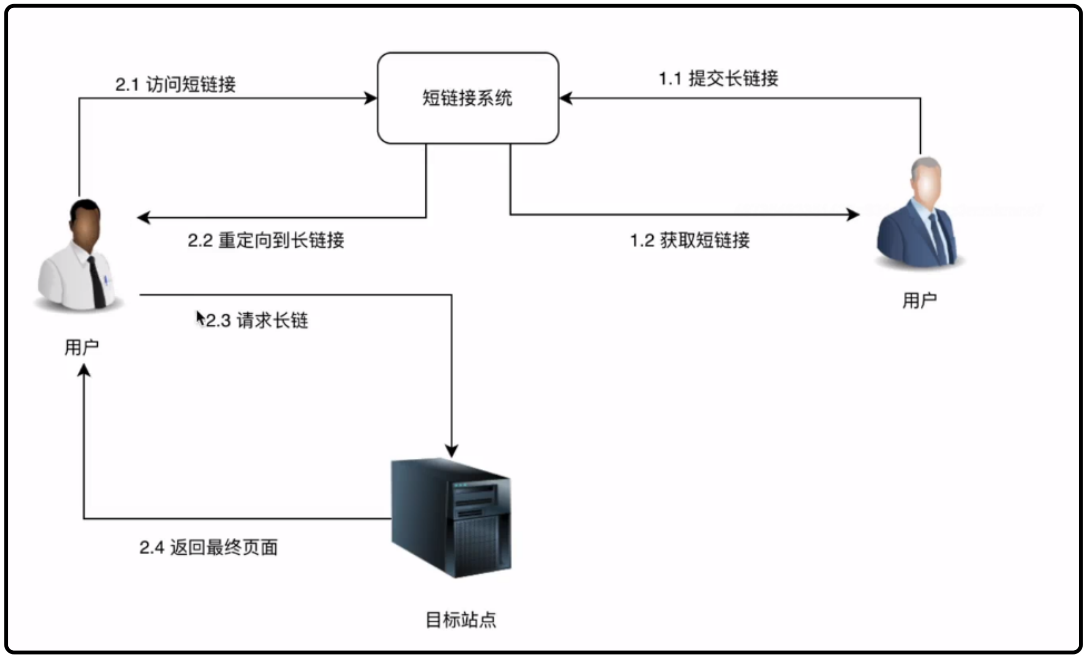
SpringBoot 实现短链接系统

项目简介以及环境搭建

什么是短链接系统?

短链接系统是一种将长网址转化为短网址的服务，旨在缩短 URL 并且提供友好的链接形式，长网址可能包含大量字符，不方便分享、输入或记忆，而短链接则可以将长网址映射到短字符串，使得链接更加简洁和方便使用。



短链接系统的优势

短链接系统的主要优势和应用包括：

1. **便捷分享**：短链接更短，更加易于分享，适用于社交媒体、短信、邮件等场景，提供更加美观的外观
2. **提高用户体验：**短链接可以简化用户输入，减少用户访问链接时的操作，提升用户体验
3. **推广以及营销：**短链接可以用于推广和营销活动，跟踪广告点击以及转化率，帮助优化营销策略

创作目的

- 短链接系统在互联网界的应用十分广泛
- 短链接系统业务逻辑简单，但是可能涉及到许多很复杂的技术（例如缓存，异地多活，分库分表）
- 设计一个短链接系统，是国内外很多互联网公司的高频面试题（例如 Google、FaceBook、字节跳动、快手、阿里）

使用技术栈

- 编程语言：Java
- 使用技术：Spring Boot 3 .X 、JPA 、MySQL 、Guava、JUNIT 5、Lomok、Druid 数据库连接池
- 项目使用纯后端实现，没有前端，可以做成一个 starter 引入到其他项目中去

准备工作

阿里云加速

```
<!--    阿里云镜像    -->
<repositories>
    <repository>
        <id>central</id>
        <name>aliyun maven</name>
        <url>https://maven.aliyun.com/nexus/content/groups/public/</url>
        <layout>default</layout>
        <!--    是否开启发布版构件下载          -->
        <releases>
            <enabled>true</enabled>
        </releases>
        <!--    是否开启快照版构件下载          -->
        <snapshots>
            <enabled>false</enabled>
        </snapshots>
    </repository>
</repositories>
```

继承 SpringBoot 项目

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.1.0</version>
    <relativePath/>
    <!-- lookup parent from repository -->
</parent>
```

引入 Spring Boot Maven 插件

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <version>3.1.0</version>
        </plugin>
    </plugins>
</build>
```

引入 Spring Boot 提供的 starter

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

引入 Lombok 插件

```xml
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.28</version>
    <optional>true</optional>
</dependency>
```

增加配置文件

此配置文件为连接 MySQL 之后的配置文件，在此之前，没有连接数据库的时候，可以使用内置数据库 H2，进需要引入 H2 的依赖既可

```yaml
spring:
  application:
    name: shorten-service
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url:  jdbc:mysql://localhost:3306/shortenservice?
serverTimezone=UTC&useUnicode=true&characterEncoding=utf-
8&useSSL=false&allowPublicKeyRetrieval=true
    username: root
    password: 123456
    type: com.alibaba.druid.pool.DruidDataSource
    druid:
      initial-size: 20
      min-idle: 20
      max-active: 100
      max-wait: 60000
  jpa:
    hibernate:
      # 上线的时候需要关闭
      ddl-auto: create-drop
    properties:
      hibernate:
        # 开发过程中可以打开
        format_sql: true
        show_sql: true

server:
  port: 8888
```

