1. 项目规模，一天/月数据量，各组件版本?

数据规模：一般100M数据由300万条数据；数据量：上百G；条数：达到几十亿条数据。

美团数据规模：负责每天数百GB的数据存储和分析。

2. Spark 2.x 和Spark 1.x版本的区别?

- Spark2.x实现了Spark sql和Hive Sql操作API的统一。
- Spark2.0中引入了 SparkSession 的概念，它为用户提供了一个统一的切入点来使用 Spark 的各项功能， 统一了旧的SQLContext与HiveContext。
- 统一 DataFrames 和 Datasets 的 API
- Spark Streaming基于Spark SQL(DataFrame / Dataset)构建了high-level API，使得Spark Streaming充分受益Spark SQL的易用性和性能提升。

3. 项目中的遇见的问题，如何解决?

讲了数据倾斜。

4. Hive元数据存储了哪些信息?

存储了hive中所有表格的信息，包括表格的名字，表格的字段，字段的类型就是表的定义。

5. 数据去重怎么做? 【UDF使用】

在hive数据清洗这里总结三种常用的去重方式。

1. distinct
2. group by
3. row_number()

实例：

```
SELECT tel, link_name, certificate_no, certificate_type, modify_time

FROM order_info

WHERE deleted = 'F'

AND pay_status = 'payed'

AND create_time >= to_date('2017-04-23', 'yyyy-MM-dd')

AND create_time < to_date('2017-04-24', 'yyyy-MM-dd')

AND row_number() over(PARTITION BY tel ORDER BY tel DESC) = 1
```

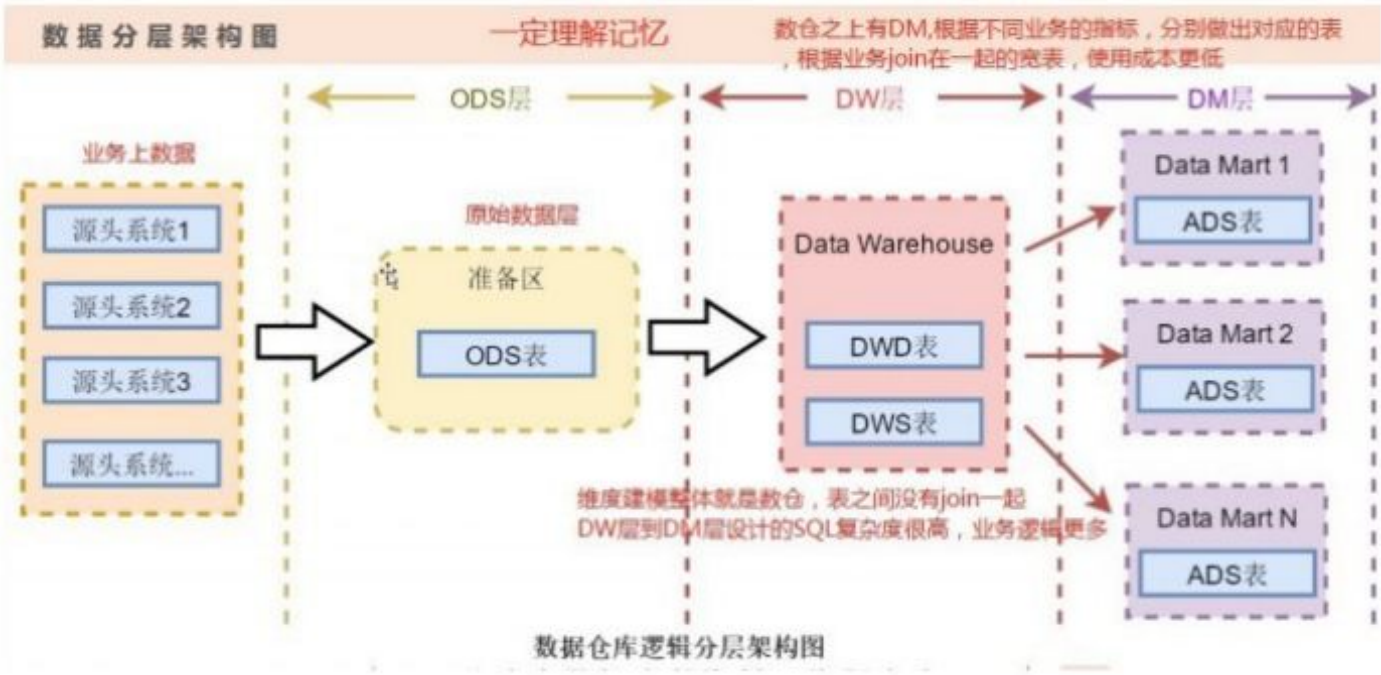
上面SQL对某一字段（tel）排序后分区去重，这样避免了其对不相干字段的数据干扰，影响数据处理的效率。

（推荐方法三）

6. udf, udaf, udtf 有什么区别?

