日常问题总结

工作中总是会遇到各种各样的问题，只有总结下来才是一笔财富。

fastjson的一些技巧

在工作中总是遇到给前端的字段需要是下划线的，这时候可以通过全局配置来实现。

添加这个配置bean后，所有的http请求都会进行转换，即会将返回参数改为下划线

```
@Bean
public HttpMessageConverters fastJsonHttpMessageConverters(){
    FastJsonHttpMessageConverter converter = new FastJsonHttpMessageConverter();
    FastJsonConfig fastJsonConfig = new FastJsonConfig();
    // 格式化输出，也就是换行等处理
    fastJsonConfig.setSerializerFeatures(SerializerFeature.PrettyFormat);
    SerializeConfig config = new SerializeConfig();
    // 转为下划线
    config.propertyNamingStrategy = PropertyNamingStrategy.SnakeCase;
    fastJsonConfig.setSerializeConfig(config);
    converter.setFastJsonConfig(fastJsonConfig);
    return new HttpMessageConverters(converter);
}
CamelCase策略，Java对象属性：personId，序列化后属性：persionId

    PascalCase策略，Java对象属性：personId，序列化后属性：PersonId
```

```
    SnakeCase策略，Java对象属性：personId，序列化后属性：person_id

    KebabCase策略，Java对象属性：personId，序列化后属性：person-id
```

例子：

```
@Data
public class TestResponse {
    private String userName;
    private String teacherName;
}
```

```
@RequestMapping("/group/test")
    public Object test(HttpServletRequest request, HttpServletResponse response)
{
        TestResponse response1 = new TestResponse();
        response1.setUserName("zt");
        response1.setTeacherName("zt");
        return response1;
    }
// 返回结果就自动转为了下划线了
{
  "teacher_name": "zt",
  "user_name": "zt"
}
```

除了这种全局配置的方式，也可以进行代码层面的配置

转为string

```
 @RequestMapping("/test")
    public String test(HttpServletRequest request, HttpServletResponse response)
{
        TestResponse response1 = new TestResponse();
        response1.setUserName("zt");
        response1.setTeacherName("zt");
        SerializeConfig config = new SerializeConfig();
        config.propertyNamingStrategy = PropertyNamingStrategy.SnakeCase;
        // 返回的json就是下划线的
        return JSON.toJSONString(response1, config);
    }
{"teacher_name":"zt","user_name":"zt"}
```

string转为对象

```
 @RequestMapping("/test2")
    public Object test2(HttpServletRequest request, HttpServletResponse
response) {
        String response1 = "{\"teacherName\":\"zt\",\"userName\":\"zt\"}";
        // 转为下划线
        ParserConfig parserConfig =  new  ParserConfig();
        parserConfig.propertyNamingStrategy = PropertyNamingStrategy.SnakeCase;
        return JSON.parseObject(response1, TestResponse.class, parserConfig);
    }
// 很奇怪应该不是这样的
{
    "userName": "zt",
    "teacherName": "zt"
}
```

这个本来应该也是的，但是现在有点尴尬，不得行

Java 中是“值传递”

```
java中方法参数传递方式是按值传递,只不过值不同。
如果参数是基本类型，传递的是基本类型的字面量值的拷贝。
如果参数是引用类型，传递的是该参量所引用的对象在堆中地址值的拷贝。
```

举例:

```
package com.zt.javastudy.grammar;

/**
 * @author zhengtao
 * @description java中方法参数传递方式是按值传递。
 * 如果参数是基本类型，传递的是基本类型的字面量值的拷贝。
 * 如果参数是引用类型，传递的是该参量所引用的对象在堆中地址值的拷贝。
 * @date 2021/4/25
 */
public class QuoteStudy {
    public static void main(String[] args) {
        int a = 0;
        add(a);
        System.out.println(a);
        String b = "hello";
        add(b);
        System.out.println(b);
        StringBuilder c = new StringBuilder("hello");
        add(c);
        System.out.println(c);
        StringBuilder d = new StringBuilder("hello");
        move(d);
        System.out.println(d);
    }

    /**
     * 基本类型传递的值的拷贝，所以不影响原值
     * @param a
     */
```

```
    private static void add(int a){
        a = 1;
    }

    /**
     * 对象传递的是对象的引用的拷贝，所以如果改变对象的属性是可以改变，如果将引用赋值给另一个对象，则不会改变原对象的引用
     * @param b
     */
    private static void add(String b){
        // 在字符中 = ，就相当于重新new对象，因为string类型是不可变的，等价于b = new String("helloWorld")
        b = "helloWorld";
    }

    /**
     * 对象传递的是对象的引用的拷贝,改变对象属性可以成功
     * @param c
     */
    private static void add(StringBuilder c){
       c = c.append("world!");
    }

    /**
     * 改变引用的指向，不成功
     * @param c
     */
    private static void move(StringBuilder c){
        c = new StringBuilder("helloworld!");
    }
}
// 结果
0
hello
helloworld!
hello
```

@Autowired 的一点发现

在日常写代码中基本上都是，写**一个**service，写**一个**impl，然后将@Service注解加在impl类上，在代码中直接使用@Autowired 自动注入。

@autowired注释可以对类成员变量、方法、构造函数进行标注，完成自动装配功能。@autowired查找bean首先是先通过**byType**查，如果发现找到有很多**bean**，则按照byName方式对比获取，若有名称一样的则可以加上@Qualifier("XXX")配置使用。

所以说当一个service只有一个一个impl实现时，自动注入根据byType发现只有一个实现，所以就能正确进行装配，但是如果有多个实现则会报错。

```
public interface TestService {
    void test();
}
@Service
public class TestServiceImpl implements TestService {
    @Override
```

```
    public void test() {
        System.out.println("实现1");
    }
}
@Service
public class TestServiceI2mpl implements TestService {
    @Override
    public void test() {
        System.out.println("实现2");
    }
}
```

这时可以使用@Qualifier注解来完成正确的装配

```
@Slf4j
@RunWith(SpringRunner.class)
@SpringBootTest()
public class AopStudyTest {
    // 多个实现类，使用@Qudalifier使用byName注入
    @Qualifier("testServiceImpl")
    @Autowired
    private TestService testService;
    @Qualifier("testServiceI2mpl")
    @Autowired
    private TestService testService2;
    @Test
    public void testService(){
        testService.test();
        testService2.test();
    }

}
```

公司的一套框架

web容器同步请求

Web容器（比如tomcat）默认情况下会为每个请求分配一个请求处理线程（在tomcat7/8中，能够同时处理到达的请求的线程数量默认为200），默认情况下，在响应完成前，该线程资源都不会被释放。如图所示：

