# Redis缓存介绍

## 缓存策略

### 缓存机制

缓存是介于应用程序和物理数据源之间，其作用是为了降低应用程序对物理数据源访问的频次，从而提高了应用的运行性能。缓存内的数据是对物理数据源中的数据的复制，应用程序在运行时从缓存读写数据，在特定的时刻或事件会同步缓存和物理数据源的数据。

缓存的介质一般是内存，所以读写速度很快。但如果缓存中存放的数据量非常大时，也会用硬盘作为缓存介质。缓存的实现不仅仅要考虑存储的介质，还要考虑到管理缓存的并发访问和缓存数据的生命周期。

### 缓存中存在的问题

1. 缓存中的数据是真实数据库数据的备份,如何保证数据一致性.
2. 数据都存储在内存中,如果断电/宕机/内存数据如何恢复.
3. 如果缓存中的数据如果达到临界值(512M),缓存大小如何维护.
4. 缓存中保存数据应该采用什么样的数据存储结构.

## 缓存三大问题

### 缓存穿透

条件:访问一个不存在的数据

说明:当访问一个不存在的数据时,因为缓存中没有这个key,导致缓存形同虚设.最终访问后台数据库.但是数据库中没有该数据所以返回null.

隐患:如果有人恶意频繁查询一个不存在的数据,可能会导致数据库负载高导致宕机.

总结:业务系统访问一个不存在的数据,称之为缓存穿透.

### 缓存击穿

条件:当缓存key失效/过期/未命中时,高并发访问该key

说明:如果给一个key设定了失效时间,当key失效时有一万的并发请求访问这个key,这时缓存失效,所有的请求都会访问后台数据库.称之为缓存击穿.

场景:微博热点消息访问量很大,如果该缓存失效则会直接访问后台数据库,导致数据库负载过高.

### 缓存雪崩

前提:高并发访问,缓存命中较低或者失效时

说明:假设缓存都设定了失效时间,在同一时间内缓存大量失效.如果这时用户高并发访问.缓存命中率过低.导致全部的用户访问都会访问后台真实的数据库.

场景:在高并发条件下.缓存动态更新时

## Redis安装

### Redis介绍

redis是一个key-value[存储系统](https://baike.baidu.com/item/%E5%AD%98%E5%82%A8%E7%B3%BB%E7%BB%9F)。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](https://baike.baidu.com/item/%E9%93%BE%E8%A1%A8))、set(集合)、zset(sorted set --有序集合)和hash（哈希类型）。这些[数据类型](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B" \t "_blank)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。它可以用作数据库、缓存和消息中间件.在此基础上，redis支持各种不同方式的排序。与memcached一样，为了保证效率，数据都是缓存在内存中。区别的是redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步。

Redis 是一个高性能的key-value数据库。 redis的出现，很大程度补偿了[memcached](https://baike.baidu.com/item/memcached" \t "_blank)这类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。它提供了Java，C/C++，C#，PHP，JavaScript，Perl，Object-C，Python，Ruby，Erlang等客户端，使用很方便。[1]

Redis支持主从同步。数据可以从主服务器向任意数量的从服务器上同步，从服务器可以是关联其他从服务器的主服务器。这使得Redis可执行单层树复制。存盘可以有意无意的对数据进行写操作。由于完全实现了发布/订阅机制，使得从数据库在任何地方同步树时，可订阅一个频道并接收主服务器完整的消息发布记录。同步对读取操作的可扩展性和数据冗余很有帮助。

