**缓存组件说明**

|  |  |  |
| --- | --- | --- |
| 版本 | 编写人员 | 备注-最新修改时间 |
| 1.0 | 幺昌龙 | 2019-1-25 |

目录

[1 背景 2](#_Toc536182955)

[2说明 2](#_Toc536182956)

[3 使用示例 2](#_Toc536182957)

[3.1 导入依赖 2](#_Toc536182958)

[3.2 配置连接池 2](#_Toc536182959)

[3.3构建缓存对象 3](#_Toc536182960)

[3.3.1 原生接口 3](#_Toc536182961)

[3.3.2 组件统一的Cache操作 4](#_Toc536182962)

[4配置说明 7](#_Toc536182963)

[5接口说明 8](#_Toc536182964)

[5.1 原生连接接口——](#_Toc536182965)*[JedisPoolProxy](#_Toc536182965)* [8](#_Toc536182965)

[5.2分组接口 8](#_Toc536182966)

[5.3 缓存接口 9](#_Toc536182967)

1 背景

第三方组件的缓存，因为涉及组件自身整体架构特点，无法统一适配，只能根据组件提供的配置和扩展机制方面着手进行缓存优化。组件设计的目标是给友云音提供一个统一的缓存处理平台，适配友云音所有需要缓存的业务场景。

设计

网址：<http://123.103.9.194:4003/jin-shu-bao-jia-gou-she-ji/huan-cun-she-ji.html>

模块: 缓存设计

2说明

* 组件基于JDK1.8
* 缓存组件对外提供统一缓存接口，通过配置文件配置具体底层实际缓存服务器类型及相关连接信息即可
* 底层缓存连接同时支持配置文件配置和程序配置
* 缓存组件本身无状态，不保存数据信息，统一通过底层缓存保存，同时暴露底层缓存接口以供特殊需要使用
* 目前底层缓存服务器只实现了redis扩展
  + redis扩展支持单机、哨兵、分片、集群，对外提供统一连接配置管理和原生接口
  + redis缓存统一使用连接池配置，缓存池名称唯一标识，需要配置好默认连接的缓存池

3 使用示例

3.1 导入依赖

*<dependency>  
 <groupId>yyy.yonyoucloud.com</groupId>  
 <artifactId>yyy-cache</artifactId>  
 <version>1.0-RELEASE</version>  
</dependency>*

3.2 配置连接池

* 连接池配置必须指定 名称、类型、连接地址
* 连接池名称唯一标识一个连接池，已存在的名称不会再次加载
* **配置文件**

**配置文件地址：conf/cache.cfg**

配置文件中可提供多个连接池，但必须指定一个默认的连接池

1.指定默认的连接池名称 , 默认使用jedis

cache.redis.pool.default=jedis # 可选

2.声明要配置的连接池名称，必须包含上述默认的连接池

**cache.redis.pool.names**=jedis,cluster #jedis为1中，或默认指定必须配置的

3.配置声明的连接池，以声明的连接池名称前缀

type、server为必须指定项，其他参考后面配置项说明

**jedis.type**=jedis #必须制定pool类型

**jedis.password** = ycm30 # 没有就不需要设置

**jedis.server** = host@port #必须指定

#jedis.timeout = 100000

#jedis.db = 0

#jedis.maxIdle = 8

...

cluster.type=cluster

cluster.server = host@port,host @port

* **程序中配置连接**

程序中未提供配置对象，或未提供连接池名称、地址、类型时，使用默认的连接池，否则使用用户自定义的连接池

程序中可以在构建缓存时提供连接属性对象

**private static** Properties propeties = **new** Properties();  
propeties.put(**"name"**, **"otherPool"**);  
propeties.put(**"type"**, **"jedis"**);  
propeties.put(**"password"**, **"ycm30"**);  
propeties.put(**"server"**, **"172.20.12.72@6379"**);

3.3构建缓存对象

3.3.1 原生接口

*//获取指定连接池*

*//此处获取的默认连接池,由***cache.redis.pool.default**=**jedis** *指定，并在配置文件中通过 "默认名称".\* 配置连接池的连接信息。*

JedisPoolProxy pool = JedisPoolFactory.*instance*().getPool(CacheConstants.*CACHE\_POOL\_REDIS\_DEFAULT*);

* + 手动管理redis连接，需要手动关闭连接

JedisCommands redis = pool.getResource();  
pool.releaseResource(redis);*//手动释放资源  
//JedisPoolProxy对外接口，无需管理连接*

* + 统一的redis回调，无需管理资源 *//redis操作*String value = pool.oper((r) -> r.get(**"key"**));
  + 统一的pipline操作，无需管理资源