处理HTTP请求和执行具体业务代码的线程是同一个线程！

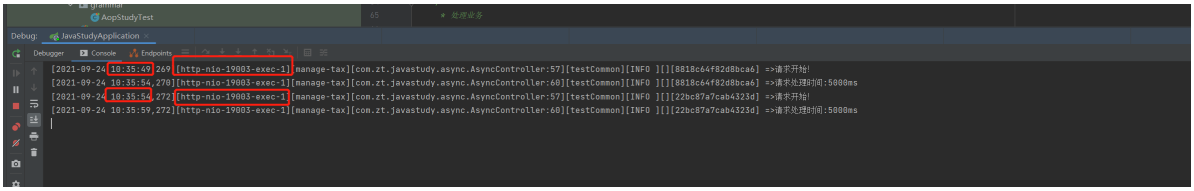
如果业务代码处理时间比较长，那么请求处理线程将一直被占用，直到任务结束，这种情况下，随着并发请求数量的增加，将可能导致处理请求线程全部被占用，此时tomcat会将后来的请求堆积到内部阻塞队列容器中，如果存放请求的阻塞队列也满了，那么后续的进来请求将会遭遇拒绝服务，直到有线程资源可以处理请求为止。

实践是检验真理的唯一标准 将工作线程设为1，方便测试

```yaml
server:
  port: 19003
  tomcat:
    uri-encoding: UTF-8
    max-threads: 1 #最大工作线程数量
    min-spare-threads: 1 #最小工作线程数量
    #max-connections: 10000 #一瞬间最大支持的并发的连接数
    accept-count: 1 #等待队列长度
```

```java
    @RequestMapping("/testCommon")
    public String testCommon() throws InterruptedException {
        log.info("请求开始！");
        start = System.currentTimeMillis();
        Thread.sleep(5000);
        log.info("请求处理时间:{}ms", (System.currentTimeMillis() - start));
        return "hello world!";
    }
```

返回结果为：



很明显的看到当有两个请求过来时，第二个会阻塞，直到第一个请求完成后才会开始处理，而且执行请求的线程和处理业务的线程是同一个线程。

web容器异步请求

有同步请求当然就有异步请求，Servlet 3.0开始支持异步处理请求。在接收到请求之后，**Servlet线程可以将耗时的操作委派给另一个线程来完成，自己在不生成响应的情况下返回至容器**，以便能处理另一个请求。此时当前请求的响应将被延后，在异步处理完成后时再对客户端进行响应（异步线程拥有 ServletRequest 和 ServletResponse 对象的引用）。开启异步请求处理之后，Servlet 线程不再是一直处于阻塞状态以等待业务逻辑的处理，而是启动异步线程之后可以立即返回。异步处理的特性可以帮助应用节省容器中的线程。如图所示：

我们还能发现，`实际上这里的异步请求处理对于客户端浏览器来说仍然是同步输出，它并没有提升响应速度，用户是没有感知的`**，但是异步请求处理解放了服务器端的请求处理线程的使用，处理请求线程并没有卡在业务代码那里等待，当前的业务逻辑被转移给其他线程去处理了，能够让tomcat同时接受更多的请求，从而提升了并发处理请求的能力！**

代码说话

```java
package com.zt.javastudy.async;

import brave.Tracing;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.beans.factory.config.ConfigurableBeanFactory;
import org.springframework.cloud.sleuth.SpanNamer;
```

```java
import org.springframework.cloud.sleuth.instrument.async.TraceRunnable;
import org.springframework.context.annotation.Scope;
import org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor;
import org.springframework.util.ObjectUtils;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.servlet.AsyncContext;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

/**
 * 测试异步http请求
 *
 * @author zhengtao on 2021/9/23
 */
@RestController
@Slf4j
@Scope(value = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public class AsyncController {
    @Autowired
    @Qualifier("httpWorkThreadPool")
    private ThreadPoolTaskExecutor executor;
    @Autowired
    private Tracing tracing;
    @Autowired
    private SpanNamer defaultSpanNamer;
//    private static LongAdder start = new LongAdder();
    private volatile long start;

    @RequestMapping("/testAsync")
    public void test(HttpServletRequest request, HttpServletResponse response) {
        log.info("请求开始! ");
        start = System.currentTimeMillis();
        AsyncContext asyncContext = request.startAsync(request, response);
        // 设置监听
        asyncContext.addListener(new HttpAsyncListener());
        executor.execute(new TraceRunnable(tracing, defaultSpanNamer, () -> {
            try {
                doInvoke(asyncContext);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }));
    }
    /**
     * 处理业务
     *
     * @param asyncContext
     */
    private void doInvoke(AsyncContext asyncContext) throws InterruptedException
{
        Thread.sleep(5000);
        completeResponse("这是一个异步的http请求", 200, asyncContext);
    }
```

```
    /**
     * 将
     * @param context
     * @param status
     * @param asyncContext
     */
    private void completeResponse(String context, int status, AsyncContext
asyncContext) {
        HttpServletResponse servletResponse = (HttpServletResponse)
asyncContext.getResponse();
        if (!ObjectUtils.isEmpty(context)) {

 servletResponse.setContentType(asyncContext.getRequest().getContentType());
            servletResponse.setStatus(status);
            completeResponse(servletResponse, context);
        }
        // 调用了complete方法后才算请求完成
        asyncContext.complete();
        log.info("请求处理时间:{}ms", (System.currentTimeMillis() - start));
    }

    private void completeResponse(HttpServletResponse servletResponse, String
context) {
        ServletOutputStream out = null;
        try {
            byte[] buff = context.getBytes();
            servletResponse.setContentLength(buff.length);
            out = servletResponse.getOutputStream();
            out.write(buff);
            out.flush();
        } catch (IOException e) {
            log.error("complete http reqeust error", e);
        } finally {
            if (out != null) {
                try {
                    out.close();
                } catch (Exception e) {
                    log.error(e.getMessage(), e);
                }
            }
        }
    }
}
```

```
package com.zt.javastudy.async;

import lombok.extern.slf4j.Slf4j;

import javax.servlet.AsyncEvent;
import javax.servlet.AsyncListener;
import java.io.IOException;

/**
 * 异步监听器
 *
 * @author zhengtao on 2021/9/23
```