Redis的运行速度:读速度110000/s,写速度81000次每秒.平均10万/s

### Redis下载

网址: <https://redis.io/download>

如图-1所示

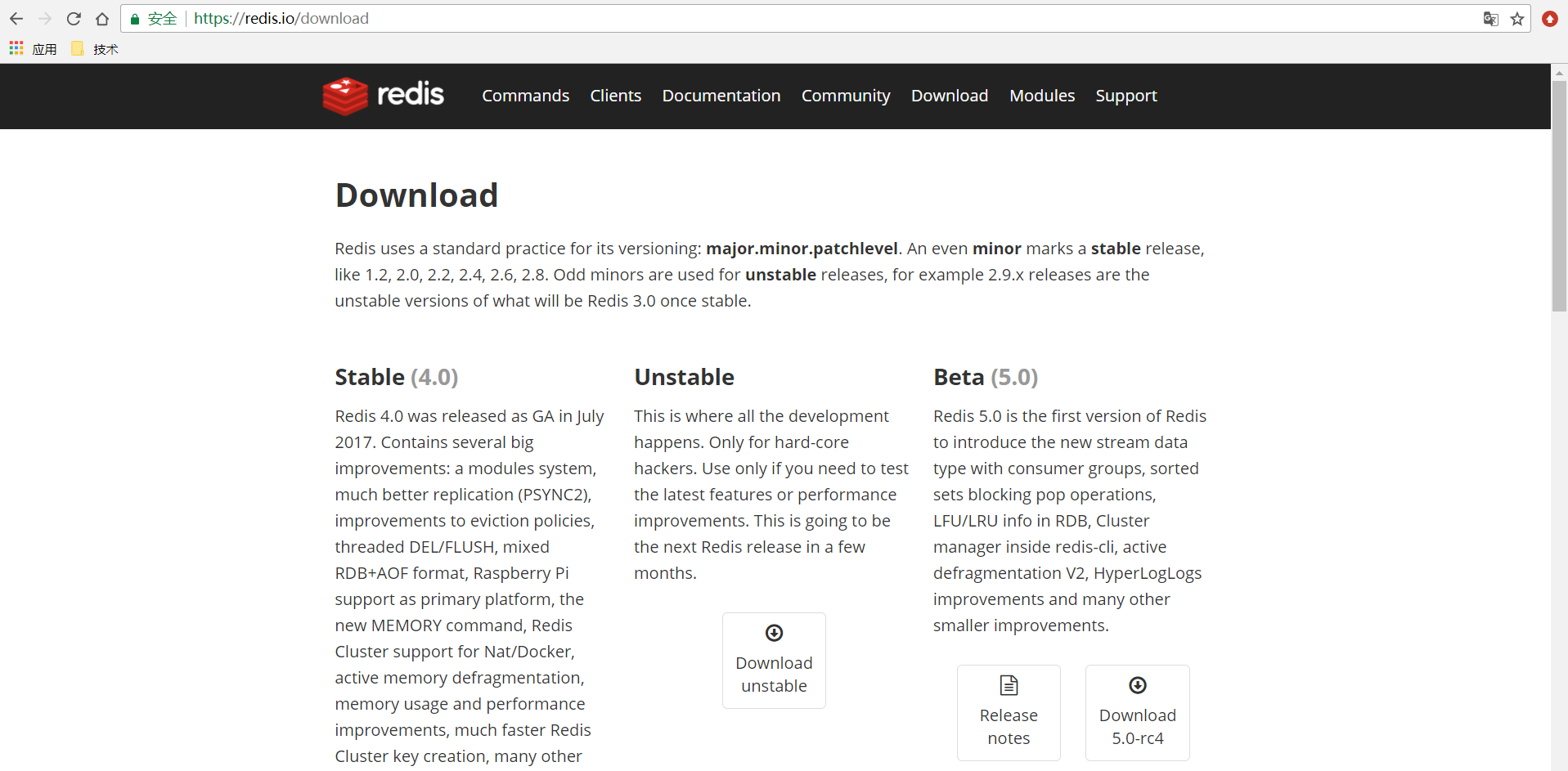


图- 1

下载方式2:

wget <http://download.redis.io/releases/redis-4.0.11.tar.gz>

选择正确的版本号即可.课上演示为3.2.8版本

### Redis安装

上传redis安装包,到指定目录下/usr/local/src下

如图-2所示

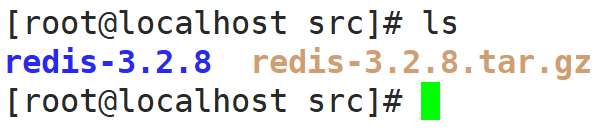


图- 2

解压redis命令:

tar -xvf redis.3.2.8.tar.gz

### 校验编译Redis

1. Make 编译

2.Make install 安装

LINK redis-server #redis服务启动项

INSTALL redis-sentinel #redis哨兵的机制(实现redis高可用)

