回的是 hash 数据中索引的个数
long	hashSize(java.lang.String key, int expire)
void	set(java.lang.String key, java.lang.String value) 存储单纯的字符串类型的键值对
void	set(java.lang.String key, java.lang.String value, int expire)
void	setObject(java.lang.String key, java.io.Serializable obj) 存储对象类型的数据，该对象需要实现 Serializable 接口.

void	setObject(java.lang.String key, java.io.Serializable obj, int expire)