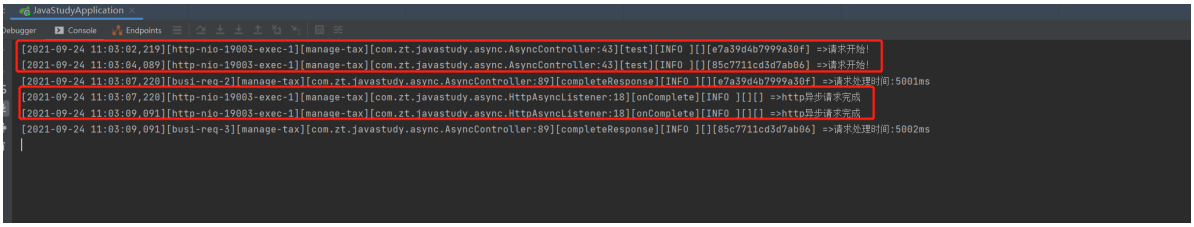
```
 */
@Slf4j
public class HttpAsyncListener implements AsyncListener {
    @Override
    public void onComplete(AsyncEvent event) throws IOException {
        log.info("http异步请求完成");
    }

    @Override
    public void onTimeout(AsyncEvent event) throws IOException {
        log.info("http请求超时");
    }

    @Override
    public void onError(AsyncEvent event) throws IOException {
        log.info("http请求失败");
    }

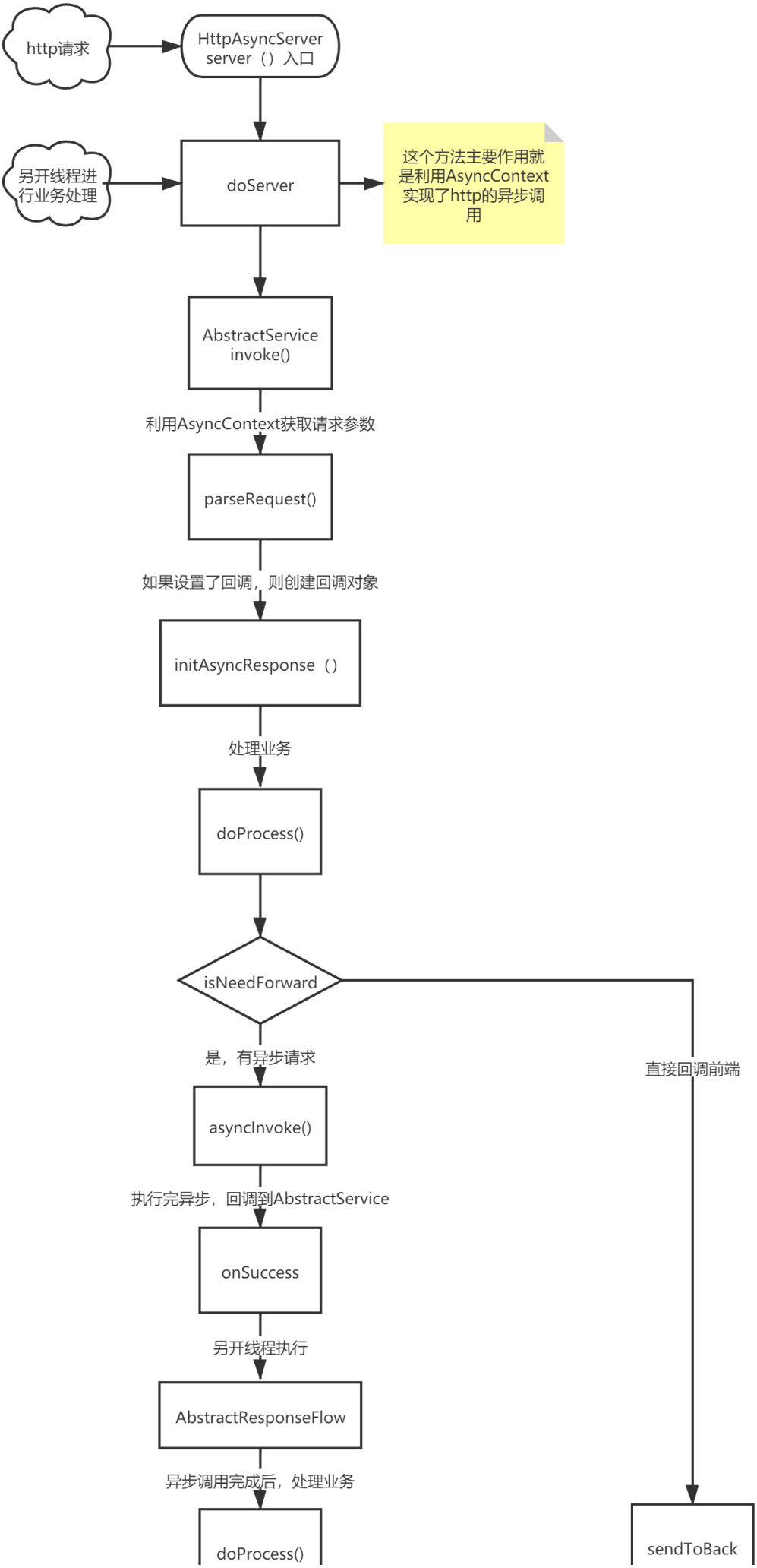
    @Override
    public void onStartAsync(AsyncEvent event) throws IOException {
        log.info("http异步请求开始");
    }
}
```

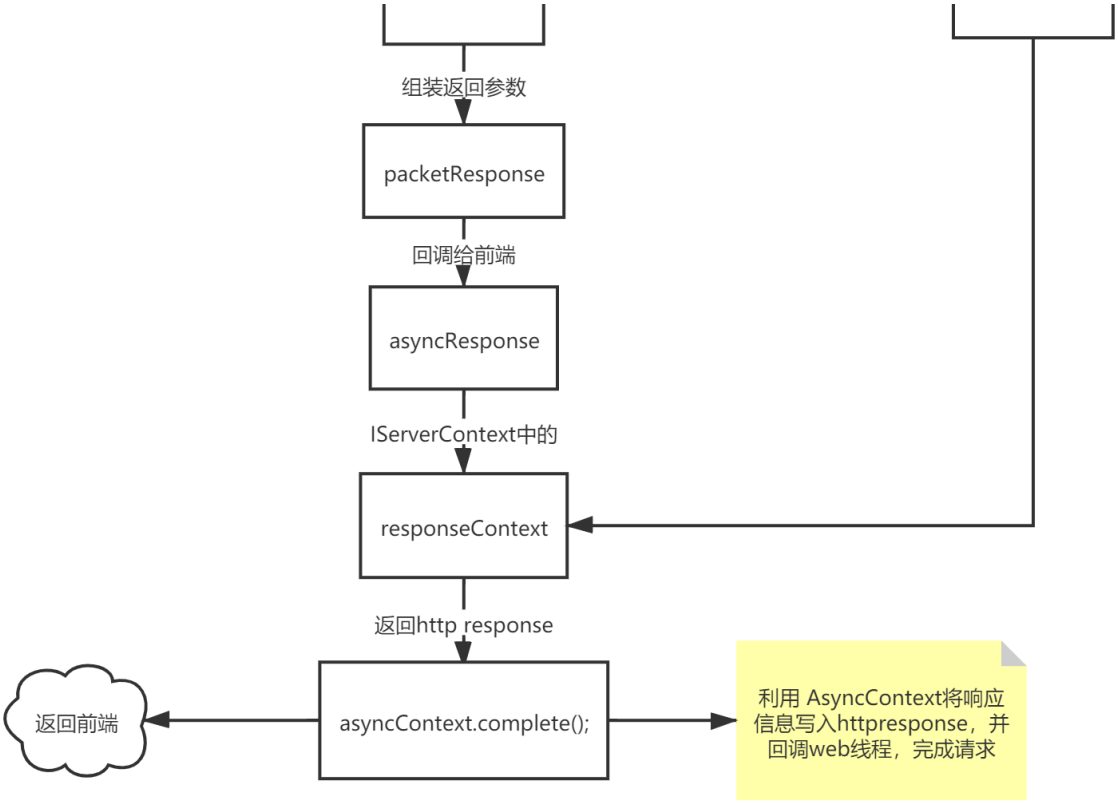
测试，同样是发两个请求：



可以很明显的看到，请求线程是同一个，但是第一个请求还没结束第二个请求就已经开始处理了。业务代码则是自定义的线程池处理的。

搞懂这个之后看公司的代码就很简单了


http请求
HttpAsyncServer
server（）入口
另开线程进行业务处理
doServer
这个方法主要作用就是利用AsyncContext实现了http的异步调用
AbstractService
invoke()
利用AsyncContext获取请求参数
parseRequest()
如果设置了回调，则创建回调对象
initAsyncResponse（）
处理业务
doProcess()
isNeedForward
是，有异步请求
直接回调前端
asyncInvoke()
执行完异步，回调到AbstractService
onSuccess
另开线程执行
AbstractResponseFlow
异步调用完成后，处理业务
doProcess()
sendToBack