LINK redis-cli #连接redis客户端程序

INSTALL redis-check-rdb #校验redis持久化方式

LINK redis-check-aof

如图-3所示

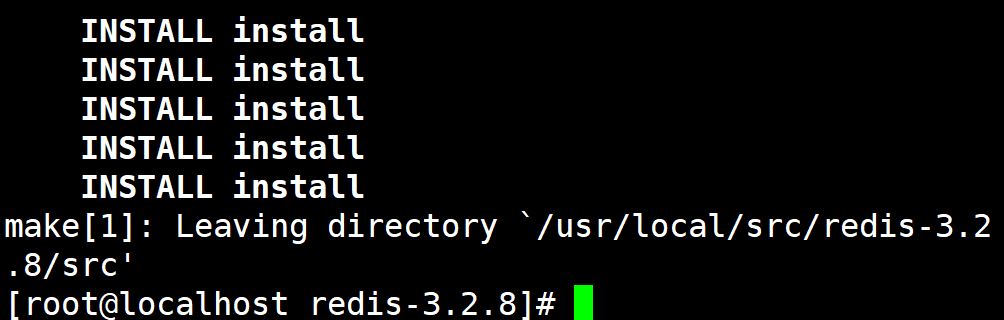


图- 3

### 修改配置文件

说明:作为第一个

注释IP绑定

如图-4所示

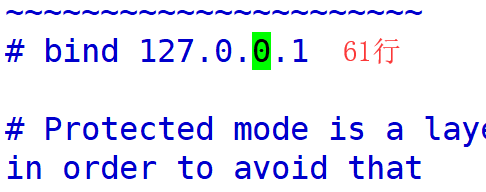


图- 4

关闭保护模式

如图-5所示

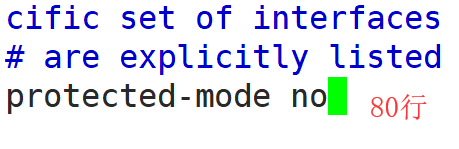


图- 5

开启后台启动

如图-6所示

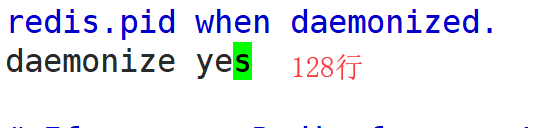


图- 6

### Redis启动命令

安装前提:需要安装GCC

 yum -y install gcc automake autoconf libtool make

1.redis-server 缺点:不能执行后续操作

2.redis-server & 表示有后续操作.

总结:以上的命令 启动时redis的设置都是默认的 IP绑定/保护模式默认值处理

3.redis-server redis.conf 按照配置文件方式启动

4.kill -9 pid

5.redis-cli shutdown或者 redis-cli -p 6379 shutdown

### 启动效果图

如图-7所示

计算机生成了可选文字:
助dis4.0.1(00以拍以犯／0)64bit
、、、＼/
RIJnnln
POrt:
PID:
口一里旦
6379
St己nd己IOnemode
6181
http://redis.10
、．口口．.................1
｝一一
一一一
｝一一
一一一一
一一一
『一一
}｝一

图- 7

## Redis命令

### String类型

|  |  |  |
| --- | --- | --- |
| 命令 | 说明 | 案例 |
| set | 添加key-value | set username admin |
| get | 根据key获取数据 | get username |
| strlen | 获取key的长度 | strlen key |
| exists | 判断key是否存在 | exists name  返回1存在 0不存在 |
| del | 删除redis中的key | del key |
| Keys | 用于查询符合条件的key | keys \* 查询redis中全部的key  keys n?me 使用占位符获取数据  keys nam\* 获取nam开头的数据 |
| mset | 赋值多个key-value | mset key1 value1 key2 value2 key3 value3 |
| mget | 获取多个key的值 | mget key1 key2 |
| append | 对某个key的值进行追加 | append key value |
| type | 检查某个key的类型 | type key |
| select | 切换redis数据库 | select 0-15 redis中共有16个数据库 |
| flushdb | 清空单个数据库 | flushdb |
| flushall | 清空全部数据库 | flushall |
| incr | 自动加1 | incr key |
| decr | 自动减1 | decr key |
| incrby | 指定数值添加 | incrby 10 |
| decrby | 指定数值减 | decrby 10 |
| expire | 指定key的生效时间 单位秒 | expire key 20  key20秒后失效 |
| pexpire | 指定key的失效时间 单位毫秒 | pexpire key 2000  key 2000毫秒后失效 |
| ttl | 检查key的剩余存活时间 | ttl key |
| persist | 撤销key的失效时间 | persist key |