引入 ORM 框架 JPA

项目前期使用的 ORM 框架为 mybatis-plus，后面为了使得项目更加轻巧，所以使用 JPA 作为项目 ORM 的框架

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

连接 MySQL 数据库

配置文件使用同上，只需要在上面创建对应的数据库就可以了

自定义 Restful 结果封装类

响应封装类

```
public class ResponseResult<T> {
    /**
     * 响应状态码
     */
    private String code;
    /**
     * 响应状态码叙述
     */
    private String message;
    /**
     * 响应封装的数据
     */
    private T data;
}
```

响应工具类

```
package com.qiyao.shortenservice.common;

/**
 * @ClassName ResultUtils
 * @Description 返回结果工具类
 * @Version 1.0.0
 * @Date 2023/08/28
 * @Author LinQi
 */
public class ResultUtils {
    /**
     * 私有化构造参数，使得其返回只可以通过方法的方式返回
     */
    private ResultUtils() {

    }

    /**
     * 执行结果成功（需要返回数据）
     *
     * @param data 数据
     * @param <T>  泛型
     * @return 执行结果包装类
     */
    public static <T> ResponseResult success(T data) {
        return build("200", "message", data);
    }

    /**
     * 执行结果成功（不需要返回数据）
     *
     * @return 执行结果包装类
     */
    public static ResponseResult success() {
        return build("200", "message", null);
    }


    /**
     * 判断是否成功
```

```
     *
     * @param code 状态码
     * @return 包装类
     */
    public static boolean isSuccess(String code) {
        return "200".equals(code);
    }

    /**
     *执行失败，值返回错误信息
     * @param msg 错误信息
     * @return 包装类
     */
    public static ResponseResult failure(String msg){
        return build("500", "message",null);
    }

    /**
     * 执行失败
     *
     * @param code 响应码
     * @param msg  响应叙述
     * @return
     */
    public ResponseResult failure(String code, String msg) {
        return build(code, msg, null);
    }

    public <T> ResponseResult failure(String code, String msg,T data) {
        return build(code, msg, data);
    }

    /**
     * 构造返回结果
     *
     * @param code 响应码
     * @param msg  响应叙述
     * @param data 数据
     * @param <T>  泛型参数
     * @return 返回结果
     */
    public static <T> ResponseResult build(String code, String msg, T data) {
        return new ResponseResult(code, msg, data);
    }

}
```

系统分析与设计

场景分析

功能

- 长链接 -> 短链接 https://www.baidu.com -> t.cn/abcdef
- 访问的时候，进行重定向（301 or 302）用 302 方便统计

非功能（QPS / 数据量 / 接口延迟）

需要问一下面试官，日活，假设是微博，1 亿（100 M）

根据日活进行一个简单的推算

1. 推算写 QPS：假设每个用户平均一天发 0.1 个带链接的微博
   - 平均写 QPS : 100 M * 0.1 / 86400 约等于 100 +
   - 峰值 QPS : 100 * 2 大概 200+
2. 推算读 QPS : 假设每个用户平均每天点击一个链接
   - 平均读 QPS : 100 M * 1 / 86400 约等于 1 K
   - 峰值： 1 K * 2 = 2 K+
3. 推算数据量：
   - 100 M * 0.1 等于 10 M
   - 预估一下每条链接占用 100 Byte ，10 M * 100 B，大概 1 G

假设 1T 硬盘，可以用三年，进行QPS设计

服务设计

一个服务就够了

函数（RPC 接口）设计

- shortKey encode（longUrl）
- longUrl decode（shortUrl）

HTTP 接口

1. POST /shorten
   - 入参：longUrl
   - 出参：shortKey
2. GET /{shortKey}
   - 返回
     - 状态码 302 ，location：longUrl
     - 异常：返回一个降级页面，告诉用户这个短链接不存在

实现细节设计

- 核心算法: 如何将 LongUrl 转换为 shortKey ，不同堵塞 longUrl 不能用同一个 key?

MD 5 (还有其他 hash 算法) 摘要算法

- 优点：效率高
- 缺点：长度比较大，如果截取一部分，但是不能解决冲突问题

随机数

```
while(true) {
    var shortKey = random(longUrl);
    if(dao.notExists(shortKey)) {
        save();
        return;
    }
}
```

- 优点：实现简单
- 缺点：数据越多，实现越慢

雪花算法

- 优点：效率高
- 缺点：长度比较长，如果截取一部分，当时不能解决冲突的问题，时钟回拨问题不好解决

ID 自增 + base 62

> Base62编码是一种将数字、字母大小写都包含的字符串表示形式。它使用了62个字符，分别是0-9、a-z、A-Z，可以作为URL短链接、文件名等场景的字符串表示，相对于16进制或64进制等其他编码，Base62具有更高的可读性和稳定性。
>
> Base62编码的主要应用场景有：
>
> - URL短链接：Base62编码可以将长URL转换为短字符串，提高URL传送效率。
> - 文件命名：Base62编码可以将文件名转换为短字符串，减少文件名——路径的长度，更加有利于文件存储和管理。
> - 防止猜测：Base62编码可以将其他编码方式的数据进行混淆，从而达到一定的防猜测的效果。

5位：62^5=0～9+亿

6位：62^6=0~560+亿

7位：62^7=0～35000+亿

- 优点：效率高
- 缺点：依赖自增 ID ，自增 ID 容易给遍历

存储设计

- 选型 -> 设计结构

方案一：MySQL

字段	类型	备注
id	int	主键
long_url	varchar(2048)	
create_time	timestamp	
expire_time	timestamp	