静态方法中使用bean

写一些工具类时，可能会用到其他的bean。第一眼是这样写的

```
@Component
public class MerchUtils {
    @Autowired
    private static IAgentService iAgentService;

    private static ConcurrentHashMap<String, Agent> agentMap = new
ConcurrentHashMap<>();

    public static String getAgentName(String agentId) {
        Agent agent = agentMap.get(agentId);
        if (agent == null && !agentMap.containsKey(agentId)) {
            agent = iAgentService.get(agentId);
            if (agent != null && StringUtils.isNotEmpty(agentId)) {
                agentMap.put(agentId, agent);
            }
        }
        if (agent == null) {
            return agentId;
        }
        return agent.getAgentNameCn();
    }
}
```

这样会报null，因为静态方法优先于bean的注入。

正确写法为：

```
public class MerchUtils {
    private static IAgentService iAgentService;
```

```
    @Autowired
    public MerchUtils(IAgentService iAgentService) {
        MerchUtils.iAgentService = iAgentService;
    }

    private static ConcurrentHashMap<String, Agent> agentMap = new
ConcurrentHashMap<>();

    public static String getAgentName(String agentId) {
        Agent agent = agentMap.get(agentId);
        if (agent == null && !agentMap.containsKey(agentId)) {
            agent = iAgentService.get(agentId);
            if (agent != null && StringUtils.isNotEmpty(agentId)) {
                agentMap.put(agentId, agent);
            }
        }
        if (agent == null) {
            return agentId;
        }
        return agent.getAgentNameCn();
    }
}
```

@ConditionalOnProperty

有时需要根据配置文件来决定是否创建 bean

```
package com.zt.javastudy.grammar;

import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.autoconfigure.condition.ConditionalOnProperty;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * 注解学习
 *
 * @author zhengtao on 2021/10/27
 */
@Configuration
@ConditionalOnProperty(
        prefix = "test",
        name = {"enable"},
        havingValue = "true",
        matchIfMissing = false
)
@Slf4j
public class ConditionalTest {
    @Bean(initMethod = "start", destroyMethod = "shutdown")
    public ConditionalTest buildProducer() {
        log.info("创建bean了");
        ConditionalTest conditionalTest = new ConditionalTest();
        return conditionalTest;
    }
```

```
    private void start() {
    }

    private void shutdown() {
    }
}
```

其中 name 就是配置项的名称，havingValue 就是匹配的值，在这个代码中只有test.enable=true,才会创建bean。

分布式id解决方案

uuid

UUID(Universally Unique Identifier)的标准型式包含32个16进制数字，以连字号分为五段，形式为8-4-4-4-12的36个字符，示例：`550e8400-e29b-41d4-a716-446655440000`

优点：

- 性能非常高：本地生成，没有网络消耗。

缺点：

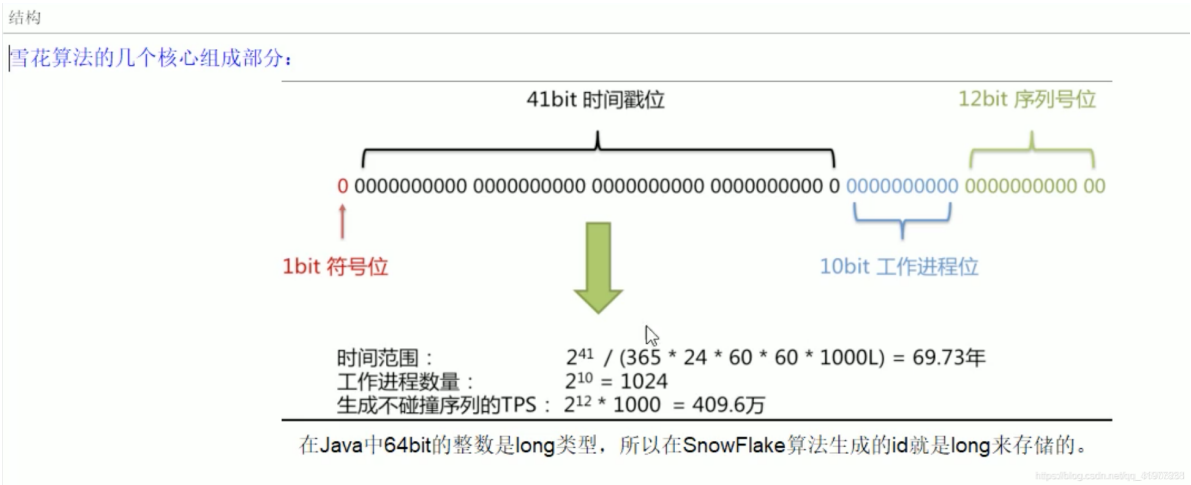
- 不易于存储：UUID太长，16字节128位，通常以36长度的字符串表示，很多场景不适用。
- 信息不安全：基于MAC地址生成UUID的算法可能会造成MAC地址泄露，这个漏洞曾被用于寻找梅丽莎病毒的制作者位置。
- ID作为主键时在特定的环境会存在一些问题，比如做DB主键的场景下，UUID就非常不适用：

  ① MySQL官方有明确的建议主键要尽量越短越好，36个字符长度的UUID不符合要求。

  ② 对MySQL索引不利：如果作为数据库主键，在InnoDB引擎下，UUID的无序性可能会引起数据位置频繁变动，严重影响性能。

snowflake(雪花算法)

雪花算法（Snowflake）是twitter公司内部分布式项目采用的ID生成算法，具体规则如下：



- 1位，不用。二进制中最高位为1的都是负数，但是我们生成的id一般都使用整数，所以这个最高位固定是0
- 41位，用来记录时间戳（毫秒）。 - 41位可以表示 2^{41}-1 个数字，
    - 如果只用来表示正整数（计算机中正数包含0），可以表示的数值范围是：0 至 2^{41}-1，减1是因为可表示的数值范围是从0开始算的，而不是1。 - 也就是说41位可以表示 2^{41}-1 个毫秒的值，转化成单位年则是 (2^{41}-1) / (1000 *60* 60 *24* 365) = 69 年
- 10位，用来记录工作机器id。 - 可以部署在 2^{10} = 1024 个节点，包括 5位 datacenterId 和 5位 workerId - 5位（bit）可以表示的最大正整数是 2^{5}-1 = 31 ，即可以用 0、1、2、3、....31 这 32 个数字，来表示不同的 datecenterId 或 workerId
- 12位，序列号，用来记录同毫秒内产生的不同id。 - 12位（bit）可以表示的最大正整数是 2^{12}-1 = 4095 ，即可以用 0、1、2、3、....4094 这 4095 个数字，来表示同一机器同一时间截（毫秒)内产生的 4095 个 ID 序号。