### Hash类型

说明:可以用散列类型保存对象和属性值

例子:User对象{id:2,name:小明,age:19}

|  |  |  |
| --- | --- | --- |
| 命令 | 说明 | 案例 |
| hset | 为对象添加数据 | hset key field value |
| hget | 获取对象的属性值 | hget key field |
| hexists | 判断对象的属性是否存在 | HEXISTS key field  1表示存在 0表示不存在 |
| hdel | 删除hash中的属性 | hdel user field [field ...] |
| hgetall | 获取hash全部元素和值 | HGETALL key |
| hkyes | 获取hash中的所有字段 | HKEYS key |
| hlen | 获取hash中所有属性的数量 | hlen key |
| hmget | 获取hash里面指定字段的值 | hmget key field [field ...] |
| hmset | 为hash的多个字段设定值 | hmset key field value [field value ...] |
| hsetnx | 设置hash的一个字段,只有当这个字段不存在时有效 | HSETNX key field value |
| hstrlen | 获取hash中指定key的长度 | HSTRLEN key field |
| hvals | 获取hash的所有值 | HVALS user |

### List类型

说明:Redis中的List集合是双端循环列表,分别可以从左右两个方向插入数据.

List集合可以当做队列使用,也可以当做栈使用

队列:存入数据的方向和获取数据的方向相反

栈:存入数据的方向和获取数据的方向相同

|  |  |  |
| --- | --- | --- |
| 命令 | 说明 | 案例 |
| lpush | 从队列的左边入队一个或多个元素 | LPUSH key value [value ...] |
| rpush | 从队列的右边入队一个或多个元素 | RPUSH key value [value ...] |
| lpop | 从队列的左端出队一个元素 | LPOP key |
| rpop | 从队列的右端出队一个元素 | RPOP key |
| lpushx | 当队列存在时从队列的左侧入队一个元素 | LPUSHX key value |
| rpushx | 当队列存在时从队列的右侧入队一个元素 | RPUSHx key value |
| lrange | 从列表中获取指定返回的元素 | LRANGE key start stop  Lrange key 0 -1 获取全部队列的数据 |
| lrem | 从存于 key 的列表里移除前 count 次出现的值为 value 的元素。 这个 count 参数通过下面几种方式影响这个操作：   * count > 0: 从头往尾移除值为 value 的元素。 * count < 0: 从尾往头移除值为 value 的元素。 * count = 0: 移除所有值为 value 的元素。 | LREM list -2 “hello” 会从存于 list 的列表里移除最后两个出现的 “hello”。  需要注意的是，如果list里没有存在key就会被当作空list处理，所以当 key 不存在的时候，这个命令会返回 0。 |
| Lset | 设置 index 位置的list元素的值为 value | LSET key index value |

### Redis事务命令

说明:redis中操作可以添加事务的支持.一项任务可以由多个redis命令完成,如果有一个命令失败导致入库失败时.需要实现事务回滚.

|  |  |  |
| --- | --- | --- |
| 命令 | 说明 | 案例 |
| multi | 标记一个事务开始 | 127.0.0.1:6379> MULTI  OK |
| exec | 执行所有multi之后发的命令 | 127.0.0.1:6379> EXEC  OK |
| discard | 丢弃所有multi之后发的命令 |  |

# Redis高级应用

## Redis入门案例

### 添加jar包文件

说明:在JT-PARENT项目中添加jar包文件

<!-- jedis -->

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>${jedis.version}</version>

</dependency>

<!--添加spring-datajar包 -->

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-redis</artifactId>

<version>1.4.1.RELEASE</version>

</dependency>

### 入门案例-String

/\*\*

\* 连接单台redis

\* 参数介绍:

\* redisIP地址.