- uniq:long_url
- index:expire_time

方案二：Redis

shortKey -> longUrl

longUrl -> shortKey

系统分析方法论总结

场景分析

- 功能需求：
  - 一般面试中要询问面试官，或者自己假设后跟面试官确认
  - 如果是工作中多跟业务方交流
- 非功能：
  - 一般分析系统的 QPS,如果没有数据就只能推算，比如通过 日活 + 我们的经验或者行业的经验
    - 1 K - 2 K MySQL / PgSQL 等关系型数据库能顶住
    - 10 k - 50 k, HBase / Cassandra
    - 100 K，Redis / Memcached
    - 更高：分布式方案
  - QPS 场景
    - QPS 看具体接口、具体场景、需要理解业务
    - 如果是娱乐，社交，一般 QPS 都比较高
    - 电商一般是在做活动的时候 QPS 比较高 （双十一淘宝 2022 下单 QPS 大概 40 w，平时的话 几K 不到 1 w）
    - 其他的例如金融，平时没有多少的 QPS
    - QPS 的峰值更加重要

服务设计

- 微服务架构的优势
  - 不同业务之间相互独立管理，独立演进，独立缩容扩容
  - 技术异构
  - 系统解耦
  - 迭代和部署快
  - 故障隔离，资源隔离

存储设计

先做技术选型

一般是设计数据结构（表 结构或者 kv 结构），索引等，考虑高效写入以及查询

系统优化

- 性能
- 扩展性

项目核心功能实现

Base 62 算法实现

工具类开发

```
public class Base62Utils {
    private static final String BASE62 =
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";

    private Base62Utils() {}

    /**
     * Base 62 算法 id 转为短链接 key 实现
     * @param id 长链接 id
     * @return  长链接对应的短链接键值
     */
    public static String idToShortKey(long id){
        StringBuilder stringBuilder = new StringBuilder();

        // 当 id 大于 0 的时候，一致执行stringBuilder 添加操作，不断添加字符串
        while (id > 0){
            stringBuilder.append(BASE62.charAt((int)(id % 62)));
            id = id / 62;
        }

        while(stringBuilder.length() < 6){
            stringBuilder.append(0);
        }

        return stringBuilder.reverse().toString();
    }

    /**
     * Base 62 算法 id 短链接键值 转化为 ID
     * @param shortKey
     * @return
     */
```

```
    public static long shortKeyToId(String shortKey){
        long id = 0;

        for(int i = 0;i < shortKey.length() ;i++){
            id = id * 62 + BASE62.indexOf(shortKey.charAt(i));
        }

        return  id;
    }
}
```

工具类测试

```
package com.qiyao.shortenservice.utils;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

import static org.junit.jupiter.api.Assertions.*;

@SpringBootTest
class Base62UtilsTest {
    /**
     * 短链接键值转化为 id
     */
    @Test
    void testShortKeyToId() {
        System.out.println(Base62Utils.shortKeyToId("000001"));
        System.out.println(Base62Utils.shortKeyToId("00fxSJ"));
        System.out.println(Base62Utils.shortKeyToId("Aiq5v2IOhhA"));



    }

    /**
     * id 转化为短链接键值
     */
    @Test
    void testIdToShortKey() {
        System.out.println(Base62Utils.idToShortKey(1));
        System.out.println(Base62Utils.idToShortKey(9999999));
        System.out.println(Base62Utils.idToShortKey(900000000000000000L));
    }
}
```

数据访问层开发

UrlMap 实体类开发

```
@Entity
@Table(name = "t_url_map", indexes = {@Index(columnList = "longUrl",unique = true),
@Index(columnList = "expireTime",unique = false)})
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class UrlMap {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String longUrl;

    private Instant expireTime;

    @CreationTimestamp
```

```
    private Instant creationTime;

}
```

Dao 层 UrlMapDao 创建

```
public interface UrlMapDao extends CrudRepository<UrlMap, Long> {
    /**
     * 通过长链接主键 查找 UrlMap 实体类
     * @param longUrl
     * @return
     */
    UrlMap findFirstByLongUrl(String longUrl);
}
```

长链接转短链接接口实现

Service 层接口开放 -- 实现接口

```
public interface UrlMapService {
    /**
     * 解码
     * @param longUrl
     * @return
     */
    String encode(String longUrl);
}
```

Service 层接口开发 -- 实现类

```
@Service
@Slf4j
public class UrlMapServiceImpl implements UrlMapService {
    /**
     * UrlMap 接口注入
     */
    @Autowired
    private UrlMapDao urlMapDao;

    /**
     * 为长链接创建对应的键值
     * @param longUrl 需要进行短链接 key 编码的长链接
     * @return 短链接的键值
     */
    @Override
    public String encode(String longUrl) {
        UrlMap urlMap = urlMapDao.findFirstByLongUrl(longUrl);

        if (urlMap == null) {
            urlMap = urlMapDao.save(UrlMap.builder()
                    .longUrl(longUrl)
                    .expireTime(Instant.now().plus(30, ChronoUnit.DAYS))
                    .build()
            );
            log.info("create urlMap:{}", urlMap);
        }
        return Base62Utils.idToShortKey(urlMap.getId());
    }
}
```

Controller 层开发

```java
@RestController
@RequestMapping("/urlmap")
public class UrlMapController {

    private static final String DOMAIN = "http://127.0.0.1:8888/";

    @Autowired
    private UrlMapService urlMapService;

    @PostMapping("/shorten")
    public ResponseResult<Map> shorten(@RequestParam("longUrl") String longUrl) {
        String encode = urlMapService.encode(longUrl);
        return ResultUtils.success(Map.of("shortKey", encode, "shortUrl", DOMAIN + encode));
    }
}
```

接口测试

```
###
POST http://localhost:8888/urlmap/shorten?longUrl=www.baidu.com
```

```
{
  "code": "200",
  "message": "message",
  "data": {
    "shortUrl": "http://127.0.0.1:8888/000001",
    "shortKey": "000001"
  }
}
```