由于在 Java 中 64bit 的整数是 long 类型，所以在 Java 中 SnowFlake 算法生成的 id 就是 long 来存储的。

优点：

- 毫秒数在高位，自增序列在低位，整个ID都是趋势递增的。
- 不依赖数据库等第三方系统，以服务的方式部署，稳定性更高，生成ID的性能也是非常高的。
- 可以根据自身业务特性分配bit位，非常灵活。

缺点：

- 强依赖机器时钟，如果机器上时钟回拨，会导致发号重复或者服务会处于不可用状态。

基于Redis模式

`Redis`也同样可以实现，原理就是利用`redis`的`incr`命令实现ID的原子性自增。

```
127.0.0.1:6379> set seq_id 1     // 初始化自增ID为1
OK
127.0.0.1:6379> incr seq_id      // 增加1，并返回递增后的数值
(integer) 2
```

用`redis`实现需要注意一点，要考虑到redis持久化的问题。`redis`有两种持久化方式`RDB`和`AOF`

- `RDB`会定时打一个快照进行持久化，假如连续自增但`redis`没及时持久化，而这会Redis挂掉了，重启Redis后会出现ID重复的情况。
- `AOF`会对每条写命令进行持久化，即使`Redis`挂掉了也不会出现ID重复的情况，但由于incr命令的特殊性，会导致`Redis`重启恢复的数据时间过长。

简单代码演示：

```
Long serialNo = RedisTemplateUtil.incrBy(REDIS_KEY_PREFIX +
seq.getSequenceName(), seq.getStep());
public static Long incrBy(String key, long delta){
    return getRedisTemplate().opsForValue().increment(key, delta);
}
```

mybatis - 动态数据源

动态数据源，主要是为了解决读写分离的场景。

那么创建一个数据源主要有哪几步呢?

- 配置 dao，model(bean)，xml mapper文件的扫描路径
- 注入数据源配置属性，创建数据源。
- 将数据源设置到SQL会话工厂和事务管理器。

这样当进行数据库操作时，**就会通过我们创建的动态数据源去获取要操作的数据源了。**一步一步操作完成功能

创建多个数据源

```
package com.jlpay.saas.common.db.datasource;

import com.alibaba.druid.spring.boot.autoconfigure.DruidDataSourceBuilder;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.Map;

/**
 * @Description: 配置多数据源
 * @Author shuqingzhou
 * @Date 2021/11/15 9:09
 * @Version 1.0
 */
@Configuration
@Slf4j
public class DynamicDataSourceConfig {
    /**
     * 创建 DataSource Bean
     */
    @Primary
    @Bean("masterDataSource")
    @ConfigurationProperties(prefix = "spring.datasource.druid.master")
    public DataSource masterDataSource(){
        DataSource dataSource = DruidDataSourceBuilder.create().build();
        return dataSource;
    }

    /**
     * 创建 DataSource Bean
     */
    @Bean("slaveDataSource")
    @ConfigurationProperties(prefix = "spring.datasource.druid.salve")
    public DataSource slaveDataSource(){
        DataSource dataSource = DruidDataSourceBuilder.create().build();
```

```
        return dataSource;
    }

    /**
     * 如果还有数据源,在这继续添加 DataSource Bean
     */
    @Bean("dynamicDataSource")
    @Primary
    public DynamicDataSource dynamicDataSource(@Qualifier("masterDataSource")
DataSource masterDataSource, @Qualifier("slaveDataSource")DataSource
slaveDataSource) {
        Map<Object, Object> targetDataSources = new HashMap<>(2);
        targetDataSources.put(DataSourceNames.MASTER, masterDataSource);
        targetDataSources.put(DataSourceNames.SLAVE, slaveDataSource);
        // 还有数据源,在targetDataSources中继续添加
        log.info("DataSources:{}", targetDataSources);
        return new DynamicDataSource(masterDataSource, targetDataSources);
    }
}
```

将数据源设置到SQL会话工厂和事务管理器

```
package com.jlpay.saas.common.db.datasource;

import org.apache.ibatis.session.SqlSessionFactory;
import org.mybatis.spring.SqlSessionFactoryBean;
import org.mybatis.spring.SqlSessionTemplate;
import org.mybatis.spring.boot.autoconfigure.MybatisProperties;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import
org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.*;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.PlatformTransactionManager;

import java.io.File;
import java.io.FileInputStream;
import java.io.InputStream;

/**
 * @Description: Mybatis数据源配置
 * @Author shuqingzhou
 * @Date 2021/11/15 9:09
 * @Version 1.0
 */
@Configuration
@EnableConfigurationProperties(MybatisProperties.class)
public class MybatisConfig {

    private MybatisProperties mybatisProperties;

    public MybatisConfig(MybatisProperties properties) {
```

```
        this.mybatisProperties = properties;
    }

    @Autowired
    @Qualifier(value = "dynamicDataSource")
    private DynamicDataSource dynamicDataSource;

    /**
     * 配置mybatis的sqlSession连接动态数据源
     * @throws Exception
     */
    @Bean
    public SqlSessionFactory sqlSessionFactory() throws Exception {
        SqlSessionFactoryBean bean = new SqlSessionFactoryBean();
        bean.setDataSource(dynamicDataSource);
        bean.setMapperLocations(mybatisProperties.resolveMapperLocations());
        bean.setTypeAliasesPackage(mybatisProperties.getTypeAliasesPackage());
        Resource resource = new
DefaultResourceLoader().getResource(mybatisProperties.getConfigLocation());
        bean.setConfigLocation(resource);
        bean.setConfiguration(mybatisProperties.getConfiguration());
        return bean.getObject();
    }

    @Bean
    public SqlSessionTemplate sqlSessionTemplate() throws Exception {
        return new SqlSessionTemplate(sqlSessionFactory());
    }

    /**
     * 将动态数据源添加到事务管理器中，并生成新的bean
     * @return the platform transaction manager
     */
    @Bean
    public PlatformTransactionManager transactionManager() {
        return new DataSourceTransactionManager(dynamicDataSource);
    }
}
```

tips：**在访问数据库时会调用 `determineCurrentLookupKey()` 方法获取数据库实例的 key**

```
protected DataSource determineTargetDataSource() {
    Assert.notNull(this.resolvedDataSources, "DataSource router not
initialized");
    Object lookupKey = determineCurrentLookupKey();
    DataSource dataSource = this.resolvedDataSources.get(lookupKey);
    if (dataSource == null && (this.lenientFallback || lookupKey == null)) {
        dataSource = this.resolvedDefaultDataSource;
    }
    if (dataSource == null) {
        throw new IllegalStateException("Cannot determine target DataSource for
lookup key [" + lookupKey + "]");
    }
    return dataSource;
```

```
}

@Nullable
protected abstract Object determineCurrentLookupKey();
```