\* redis:6379

\*/

@Test

**public** **void** test01(){

Jedis jedis = **new** Jedis("192.168.126.166",6379);

jedis.set("redis", "redis入门案例");

System.***out***.println

("获取redis中的数据:"+jedis.get("redis"));

//为数据设定超时时间 单位秒

jedis.setex("1804", 100, "1804班");

}

### 入门案例-hash

@Test

**public** **void** test01(){

Jedis jedis = **new** Jedis("192.168.126.148", 6379);

jedis.hset("user", "id", "1");

jedis.hset("user", "name", "tomcat");

jedis.hset("user", "age", "18");

System.***out***.println("操作完成!!!"+jedis.hget("user", "id"));

Map<String,String> map = jedis.hgetAll("user");

System.***out***.println(map);

}

结果展现:

操作完成!!!1

{name=tomcat, age=18, id=1}

### 入门案例-List

@Test

**public** **void** test02(){

Jedis jedis = **new** Jedis("192.168.126.148", 6379);

Long number = jedis.lpush("list", "a","b","c","d","e");

System.***out***.println("获取数据"+number);

List<String> list= jedis.lrange("list", 0, -1);

System.***out***.println("获取参数:"+list);

}

结果展现:

获取数据5

获取参数:[e, d, c, b, a]

## Spring整合Redis

### 编辑properties配置文件

redis.host=192.168.126.166

redis.port=6379

redis.maxTotal=1000

redis.maxIdle=100

redis.testOnBorrow=true

### Spring添加配置文件

<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

<!--Resource[] locations; -->

<property name="locations">

<list>

<value>classpath:/properties/jdbc.properties</value>

<value>classpath:/properties/redis.properties</value>

</list>

</property>

</bean>

### Spring整合Redis(初级)

<!--通过bean标签管理Jedis对象 -->

<bean id=*"jedis"* class=*"redis.clients.jedis.Jedis"*>

<constructor-arg name=*"host"* value=*"${redis.host}"*/>

<constructor-arg name=*"port"* value=*"${redis.port}"*/>

</bean>

### Spring整合Redis(高级)

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc-4.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

<!--每次都创建jedis对象性能较低 创建jedis线程池 -->

<bean id="poolConfig" class="redis.clients.jedis.JedisPoolConfig">

<!--定义连接总数 -->

<property name="maxTotal" value="${redis.maxTotal}"/>

<!--定义最大闲置资源 -->

<property name="maxIdle" value="${redis.maxIdle}"/>

<!--定义是否自动连接测试 测试当前连接是否可用,如果有问题,则获取新的连接-->

<property name="testOnBorrow" value="${redis.testOnBorrow}"/>

</bean>

<!--定义jedis工厂模式 -->

<bean id="jedisConnectionFactory" class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory" >

<!--定义远程主机 -->

<property name="hostName" value="${redis.host}"/>

<!--定义端口号 -->

<property name="port" value="${redis.port}"/>

<!--引入配置文件 -->

<property name="poolConfig" ref="poolConfig"/>

</bean>

<!--定义redisTemplate方便操作redis -->

<bean id="stringRedisTemplate" class="org.springframework.data.redis.core.StringRedisTemplate">

<property name="connectionFactory" ref="jedisConnectionFactory"/>

</bean>

</beans>

### Spring整合Redis-工具类(高级适用)

编辑工具类代码RedisService

package com.jt.common.service;

import java.util.concurrent.TimeUnit;

import org.apache.log4j.Logger;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.data.redis.core.RedisTemplate;

import org.springframework.data.redis.core.StringRedisTemplate;

import org.springframework.data.redis.core.ValueOperations;

import org.springframework.stereotype.Component;

import org.springframework.stereotype.Service;

@Service

public class RedisService {

@Autowired

private StringRedisTemplate redisTemplate;

//获取数据

public String get(String key){

ValueOperations<String, String> operations =redisTemplate.opsForValue();

return operations.get(key);

}

//插入数据

public boolean set(String key,String value){

ValueOperations<String, String> operations = redisTemplate.opsForValue();

try {

operations.set(key, value);

return true;

} catch (Exception e) {

e.printStackTrace();

return false;

}

}

//插入缓存定义超时时间为秒

public boolean set(String key,String value,Long expireTime){

try {

ValueOperations<String, String> operations = redisTemplate.opsForValue();

operations.set(key, value, expireTime, TimeUnit.DAYS);

return true;

} catch (Exception e) {

e.printStackTrace();

return false;

}

}

}

## 实现商品分类缓存处理

### 需求分析

说明:

当点击商品分类列表时,应该使用redis缓存,parentId充当redis中的Key,获取当前节点的全部子节点信息充当value(JSON串)

实现思路:

1. 根据parentID先查询redis缓存
2. 如果redis中有数据,则JSON串转化为java对象之后返回.
3. 如果redis中没有数据.则查询数据库,需要将javaList集合转化为JSON串.之后保存到redis中.之后返回用户.