短链接重定向接口开发

Service 层接口开发 -- 实现接口

```java
package com.qiyao.shortenservice.service;

import java.util.Optional;

/**
 * @ClassName UrlMapService
 * @Description UrlMap
 * @Version 1.0.0
 * @Author LinQi
 * @Date 2023/08/29
 */
public interface UrlMapService {
    /**
     * 编码
     * @param longUrl 需要进行短链接 Key 值编码的长链接
     * @return 编码后的短链接 Key
     */
    String encode(String longUrl);

    /**
     * 解码
     * @param shortKey 需要进行解码的短链接 Key 值
     * @return
     */
    Optional<String> decode(String shortKey);
```

```
}
```

Service 层接口开发 -- 实现类

```java
package com.qiyao.shortenservice.service.impl;

import com.qiyao.shortenservice.dao.UrlMapDao;
import com.qiyao.shortenservice.model.UrlMap;
import com.qiyao.shortenservice.service.UrlMapService;
import com.qiyao.shortenservice.utils.Base62Utils;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.util.Optional;

/**
 * @ClassName UrlMapService
 * @Description
 * @Version 1.0.0
 * @Author LinQi
 * @Date 2023/08/29
 */
@Service
@Slf4j
public class UrlMapServiceImpl implements UrlMapService {
    /**
     * UrlMap 接口注入
     */
    @Autowired
    private UrlMapDao urlMapDao;

    /**
     * 为长链接创建对应的键值
     *
     * @param longUrl 需要进行短链接 key 编码的长链接
     * @return 短链接的键值
     */
    @Override
    public String encode(String longUrl) {
        UrlMap urlMap = urlMapDao.findFirstByLongUrl(longUrl);

        if (urlMap == null) {
            urlMap = urlMapDao.save(UrlMap.builder()
                    .longUrl(longUrl)
                    .expireTime(Instant.now().plus(30, ChronoUnit.DAYS))
                    .build()
            );
            log.info("create urlMap:{}", urlMap);
        }
        return Base62Utils.idToShortKey(urlMap.getId());
    }

    /**
     * 短链接重定向开发
     * @param shortKey 需要进行解码的短链接 Key 值
     * @return 对应的长链接
     */
    @Override
    public Optional<String> decode(String shortKey) {
        long id = Base62Utils.shortKeyToId(shortKey);
        return urlMapDao.findById(id).map(UrlMap::getLongUrl);
    }
}
```

Controller 层开发

```
package com.qiyao.shortenservice.controller;

import com.qiyao.shortenservice.common.ResponseResult;
import com.qiyao.shortenservice.common.ResultUtils;
import com.qiyao.shortenservice.service.UrlMapService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.view.RedirectView;

import java.util.Map;

/**
 * @ClassName UrlMapController
 * @Description
 * @Version 1.0.0
 * @Author LinQi
 * @Date 2023/08/29
 */
@RestController
@RequestMapping("/urlmap")
public class UrlMapController {

    private static final String DOMAIN = "http://127.0.0.1:8888/urlmap/";

    @Autowired
    private UrlMapService urlMapService;

    /**
     * 前端传入一个长链接,后端根据长链接生成对应的短链接键值,并且将短链接键值接入到对应的 url 后面
     *
     * @param longUrl 对应的长链接 http://localhost:8888/urlmap/shorten?longUrl="www.baidu.com"
     * @return 对应生成的短链接 http://127.0.0.1:8888/000001
     */
    @PostMapping("/shorten")
    public ResponseResult<Map> shorten(@RequestParam("longUrl") String longUrl) {
        //非空检验,避免传入空的长链接参数导致错误
        if (longUrl == null) {
            throw new RuntimeException("Link parameter exception: The passed long link
parameter is abnormal.");
        }

        String encode = urlMapService.encode(longUrl);
        return ResultUtils.success(Map.of("shortKey", encode, "shortUrl", DOMAIN + encode));
    }

    @GetMapping("/{shortKey}")
    public RedirectView redirect(@PathVariable("shortKey") String shortKey) {
        return urlMapService.decode(shortKey).map(RedirectView::new)
                .orElse(new RedirectView(DOMAIN + "/sorry"));
    }

    @GetMapping("/sorry")
    public String sorry() {
        return "抱歉，未找到页面！";
    }
}
```

接口测试

```
POST http://localhost:8888/urlmap/shorten?longUrl=http://www.baidu.com

<> 2023-08-29T183439.200.json
```

```
{
  "code": "200",
  "message": "message",
  "data": {
    "shortUrl": "http://127.0.0.1:8888/urlmap/000001",
    "shortKey": "000001"
  }
}
```

```
GET http://127.0.0.1:8888/urlmap/000001

<> 2023-08-29T183458.200.html
```

```
http://127.0.0.1:8888/urlmap/000001

Redirections: Disable
-> http://www.baidu.com/

HTTP/1.1 200 OK
Content-Type: text/html
Server: bfe
Date: Tue, 29 Aug 2023 10:40:22 GMT
```

此处如果返回成功的话,会返回一个百度的 HTML ,这里就不做展示

然后下面就是测试失败的情况,如下图所示,为失败后的页面

```
GET http://localhost:8888/urlmap/100092

<> 2023-08-29T184324.200.txt
```

```
http://localhost:8888/urlmap/100092

Redirections: Disable
-> http://127.0.0.1:8888/urlmap//sorry

HTTP/1.1 200
Content-Type: text/plain;charset=UTF-8
Content-Length: 27
Date: Tue, 29 Aug 2023 10:43:24 GMT
Keep-Alive: timeout=60
Connection: keep-alive

抱歉，未找到页面!

Response code: 200; Time: 23ms (23 ms); Content length: 9 bytes (9 B)
```

定时任务清理过期数据

创建一个定时任务的类,并且将其注入 Spring 容器

```
@Component
@Slf4j
public class ClearJob {
    @Autowired
    private UrlMapDao urlMapDao;
```

```
    /**
     * 单机版的定时任务
     */
    @Scheduled(fixedRate = 5 * 1000)
    public void clear() {
        log.info("clear job");
        List<UrlMap> list = urlMapDao.findByExpireTimeBefore(Instant.now().plus(30,
ChronoUnit.DAYS));
        for (UrlMap urlMap : list) {
            log.info("delete url map {}", urlMap);
            urlMapDao.deleteById(urlMap.getId());
        }
    }


}
```