所以我们可以通过重写 `determineCurrentLookupKey` 方法来实现更改数据源

```
package com.jlpay.saas.common.db.datasource;

import org.springframework.jdbc.datasource.lookup.AbstractRoutingDataSource;

import javax.sql.DataSource;
import java.util.Map;

/**
 * @Description: 动态多数据源
 * @Author shuqingzhou
 * @Date 2021/11/15 9:09
 * @Version 1.0
 */
public class DynamicDataSource extends AbstractRoutingDataSource {
    private static final ThreadLocal<String> CONTEXT_HOLDER = new ThreadLocal<>
();

    /**
     * 配置DataSource, defaultTargetDataSource为主数据库
     */
    public DynamicDataSource(DataSource defaultTargetDataSource, Map<Object,
Object> targetDataSources) {
        super.setDefaultTargetDataSource(defaultTargetDataSource);
        super.setTargetDataSources(targetDataSources);
        super.afterPropertiesSet();
    }

    @Override
    protected Object determineCurrentLookupKey() {
        return getDataSource();
    }

    public static void setDataSource(String dataSource) {
        CONTEXT_HOLDER.set(dataSource);
    }

    public static String getDataSource() {
        return CONTEXT_HOLDER.get();
    }

    public static void remove() {
        CONTEXT_HOLDER.remove();
    }
}
```

这里使用一个ThreadLocal的变量达到每个线程都有自己的数据源的效果。我们只需要在要需要切换数据源时

DynamicDataSource.setDataSource(),即可动态的更改数据源。

但是这样做未免太low了，所以运用切面来升级一波！！！

1. 定义注解

```
package com.jlpay.saas.common.db.datasource;

import java.lang.annotation.*;

/**
 * @Description: 数据源注解
 * @Author shuqingzhou
 * @Date 2021/11/15 9:09
 * @Version 1.0
 */
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface DataSource {
    String value() default DataSourceNames.MASTER;
}
package com.jlpay.saas.common.db.datasource;

/**
 * @Description: 主备库名
 * @Author shuqingzhou
 * @Date 2021/11/15 9:09
 * @Version 1.0
 */
public interface DataSourceNames {
    String MASTER = "master";
    String SLAVE = "slave";
}
```

2. **创建一个AOP切面，拦截带 @DataSource 注解的方法，在方法执行前切换至目标数据源，执行完成后恢复到默认数据源。**

```
package com.jlpay.saas.common.db.datasource;

import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.reflect.MethodSignature;
import org.springframework.stereotype.Component;

import java.lang.reflect.Method;

/**
 * @Description: 数据源切面处理
 * @Author shuqingzhou
 * @Date 2021/11/15 9:09
 * @Version 1.0
 */
@Aspect
@Component
```

```
@Slf4j
public class DataSourceAspect {

    @Before("execution(* com.jlpay.saas.*.db.service..*(..))")
    public void before(JoinPoint point) throws Throwable {
        MethodSignature signature = (MethodSignature) point.getSignature();
        Method method = signature.getMethod();
        DataSource dataSource = method.getAnnotation(DataSource.class);
        if (dataSource == null) {
            log.debug("【默认数据源】，切入点：{}", signature.toShortString());
            DynamicDataSource.setDataSource(DataSourceNames.MASTER);
        } else {
            log.debug("【切换数据源】：{}，切入点：{}", dataSource.value(),
signature.toShortString());
            DynamicDataSource.setDataSource(dataSource.value());
        }
    }

    @After("execution(* com.jlpay.saas.*.db.service..*(..))")
    public void after() throws Throwable{
        DynamicDataSource.remove();
    }

    @AfterThrowing("execution(* com.jlpay.saas.*.db.service..*(..))")
    public void afterThrowing() {
        log.info("数据源异常，切换主库数据源");
        DynamicDataSource.remove();
        DynamicDataSource.setDataSource(DataSourceNames.MASTER);
    }
}
```

到现在如何实现动态数据源已经很透彻了。主要流程有：

- 创建数据源
- 将数据源设置到SQL会话工厂和事务管理器
- 重写 `determineCurrentLookupKey` 方法来更改数据源（到此，其实已经完成了动态数据源的功能，调用set方法即可动态的更改数据源）
- 自定义注解，并创建一个切面来调用set方法动态的更改数据源

公司的jwt公共包

Cookie、Session、Token、JWT（JSON Web Token） 这些都是登录认证中经常用到的东西，

具体链接可见https://juejin.cn/post/6844904034181070861

JWT消息构成

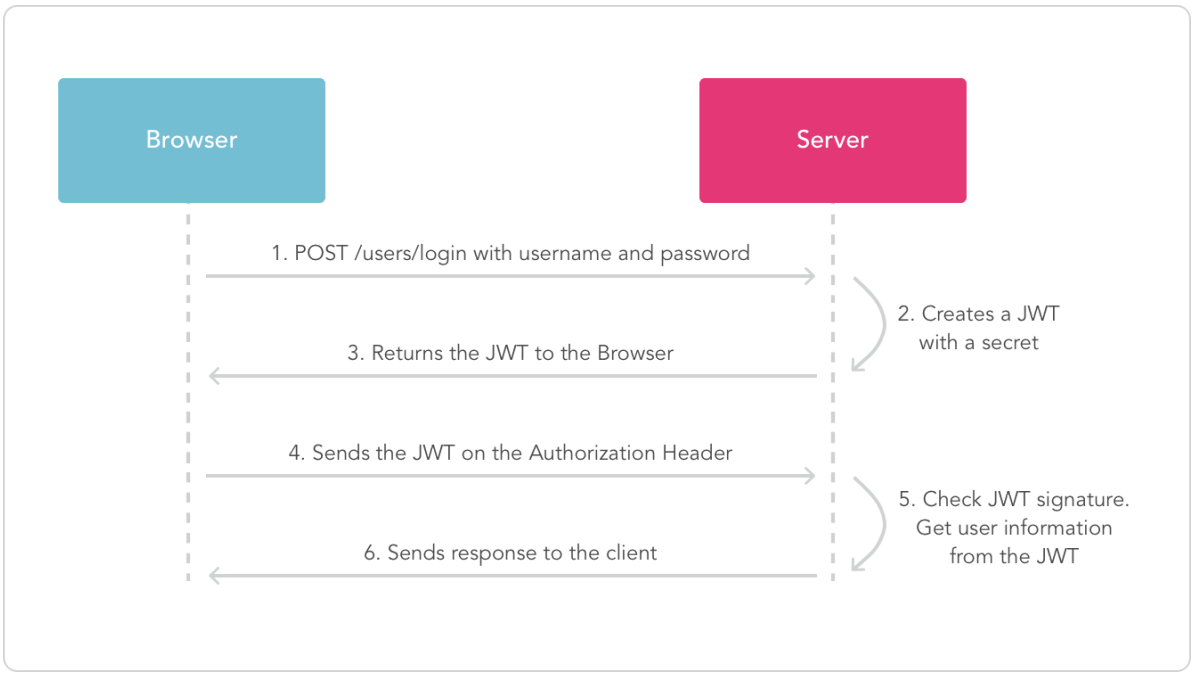
一个token分3部分，按顺序:

```
头部（header）: 一般为固定的，加密方式等
载荷（payload）: 是一个 JSON 对象，用来存放实际需要传递的数据
签证（signature）: 对前两部分进行签名，防止数据篡改
```

```
其中载荷（payload)中JWT 规定了7个官方字段
iss (issuer)：签发人
 exp (expiration time)：过期时间，这个很重要
 sub (subject)：主题
 aud (audience)：受众
 nbf (Not Before)：生效时间
 iat (Issued At)：签发时间
 jti (JWT ID)：编号
```