### 编辑Controller

说明:查询redis缓存服务

@RequestMapping("/list")

@ResponseBody

**public** List<EasyUITree> findItemCatList(

@RequestParam(defaultValue="0",value="id")Long parentId){

//查询一级商品分类目录

//Long parentId = 0L;

List<EasyUITree> treeList =

itemCatService.findItemCatCache(parentId);

**return** treeList;

}

### 编辑Service

@Override

**public** List<EasyUITree> findItemCatCache(Long parentId) {

List<EasyUITree> treeList = **null**;

String key = "ITEM\_CAT\_"+parentId;

String jsonData = jedis.get(key);

**try** {

**if**(StringUtils.*isEmpty*(jsonData)){

//缓存中没有数据,需要查询数据库

treeList = findItemCatList(parentId);

String json =

***objectMapper***.writeValueAsString(treeList);

jedis.set(key,json);

System.***out***.println("第一次查询");

**return** treeList;

}**else**{

//缓存中有数据 [{id....},{},{}]

EasyUITree[] trees =

***objectMapper***.readValue

(jsonData,EasyUITree[].**class**);

treeList = Arrays.*asList*(trees);

System.***out***.println("查询缓存");

}

} **catch** (Exception e) {

e.printStackTrace();

}

**return** treeList;

}

### 适用Redis效果展现

1. 在没有添加redis缓存条件下

如图-8所示

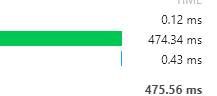


图- 8

1. 在使用redis缓存条件下

通过下图可以发现使用redis时速度提升明显.提升超过10倍.由此可见redis的性能异常快速.

如图-9所示

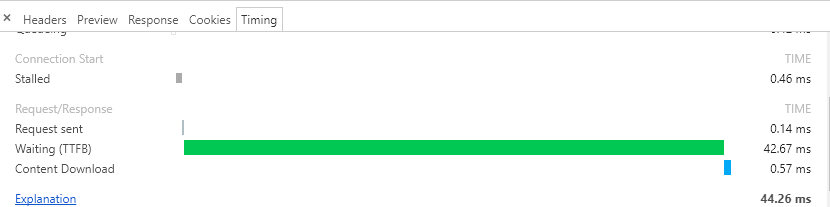


图- 9

## 关于ObjectMapper使用方法

### 引入jar包

在程序中经常需要将对象转化JSON,将JSON转化为对象.所以必须使用工具类的方式进行.ObjectMapper是jackson中提供的工具类.

<!-- Jackson Json处理工具包 -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.4.2</version>

</dependency>

### 将对象与JSON互转

/\*\*

\* 对象转化为JSON需要调用对象的get方法.

\* **@throws** IOException

\*/

@Test

**public** **void** test01() **throws** IOException{

ObjectMapper objectMapper = **new** ObjectMapper();

User user = **new** User();

user.setId(1);

user.setName("tomcat");

user.setAge(18);

user.setSex("男");

String json =

objectMapper.writeValueAsString(user);

System.***out***.println(json);

//将json串转化为java对象

User jsonToUser = objectMapper.readValue(json, User.**class**);

System.***out***.println(jsonToUser.toString());

}

### 将List集合与JSON互转

@Test

**public** **void** test02() **throws** IOException{

ObjectMapper objectMapper = **new** ObjectMapper();

User user = **new** User();

user.setId(1);

user.setName("tomcat");

user.setAge(18);

user.setSex("男");

User user1 = **new** User();

user1.setId(2);

user1.setName("tomcat");

user1.setAge(18);

user1.setSex("男");

List<User> userList = **new** ArrayList<>();

userList.add(user1);

userList.add(user);

String json =

objectMapper.writeValueAsString(userList);

System.***out***.println(json);

//[{"id":2,"age":18,"sex":"男"},{"id":1,"age":18,"sex":"男"}]

List<User> jsonList = **new** ArrayList<>();

jsonList =

objectMapper.readValue(json,jsonList.getClass());

System.***out***.println(jsonList);

}