开启定时任务

```
@SpringBootApplication
@EnableScheduling
public class ShortenServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(ShortenServiceApplication.class, args);
    }

}
```

单元测试

单元测试是软件开发中的一种测试方法,旨在对软件的最小功能单元(通常是一个函数、方法或类)进行测试，用于验证其是否可以按照预期工作。单元测试是一种自动化的测试方式，具有以下几个重要的作用：

1. **发现早期问题:**单元测试可以在代码编写的早期发现问题和错误，用来帮助开发人员及时修复缺陷，从而降低修复成本和影响。
2. **确保代码质量：** 单元测试可以确保每个功能单元的正确性以及稳定性，从而提高代码的质量和可靠性。
3. **代码重构：** 单元测试提供了一种安全的重构环境,当代码需要进行重构的时候,可以通过单元测试来验证修改后的代码是否正确.
4. **文档作用：** 单元测试可以作对功能单元的文档,开发人员可以通过阅读测试代码进而了解功能单元的用法和预期行为.
5. **支持持续集成（CI）：** 单元测试是持续集成的重要组成部分,每次代码提交后自动运行单元测试从而可以确保新代码没有引入新的问题.
6. **增加信心：** 单元测试可以给开发人员带来信心,确保他们的代码在改动之后仍然可以正确运行,减少因为改动而引起的不确定性
7. **快速反馈：** 单元测试是自动化的,可以快速运行,当开发人员提交代码的时候,可以立刻获得测试结果,从而帮助它们快速定位问题
8. **减少回归测试：** 单元测试的作用在于隔离每个功能单元,如果一个单元的测试通过,那么其可以在进行回归测试的时候不再重复这个单元

如何快速编写单元测试

3 A 法是一个用于编写单元测试的模式,他包括 "Arrange(安排) "、 “Act(操作)” 和 “Assert（断言）” 三个步骤，帮助开发人员快速组织和编写单元测试用例。以下是针对每个步骤的详细解释：

1. **Arrange(安排)**：在这个步骤中，需要进行一些准备工作，将测试环境设置为想要的状态。这可能设计创建对象、设置模拟数据、初始化模拟变量等。这个步骤主要是为测试用例提供一个正确的初始状态。
2. **Act（操作）：** 在这个步骤中，执行需要测试的代码，也就是想要测试的功能单元。这可能是调用一个函数、执行一个方法或者执行一些操作。这个步骤是为了模拟实际使用情况下的操作。
3. **Assert（断言）：** 在这个步骤中，会验证预期的结果，需要使用断言来比较实际的输出和期望的输出是否一致。如果预期和实际不符，测试会失败。这个步骤可以帮助确认代码是否按照预期工作。

编写 Spring Boot 单元测试

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

核心注解

- @SpringBootTest
- @Test

断言工具类： Assertions

编写 UrlMap 单元测试类

```java
@SpringBootTest
class UrlMapServiceImplTest {
    @Autowired
    private UrlMapService urlMapService;

    @Test
    void tryEncode() {
        //Arrange

        //Act
        String shortKey = urlMapService.encode("https://www.baidu.com");

        //Assert
        Assertions.assertNotNull(shortKey);
    }

    @Test
    void tryDecode() {
        //Arrange
        String shortKey = urlMapService.encode("https://www.baidu.com");

        //Act
        Optional<String> longUrl = urlMapService.decode(shortKey);

        //Assert
        Assertions.assertEquals("https://www.baidu.com", longUrl.get());
    }
}
```

系统性能优化方案设计

系统提速方案一：引入缓存

> 缓存是一种将数据存储在更快速、更加易于访问的位置，用于将来的访问中可以更快地检索数据的技术或者策略。缓存在计算机系统种和应用程序中被广泛应用，旨在提高系统的性能、响应速度以及资源利用率。可以减少对慢速数据源（磁盘、网络）的频繁访问，从而加速数据的读取与处理。
>
> 缓存的工作原理类似于人们日常生活中使用的记忆方法，如将常用的物品存放在容易取用的地方，以便于快速找到和使用。在计算机系统中，缓存可以是硬件缓存（如处理器缓存）或者软件缓存（如应用程序中的缓存层）。
>
> 缓存的使用有助于解决以下问题：
>
> 高并发访问：在高并发的情况下，数据源可能会受到较大的负载，为了减轻数据源的压力，提高系统的性能以及响应速度。
>
> 频繁访问：对于频繁被访问的数据，通过缓存减少重复的数据读取，提高效率
>
> 数据计算：对于一些需要复杂计算的数据，将计算机结果缓存起来可以节省计算时间和资源
>
> 数据共享：缓存可以在不同的组件、模块或服务之间共享数据，提高数据的可用性和共享性
>
> 离线访问：缓存可以在断网或者无法连接到数据源的情况下，仍然提高某些数据的访问能力。