- 对象为一个很长的字符串，字符之间通过"."分隔符分为三个子串。注意JWT对象为一个长字串，各字串之间也没有换行符，一般格式为：xxxxx.yyyyy.zzzzz 。 例如：
- ```
  yJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpv
  aG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw
  5c
  ```

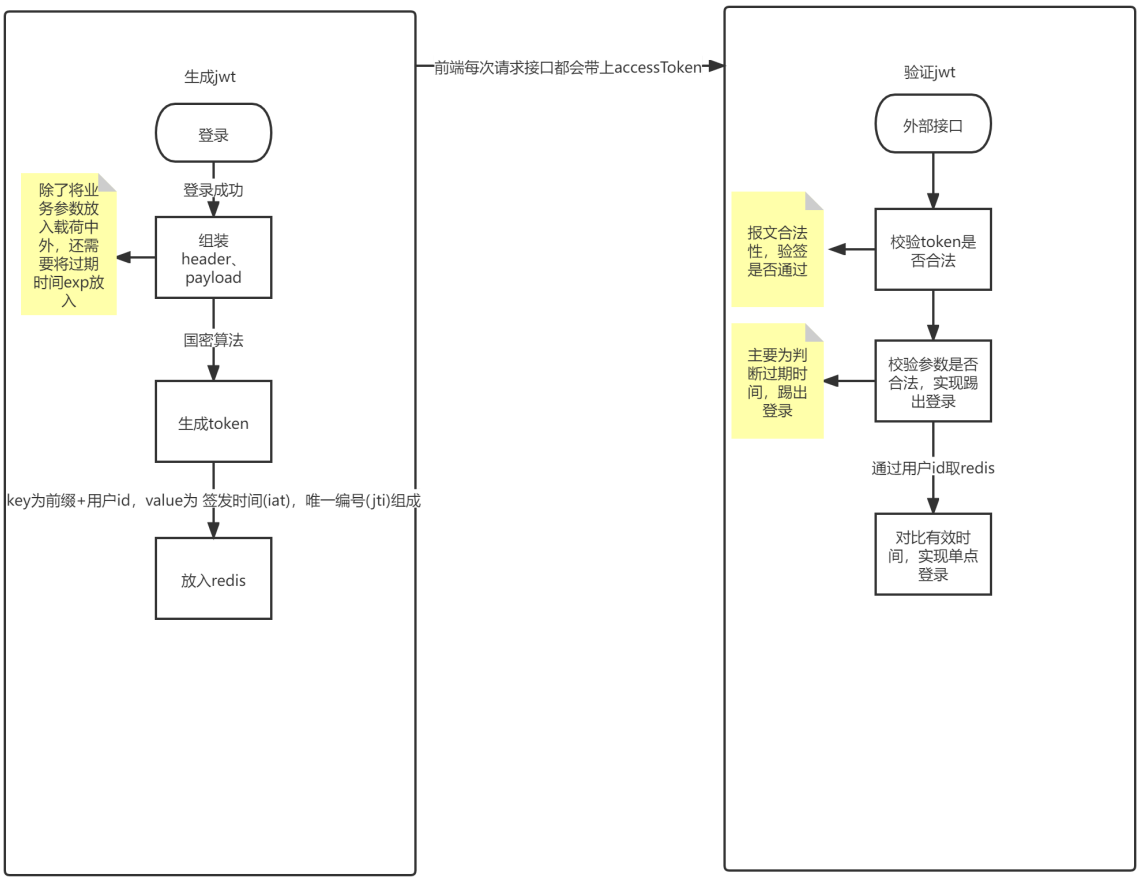
JWT的认证流程图


  ```

流程说明：
1，浏览器发起请求登陆，携带用户名和密码；
2，服务端验证身份，根据算法，将用户标识符打包生成 token，
3，服务器返回JWT信息给浏览器，JWT不包含敏感信息；
4，浏览器发起请求获取用户资料，把刚刚拿到的 token一起发送给服务器；
5，服务器发现数据中有 token，验明正身；
6，服务器返回该用户的用户资料；

公司代码应用

熟悉之前两部分后，只要抱着是怎样加密json、实现过期时间、单点登录去看即可代码即可，整体流程如下



登录

```
@Service(LoginReq.COMMAND_SERVICE_NAME)
public class LoginCmd extends BaseCmdService<LoginReq> {
    private static final String JWT_HEADER = "{ \"alg\": \"SM2SM3\", \"typ\":
\"JWT\", \"ver\": \"1\"}";
    private static final String PLAT_CODE = "MERCH";

    @Autowired
    private StaffService merchUserService;
    @Autowired
    private JwtSessionUtils jwtSessionUtils;
    @Autowired
    private IMonitorService monitorService;
```

```
    @Value("${jlpay.framework.jwt.expire.plat:86400}")
    private int jwtWxExpireSeconds;

    /**
     * 执行命令将请求参数传给实现类
     *
     * @param req 请求参数
     * @return
     * @throws Exception
     */
    @Override
    protected BaseCmdResp invoke(LoginReq req) throws Exception {
        // 登录逻辑

        //生成JWT ACCESS TOKEN
        JwtHeaderInfo jwtHeaderInfo = JSON.parseObject(JWT_HEADER,
JwtHeaderInfo.class);
        jwtHeaderInfo.setPlat(plat);
        JwtPayloadInfo payloadInfo = buildJwtPayloadInfo(req, resp);
        String jwtToken = jwtSessionUtils.getAccessToken(jwtHeaderInfo,
payloadInfo, jwtWxExpireSeconds, TimeUnit.SECONDS, monitorService);
        resp.setAccessToken(jwtToken);

        // 逻辑处理

        //返回
        return resp;
    }

    /**
     * 构造需要传递的数据
     * @param req
     * @param resp
     * @return
     */
    private JwtPayloadInfo buildJwtPayloadInfo(LoginReq req, LoginResp resp) {
        JwtPayloadInfo payloadInfo = new JwtPayloadInfo();
        payloadInfo.setUid(resp.getUserId());
        payloadInfo.setUname(resp.getUserName());
        payloadInfo.setUtype(resp.getUserType());
        payloadInfo.setCid(resp.getMerchId());
        payloadInfo.setMno(resp.getShopId());
        payloadInfo.setMname(resp.getShopName());
        payloadInfo.setAud(resp.getMerchType());
        payloadInfo.setNbf(DateEasyUtil.format(new Date(),
DateEasyUtil.NEW_FORMAT));
        return payloadInfo;
    }
}
```

其中主要代码就在getAccessToken 这个方法内