*//pip操作*pool.pipline((pip) -> {  
 Arrays.stream(**new** String[]{**"key1"**, **"key2"**})  
 .forEach((key) -> pip.del(key));  
 **return null**;  
});

3.3.2 组件统一的Cache操作

* 构建缓存对象

*// 定义缓存构造器，可以设置缓存构造器的各种规则***final** CacheBuilder userCacheBuilder = CacheBuilderFactory.*getFactory*().create()  
 *// 通过API方式设置缓存服务器属性  
 //*.props(**new** Properties())  
 *// 设置Key生成规则，如果未配置使用系统默认的FastJSON进行序列化的String为Key* .keyConvertor(**new** User.UserKeyConvertor())  
 *// 设置Value的序列化和反序列化过程，如果未配置使用系统默认的FastJSON进行序列化和反序列化* .valueConvertor(**new** User.UserValueConvertor())  
 *// 设置缓存移除的监听接口，当缓存被移除的时候会执行监听回调* .removalListener(**new** User.UserRemovalListener())  
 *// 设置分组信息* .group(***CACHE\_GROUP***)  
 *// 设置缓存到期时间  
// .expireAt(EXPIRE\_TIME, TimeUnit.MINUTES);  
 // 设置缓存的失效时间* .expireAfterAccess(***EXPIRE\_TIME***, TimeUnit.***MINUTES***);  
  
*// 通过缓存构造器生成单个缓存对象***final** Cache<Long, User> userCache = userCacheBuilder.buildCache(User.**class**);

**public static final class** UserKeyConvertor **implements** KeyConvertor<Long> {  
  
 @Override  
 **public** String convert(Long userId) **throws** ExecutionException {  
 **return "@"** + userId;  
 }  
}  
  
**public static final class** UserValueConvertor **implements** ValueConvertor<User, String> {  
 @Override  
 **public** String serialise(User user) {  
 **return** JSON.*toJSONString*(user);  
 }  
  
 @Override  
 **public** User deserialise(String value, Class<User> cls) {  
 **return** JSON.*parseObject*(value, User.**class**);  
 }  
}  
  
**public static final class** UserRemovalListener **implements** RemovalListener<Long, User> {  
 @Override  
 **public void** onRemoval(RemovalNotification<Long, User> notification) {  
 ***log***.info(**"test removeListener,{}, removed ..."**, notification.getValue());  
 }  
}

**public class** User **implements** Serializable {  
  
 **private** Long **userId**;  
  
 **private** String **name**;  
  
 **public** User() {  
 }  
}

* 查询时回调

//只查询缓存

**public static** User loadUser(Long userId, Cache<Long, User> userCache) {  
 User u = **null**;  
 **try** {  
 u = userCache.get(userId);  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 **return** u;  
}  
//查询结果为空时进行回调加载对象，并缓存  
**public static** User loadUserWithCall(Long userId, Cache<Long, User> userCache) {  
 User u = **null**;  
 **try** {  
 u = userCache.get(userId, () -> User.*loadUserFormDB*(userId));  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 **return** u;  
}

**public static** User loadUserFormDB(Long userId) {  
 User u = **null**;  
 **try** {  
 Thread.*sleep*(2 \* 1000);  
 u = **new** User(userId, **"yyy"**);  
 ***log***.info(**"load user from DB:"** + userId);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 **return** u;  
}

* 分组失效

CacheGroup group = ((AbstractCache) **userCache**).getOrCreateCacheGroup();

***log***.info(**"创建三个用户"**);  
User.*loadUserWithCall*(***USER\_ID1***, **userCache**);  
User.*loadUserWithCall*(***USER\_ID2***, **userCache**);  
User.*loadUserWithCall*(***USER\_ID3***, **userCache**);  
Set<String> gKs2 = group.lists();  
Assert.*assertEquals*(3, gKs2.size());  
u1 = User.*loadUser*(***USER\_ID1***, **userCache**);  
u2 = User.*loadUser*(***USER\_ID2***, **userCache**);  
u3 = User.*loadUser*(***USER\_ID3***, **userCache**);  
Assert.*assertNotNull*(u1);  
Assert.*assertNotNull*(u2);  
Assert.*assertNotNull*(u3);  
group.invalidAll(() -> {  
 ***log***.info(**"删除分组完毕！"**);  
 **return null**;  
});  
gKs2 = group.lists();  
Assert.*assertEquals*(0, gKs2.size());  
u1 = User.*loadUser*(***USER\_ID1***, **userCache**);  
u2 = User.*loadUser*(***USER\_ID2***, **userCache**);  
u3 = User.*loadUser*(***USER\_ID3***, **userCache**);  
Assert.*assertNull*(u1);  
Assert.*assertNull*(u2);  
Assert.*assertNull*(u3);