短链接是读多写少的场景，访问 DB 需要经过网络 IO 和 磁盘 IO ,开销比较大

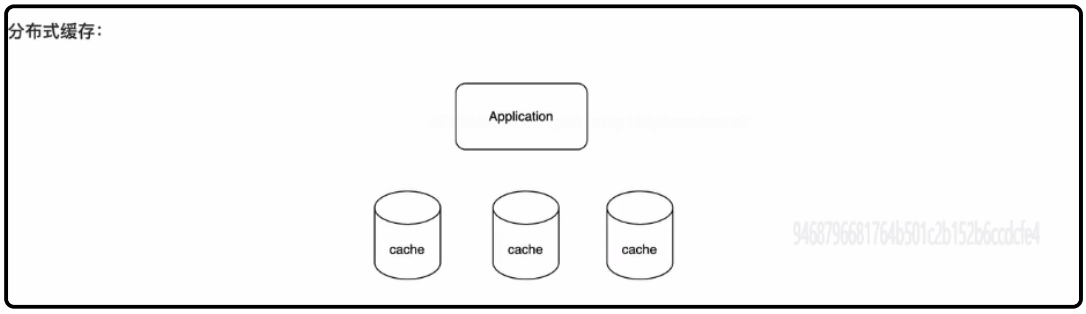
其实不需要每次都访问数据库，因为可以引入缓存，比如 localcache / redis / memchaed

Application ---> DB

Application ---> cache ---> DB

分布式缓存 与 本次缓存分析

分布式缓存



1. **概念**：分布式缓存是一种将缓存数据分布在多个服务器节点上的缓存系统，用于存储和管理大量数据

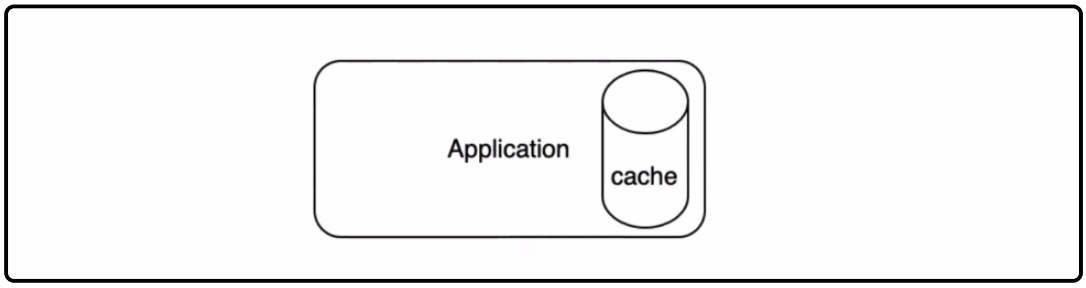
2. **优点**：

   - 可扩展性：分布式缓存可以通过增加节点来实现水平扩展，用来应对大规模的数据和高并发访问
   - 高可用性：分布式缓存通常采用复制和备份机制，确保即使有节点故障，仍然可以提供可靠的缓存服务。
   - 跨节点共享：多个应用实例可以共享同一分布式缓存，提高数据共享和协作能力
   - 灵活的存储后端：分布式缓存可以支持多种后端存储，如内存、磁盘、数据库等

3. 缺点：

- 复杂性：部署、配置和管理分布式缓存系统可能较为复杂，需要考虑分布式系统的一些挑战，比如一致性、网络延迟等。
- 性能开销：分布式缓存通常需要在网络上进行数据传输，可能引入一些性能开销。

本地缓存



1. 概念：本地缓存是将换出数据存储在应用程序的本地内存中，用于临时保存常用的数据。
2. 优点：
   - 简单性：本地缓存相对简单，不需要额外搭建的分布式缓存系统。
   - 低延迟：由于数据存储在本地内存中，本地缓存通常具有低延迟的读取速度。
   - 少量数据：本地缓存适用于存储相对较小的数据量，不需要进行分布式存储和管理。
3. 缺点：
   - 有限的扩展性：本地缓存只能在单个应用实例内使用，无法满足多实例和分布式应用的需求
   - 数据一致性：不同应用实例的本地缓存可能存在数据不一致的问题，需要额外的机制来解决。

应用场景

- 如果应用需要存储大量数据，需要水平扩展以及支持高并发访问，那么分布式缓存是一个更加合适的选择，比如 Redis 、Memcached 等
- 如果应用需要快速读取热点的数据，对数据一致性的要求不是很高，但是对于性能能要求比较高，可以考虑使用本地缓存，如 Guava Cache 、Ehcache 等

高性能本地缓存 Guava Cache 介绍

Guava Cache是Google Guava库中的一个组件，用于提供本地内存缓存的功能。Guava Cache旨在帮助开发人员在应用程序中实现简单且高效的本地缓存，以加速数据的读取和处理。它适用于那些需要快速访问、频繁读取的数据，如配置信息、计算结果等。

以下是Guava Cache的一些特点和功能：

1. **自动加载和刷新：** Guava Cache支持自动加载缓存项，当缓存中没有某个键对应的值时，可以提供加载函数来生成该值。此外，它还支持定期刷新缓存项，以确保缓存中的数据保持最新。
2. **缓存回收策略：** Guava Cache支持多种缓存回收策略，包括基于大小（元素个数）、权重（元素大小）、时间（过期时间）等。开发人员可以根据需求选择合适的回收策略，以控制缓存的大小和内存占用。
3. **并发支持：** Guava Cache内部使用了并发数据结构，因此它在多线程环境下能够高效地处理并发访问，提供线程安全的缓存操作。
4. **统计信息和监控：** Guava Cache提供了统计信息，可以了解缓存的命中率、命中次数、未命中次数等。这有助于开发人员监控和优化缓存的使用情况。
5. **可配置性：** Guava Cache提供了丰富的配置选项，可以根据具体需求进行配置，如缓存的最大大小、过期时间、回收策略等。

本地缓存的使用案例

```xml
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>32.1.1-jre</version>
</dependency>
```

```java
package com.qiyao.shortenservice.demo;

import com.google.common.cache.Cache;
import com.google.common.cache.CacheBuilder;

/**
 * @ClassName GuavaCacheExample
 * @Description GuavaCache 缓存使用示例
 * @Version 1.0.0
 * @Author LinQi
 * @Date 2023/08/30
 */

public class GuavaCacheExample {
    public static void main(String[] args) {
        Cache<String, String> cache = CacheBuilder.newBuilder()
                // 设置最大缓存大小
                .maximumSize(100)
                .build();
        // 将数据放入缓存
        cache.put("key1", "value1");
        cache.put("key2", "value2");

        // 从缓存中读取数据
        String value = cache.getIfPresent("key1");
        System.out.println("key1:" + value);
    }
}
```