```
/**
```

```
     * 按默认封装的header,payload生成JWT Token
     *
     * @param headerInfo
     * @param payloadInfo
     * @param expireTime
     * @param timeUnit
     * @return
     */
    public String getAccessToken(JwtHeaderInfo headerInfo, JwtPayloadInfo
payloadInfo, int expireTime, TimeUnit timeUnit, IMonitorService monitorService)
{
        getTokenPreProcess(headerInfo, payloadInfo);

        JSONObject headerJson = JSON.parseObject(JSON.toJSONString(headerInfo));
        JSONObject payloadJson =
JSON.parseObject(JSON.toJSONString(payloadInfo));
        // 生成token，并添加exp失效时间参数
        String accessToken = jwtHelper.generateToken(headerJson, payloadJson,
expireTime, timeUnit);
        // 放redis
        setJwtCacheInfo(headerInfo, payloadInfo, monitorService);

        return accessToken;
    }
```

生成token，就是使用国密算法加密得到token，放入redis中

```
 /**
     * 获取token前做的前置操作：设置签发时间，设置jti编号
     *
     * @param headerInfo
     * @param payloadInfo
     */
    private void getTokenPreProcess(JwtHeaderInfo headerInfo, JwtPayloadInfo
payloadInfo) {
        String deviceId = payloadInfo.getDeviceId();
        // 统一签发时间为当前执行时间
        String currentDate = DateUtil.getLongDateString(new Date());
        payloadInfo.setIat(currentDate);
        // 有设备号jti存设备号，否则取随机数
        if (StringUtils.isNotEmpty(deviceId)) {
            payloadInfo.setJti(deviceId);
        } else {
            payloadInfo.setJti(generateJti(JTI_BIT));
        }
        // 检查入参
        checkRequest(headerInfo, payloadInfo);
    }
     /**
     * 设置JwtCacheInfo
     *
     * @param headerInfo
     * @param payloadInfo
     * @param monitorService
```

```
     */
    private void setJwtCacheInfo(JwtHeaderInfo headerInfo, JwtPayloadInfo
payloadInfo, IMonitorService monitorService) {
        String loginJwtKey = getLoginJwtKey(headerInfo.getPlat(),
payloadInfo.getUid());
        JwtCacheInfo jwtCacheInfo = new JwtCacheInfo();
        jwtCacheInfo.setIat(payloadInfo.getIat());
        jwtCacheInfo.setJti(payloadInfo.getJti());
        // 存这个redis并没有加入失效时间是否可取?
        boolean setJwtCacheResult = persistService.setString(loginJwtKey,
JSON.toJSONString(jwtCacheInfo));
        if (monitorService != null && !setJwtCacheResult) {
            monitorService.monitor("写入JWT失败, 请排查Xpipe-redis-write节点是否故
障");
        }
    }
```

保证了，key为前缀+用户id，value为 签发时间(iat)，唯一编号(jti)组成

校验jwt

主要为verifyJwtAccessToken这个方法

```
public JwtInfo verifyJwtAccessToken(String accessToken, IMonitorService
monitorService, IJwtCacheCheckService cacheCheckService) throws JwtException {
        JwtInfo jwtInfo = new JwtInfo();
        try {
            // 校验jwt
            DecodedJWT jwtDecoded = jwtHelper.verify(accessToken);
            JwtHeaderInfo jwtHeaderInfo =
JSON.parseObject(jwtHelper.getHeader(jwtDecoded), JwtHeaderInfo.class);
            JwtPayloadInfo payloadInfo =
JSON.parseObject(jwtHelper.getPayloadClaims(jwtDecoded), JwtPayloadInfo.class);
            jwtInfo.setHeaderInfo(jwtHeaderInfo);
            jwtInfo.setPayloadInfo(payloadInfo);

            String loginJwtKey = getLoginJwtKey(jwtHeaderInfo.getPlat(),
payloadInfo.getUid());
            String jwtCacheStr = persistService.getString(loginJwtKey);
            JwtCacheInfo jwtCacheInfo = JSON.parseObject(jwtCacheStr,
JwtCacheInfo.class);
            // 获取到jwtCacheInfo，则校验，获取不到则不校验并告警(降级处理)
            if (jwtCacheInfo == null) {
                monitorService.monitor("读取JWT失败, 请排查Xpipe-redis-read节点是否故
障");
                return jwtInfo;
            }

            //jwtCacheInfo及JWT是否过期
            cacheCheckService.check(jwtCacheInfo, payloadInfo, iatDiff);
        } catch (JwtException e) {
            log.warn("JWT缓存时间校验不通过", e.getMessage());
            throw new JwtException(JwtExceptionCode.SESSION_KICK_OUT);
        } catch (JWTVerificationException e) {
            log.warn("JWT校验错误", e);
            throw new JwtException(JwtExceptionCode.SESSION_TIME_OUT);
```

```
        } catch (Exception e) {
            log.error("未知错误", e);
            throw new JwtException(JwtExceptionCode.SESSION_TIME_OUT);
        }
        return jwtInfo;
    }
```

这里主要实现，如果token过期，则踢出登录

```
/**
校验jwt是否合法
*/
 public DecodedJWT verify(DecodedJWT jwt) throws JWTVerificationException {
        // 验签校验
        verifyAlgorithm(jwt, algorithm);
        algorithm.verify(jwt);
        // 校验参数，主要校验过期时间，实现踢出登录
        verifyClaims(jwt, claims);
        return jwt;
    }
```

这里主要实现单点登录

```
/**
     * 默认校验规则
     *
     * @param jwtCacheInfo
     * @param payloadInfo
     * @param diff          签发时间iat允许的误差，单位秒
     * @throws JwtException
     */
@Override
public void check(JwtCacheInfo jwtCacheInfo, JwtPayloadInfo payloadInfo, int
diff) throws JwtException {

    Date cacheIat = DateUtil.parseDateLongFormat(jwtCacheInfo.getIat());
    if (cacheIat == null) {
        throw new JwtException("iat not in xpipe-redis");
    }
    // 目前的时间
    Date payloadIat = DateUtil.parseDateLongFormat(payloadInfo.getIat());
    if (payloadIat == null) {
        throw new JwtException("iat not in jwt payload");
    }

    // 通过uid和plat取用户最后一次登录生成JWT TOKEN的生效时间(多机房通过XPIPE同步)，
    // 如果当前TOKEN生效时间早于取到的时间则判断为失效，如果没取到最后一次生效时间，则认为是
TOKEN是有效的
    if (payloadIat.before(cacheIat)) {
        throw new JwtException("iat(payload) is before iat(xpipe-redis)");
    }
```

```
    // 如果当前TOKEN的签发时间早于或者晚于XPIPE中取到的签发时间3秒(可配置)以内，则jti编号必须一致才认为TOKEN是有效的
    if (cacheIat.before(DateUtil.addSeconds(payloadIat, diff)) &&
cacheIat.after(DateUtil.addSeconds(payloadIat, -diff)) &&
!payloadInfo.getJti().equals(jwtCacheInfo.getJti())) {
        throw new JwtException("jti(payload) and jti(xpipe-redis) must be
same");
    }

}
```