4配置说明

|  |  |  |  |
| --- | --- | --- | --- |
| **属性** | **必填** | **属性值** | **默认值** |
| cache.types | 否 | 缓存类型,目前只支持redis ^\_^ | redis |
| cache.redis.pool.default | 否 | 默认连接池名称  指定的默认连接池必须配置连接  对应的连接池声明和配置也需要对应修改 | jedis |
| **cache.redis.pool.names** | 是 | = jedis,..... |  |
| **连接池配置（配置文件中必须前缀对应的名称）** | | | |
| **name** | 程序  使用 | 程序中提供连接信息时，必须指定连接池名称 | cache.redis.pool.default  指定的默认连接池 |
| **type** | 是 | 连接池类型  支持jedis、shard、sentinel、cluster |  |
| **server** | 是 | 连接池地址 |  |
| **password** | 是 | 连接池密码 |  |
| timeout | 否 | 支持　JedisPool\JedisCluster | 10000 |
| db | 否 | 支持　JedisPool\JedisCluster | 0 |
| maxAttempts = | 否 | 支持　JedisCluster | 10 |
| masterName | 否 | 支持　JedisSentinelPool | master |
| maxTotal | 否 | 连接池最大资源数 | 8 |
| maxIdle | 否 | 连接池最大闲置资源数 | 8 |
| minIdle | 否 | 连接池最小资源数 | 0 |
| testOnBorrow | 否 | 获取连接是进行测试 | true |
| maxWaitMillis | 否 | 最大等待连接时间 | -1 |

5接口说明

5.1 原生连接接口——*JedisPoolProxy*

原生接口支持直接获取redis连接，支持通过回调使用redis连接、或者管道进行原生操作。

直接获取连接，必须主动释放连接资源，通过回调则不需要。 *//jedis原生连接*

*JedisCommands* ***getResource****();  
 //资源释放  
 void* ***releaseResource****(JedisCommands jedis) throws Exception;  
  
 /\*\*  
 \* 统一操作接口,JedisOper暴露redis连接  
 \*  
 \* @return JedisOper回调函数的返回结果  
 \*/  
 <T> T oper(JedisOper<T> jedisOper) throws Exception;  
   
 /\*\*  
 \* 统一pipline接口，PipelineOper暴露pipline连接  
 \*  
 \* @return PipelineOper回调函数的返回结果  
 \*/  
 <T> List<Response<T>> pipline(PipelineOper<T> pipOper) throws Exception;*

5.2分组接口

*//添加key到当前分组  
 void* ***add****(K key) throws Exception;  
 //添加多个key到当前分组  
 void* ***add****(K[] key) throws Exception;  
 //移除分组中某个key  
 void* ***remove****(K key) throws Exception;  
 //获取分组的key列表  
 Set<K>* ***lists****() throws Exception;  
 //分组失效：移除分组及数据  
 boolean invalidAll(Callable cb) throws Exception;*

5.3 缓存接口

*boolean* ***exists****(K key) throws Exception;  
   
 /\*\*  
 \* 缓存不存在则使用loader加载并缓存  
 \*/  
 V* ***get****(K key, Callable<? extends V> loader) throws Exception;  
   
 /\*\*  
 \* get from cache  
 \*/  
 V* ***get****(K key) throws Exception;  
   
 /\*\*  
 \* get from cache  
 \*/  
 List<V>* ***getAll****(K[] keys) throws Exception;  
   
 boolean* ***put****(K key, V value) throws Exception;  
   
 boolean* ***putIfAbsent****(K key, V value) throws Exception;  
   
 boolean* ***putAll****(Map<K, V> kvMap) throws Exception;  
   
 boolean* ***remove****(K key) throws Exception;  
   
 boolean* ***remove****(K key, Callable<? extends V> call) throws Exception;  
   
 /\*\*  
 \* 清除keys的缓存,移除分组  
 \*/  
 boolean* ***remove****(K[] keys, Callable<? extends V> call) throws Exception;  
   
 /\*\*  
 \* 字符串追加  
 \* value使用{@link CacheConfig#valueConvertor}序列化后追加到原有缓存值  
 \*/  
 Long* ***append****(K key, V value) throws Exception;  
   
 Long* ***incr****(K key, int step) throws Exception;  
   
 Long* ***decr****(K key, int step) throws Exception;  
   
 /\*\*  
 \* 永久缓存  
 \*/  
 boolean* ***persist****(K key) throws Exception;*