Guava Cache 使用

主要是在 Service 层的实现接口中修改

```java
package com.qiyao.shortenservice.service.impl;

import com.google.common.cache.CacheBuilder;
import com.google.common.cache.CacheLoader;
import com.google.common.cache.LoadingCache;
import com.qiyao.shortenservice.dao.UrlMapDao;
import com.qiyao.shortenservice.model.UrlMap;
import com.qiyao.shortenservice.service.UrlMapService;
import com.qiyao.shortenservice.utils.Base62Utils;
import jakarta.annotation.PostConstruct;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.time.Instant;
import java.time.temporal.ChronoUnit;
import java.util.Optional;

/**
 * @ClassName UrlMapService
 * @Description
 * @Version 1.0.0
 * @Author LinQi
 * @Date 2023/08/29
 */
@Service
@Slf4j
public class UrlMapServiceImpl implements UrlMapService {
    /**
     * UrlMap 接口注入
     */
    @Autowired
```

```
    private UrlMapDao urlMapDao;
    /**
     * 缓存对象注入
     */
    private LoadingCache<String, String> loadingCache;

    @PostConstruct
    public void init() {
        CacheLoader<String, String> cacheLoader = new CacheLoader<>() {
            @Override
            public String load(String shortKey) {
                long id = Base62Utils.shortKeyToId(shortKey);
                log.info("Loading cache {}", shortKey);
                return urlMapDao.findById(id).map(UrlMap::getLongUrl).orElse(null);
            }
        };

        loadingCache = CacheBuilder.newBuilder()
                // 设置最大缓存大小
                .maximumSize(100000)
                .build(cacheLoader);
    }

    /**
     * 为长链接创建对应的键值
     *
     * @param longUrl 需要进行短链接 key 编码的长链接
     * @return 短链接的键值
     */
    @Override
    public String encode(String longUrl) {
        UrlMap urlMap = urlMapDao.findFirstByLongUrl(longUrl);

        if (urlMap == null) {
            urlMap = urlMapDao.save(UrlMap.builder()
                    .longUrl(longUrl)
                    .expireTime(Instant.now().plus(30, ChronoUnit.DAYS))
                    .build()
            );
            log.info("create urlMap:{}", urlMap);
        }
        return Base62Utils.idToShortKey(urlMap.getId());
    }

    /**
     * 短链接重定向开发
     *
     * @param shortKey 需要进行解码的短链接 Key 值
     * @return 对应的长链接
     */
    @Override
    public Optional<String> decode(String shortKey) {
        return  Optional.ofNullable(loadingCache.getUnchecked(shortKey));
    }
}
```

接口测试

```
###
GET http://127.0.0.1:8888/urlmap/000001

<> 2023-08-30T173820.200.html
<> 2023-08-30T173641.200.html
<> 2023-08-30T173619.200.html
<> 2023-08-30T173604.200.html
```

```
###
GET http://127.0.0.1:8888/urlmap/0000092

<> 2023-08-30T173901.500.json
<> 2023-08-30T173808.500.json
```

由于没有设置访问的缓存处理，所以当输入一个错误链接的时候，这里会报一个 500 的错误

系统提速方案二：异地多活

异地多活（Geographical Redundancy or Geo-Redundancy）是一种架构和部署策略，旨在在不同地理位置（地区、城市、国家等）之间建立多个数据中心或服务副本，以提供高可用性、容灾备份和更好的性能。

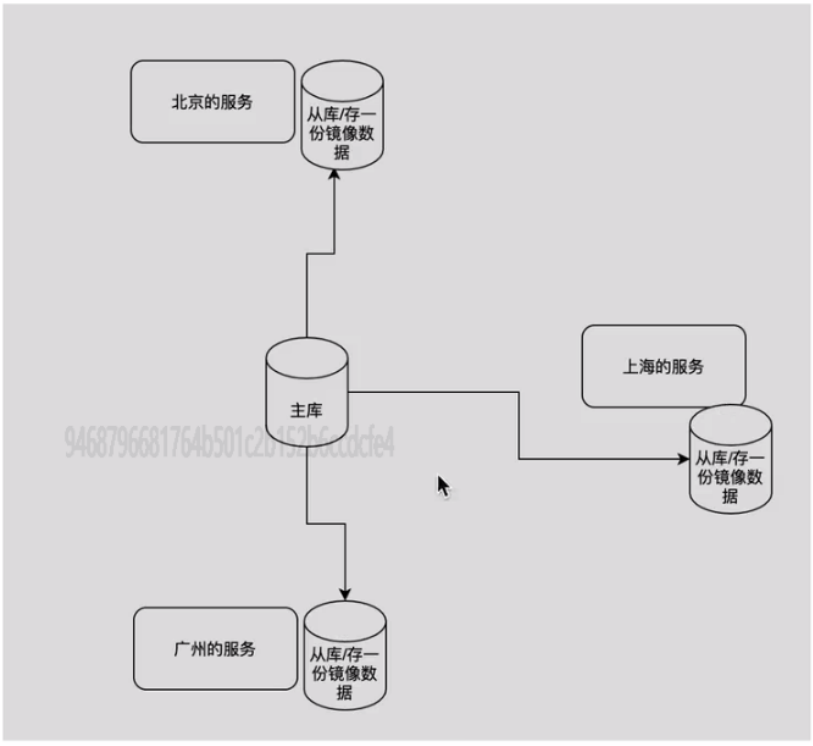
异地多活的主要思想是将系统的关键组件、数据和服务分布在不同的地理位置上，以防止单一地点的故障对整个系统的影响。这种架构可以在各种应用和服务中应用，包括网络服务、分布式应用、数据库系统等。

异地多活的优势和特点包括：

1. **高可用性：** 异地多活架构可以在一个地理区域的数据中心发生故障时，自动切换到另一个地理区域的数据中心，从而保证系统的持续可用性。
2. **容灾备份：** 数据在多个地理位置上备份，即使一个数据中心发生灾难性故障，数据仍然可以从其他地方恢复。
3. **降低延迟：** 用户可以从距离更近的数据中心获取数据，从而降低数据传输的延迟，提供更好的性能和用户体验。
4. **法规和合规要求：** 某些法规和合规要求可能需要数据在特定地理位置存储和处理，异地多活可以满足这些要求。
5. **全球分布：** 对于全球用户的应用，异地多活可以使数据和服务更接近用户，提供更快速的访问速度。

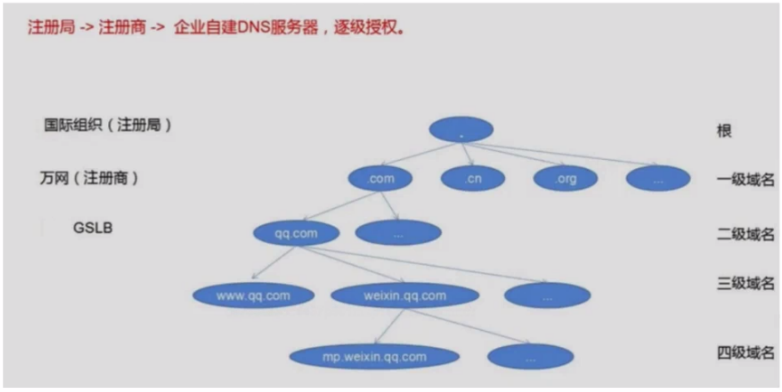
在实施异地多活时，需要考虑以下方面：

- **数据同步：** 数据在多个地理位置上需要保持同步，以确保一致性和可用性。常用的方法包括同步数据库、使用分布式存储系统，或者使用专门的数据同步工具。
- **负载均衡：** 需要实现跨多个地理位置的负载均衡，确保流量分配均匀，避免单一地点过载。
- **故障切换：** 当一个地理位置的数据中心发生故障时，需要有自动化的故障切换机制，将流量切换到备用地理位置。
- **数据隔离和安全性：** 需要考虑数据在多个地理位置之间的隔离和安全性，确保敏感数据不被未授权访问。
- **成本和复杂性：** 异地多活涉及多个数据中心和资源，可能会增加成本和复杂性。需要权衡利弊，根据实际需求进行规划。



负载均衡的问题？流量入口不一样怎么处理？全局负载均衡（GSLB），大厂做的

> **Global Server Load Balancing**
>
> 中文:全局负载均衡
>
> **SLB(Server load balancing)**是对集群内物理主机的负载均衡，而**GSLB**是对物理集群的负载均衡。
>
> 这里的负载均衡可能不只是简单的流量均匀分配,而是会根据策略的不同实现不同场景的应用交付。
>
> **GSLB**是依赖于用户和实际部署环境的互联网资源分发技术，不同的目的对应着一系列不同的技术实现。



系统扩展方案：分库分表分析与设计

假如用mysql，可能一开始估算不对，或者业务发展太快，怎么办？

> 分库分表是一种数据库架构设计方法，用于解决大规模数据存储和处理的问题。它将一个大型数据库拆分成多个小型数据库（分库），同时将每个小型数据库中的数据表进一步拆分成多个子表（分表），从而实现数据的水平分割和分布式存储。
>
> 分库分表的主要目标是提高数据库的性能、可扩展性和负载均衡能力，以应对大量数据和高并发访问的情况。这种架构在互联网应用、大数据应用和分布式系统中广泛应用。
>
> **分库分表的作用：**
>
> 1. **提高性能：** 将数据分散到多个小型数据库中，每个数据库只处理部分数据，可以提高查询和写入操作的性能。
> 2. **可扩展性：** 当数据量增大时，可以通过添加新的数据库节点和表来实现水平扩展，无需对整个数据库进行升级或迁移。
> 3. **负载均衡：** 分库分表可以将负载分摊到不同的数据库和表中，避免单一数据库成为瓶颈。
>
> **分库分表的挑战和注意事项：**
>
> 1. **数据一致性：** 分库分表会引入数据一致性的问题，需要通过一些机制来保证数据的一致性，如分布式事务。
> 2. **跨库查询：** 跨库查询可能会变得复杂，需要考虑如何处理分布在不同数据库和表中的数据。
> 3. **业务拆分：** 需要合理划分数据库和表的边界，根据业务需求进行拆分，避免出现过于复杂的关联查询。
> 4. **分表键选择：** 选择合适的分表键可以避免数据倾斜和不均匀的问题。

如何选取合理的 shardingKey

分库分表一般需要根据查询的维度来做，选取合理的shardingKey。

如何选取shardingKey?

- 用id或者shortKey？查询接口很容易实现，实现写接口的时候，只能遍历查询是否有重复数据
- 用longUrl？写接口很容易实现，但是查询的时候只能遍历
- 存两份数据？数据是一样的，分库的规则不同；理论上是可以，但是存储成本比较大，还有数据同步比较麻烦。

基因法选取 shardingKey

基因法

shardingKey = hash(longUrl) % 62 ; //也可以模其它的，62^n 就不会浪费

shortKey = shardingKey + shortKey;

做路由的时候，如果是longUrl参数查询，就hash取模做路由；

如果是shortKey参数的查询，取第一位做路由。

基因法的优缺点：

优点：不需要增加太多存储，也不需要考虑数据一致性问题

缺点：62不够用（拼两位也就是62^2=3844）