- UDF：用户自定义普通函数，1对1关系，常用于select语句。
- UDAF：用户自定义聚合函数，多对1关系,常用于group by语句。
- UDTF：用户自定义表生成函数,1对多关系 分词 输入一句话输出多个单词。

7. 项目上数仓分层如何做的?



8. Spark Streaming和Spark联系和区别?

spark (RDD) = spark streaming (DStream)

spark (RDD DAG) = spark streaming （DStreamGraph)

Dstream 是Spark Streaming特有的数据类型。

DStream代表一系列连续的RDD，带有时间维度的RDD，在原来RDD的基础上加上时间。比如上图的0到1秒有一个RDD，1-2秒有一个RDD，等等

spark-core：RDD开发，RDD-DAG图。

spark-Streaming：针对Dstream开发，DstreamGraph。

Dstream：代表了一系列连续的RDD，每一个RDD包含特定时间间隔数据。

RDD的DAG是一个空间概念，Dstream在RDD基础上加了一个时间维度。

Dstream各种操作是可以映射到内部RDD上进行的，对DStream的操作可以通过RDD的transformation生成新的Dstream。

算子方面的区别：

RDD算子：transform action。

DStream：transform ouput 保证数据有输入和输出，遇见输出的时候才激活整个DAG图。

Kafka如何保证数据的安全性和可靠性?

可靠性：

每个分区在Kafka集群的若干服务器中都有副本，这样这些持有副本的服务可以共同处理数据和请求，副本数量是可以配置的。

副本使Kafka具备了容每个分区都由一个服务器作为“leader”，零或若干服务器作为“followers”,leader负责处理消息的读和写，followers和Leader同步只负责读，fol。

followers中的一台则会自动成为leader。集群中的每个服务都会同时扮演两个角色：作为它所持有的一部分分区的leader，同时作为其他分区的follow。

安全性：

Kafka 采用的是time-based消息保留策略（SLA），默认保存时间为7天。

持久化数据存储：直接到磁盘，没有内存缓存机制。[磁盘为什么慢：大量随机文件的读写]。

可进行持久化操作。将消息持久化到磁盘，因此可用于批量消费。

持久化数据存储尽可能进行连续的读写，避免随机的读写。

10. Kafka的数据是有序的吗?

Partition的功能

目的：实现负载均衡【partition分布在不同的节点上】，需要保证消息的顺序性。

顺序性的保证：订阅消息是从头后读的，写消息是尾部追加，所以对顺序性做了一个保证在一个partition上能保证消息的顺序性，但是在多个partition不能保证全局的顺序性。

11. Spark优化?

1. 对多次使用的RDD进行持久化处理避免重复计算。

- cahce()
- persist()
- checkpoint()

2. 避免创建重复的RDD。

3. 尽可能复用同一个RDD。

   类似于多个RDD的数据有重叠或者包含的情况，应该尽量复用一个RDD，以尽可能减少RDD的数量，从而减少算子计算次数。

4. 尽量避免使用shuffle类算子。

Broadcast+map：先将数据collectAsMap收集到Driver段,然后使用map的形式做一个分发，到从节点上做一个join,这种形式只有map操作。

5. 使用map-side预聚合的shuffle操作

因为业务需要，一定要使用shuffle操作，无法用map类算子替代，尽量使用map-side预聚合的算子。在每个节点本地对相同的key进行一次聚合操作map-side预聚合之后，每个节点本地就只会有一条相同的key，因为多条相同的key都被聚合起来了。其他节点在拉取所有节点上的相同key时，就会大通常来说，在可能的情况下，建议使用reduceByKey或者aggregateByKey算子来替代掉groupByKey算子。因为reduceByKey和aggregateByKey算子而groupByKey算子是不会进行预聚合的，全量的数据会在集群的各个节点之间分发和传输，性能相对来说比较差。

6. 使用kryo优化序列化性能

12. Spark Streaming计算速度远远小于Kafka缓存的数据，怎么解决?

或者通过反压机制控制。

Spark Streaming程序中当计算过程中出现batch processing time > batch interval的情况时，(其中batch processing time为实际计算一个批次花费时间，batch interval为Streaming应用设置的批处理间隔)。

意味着处理数据的速度小于接收数据的速度，如果这种情况持续过长的时间，会造成数据在内存中堆积，导致Receiver所在Executor内存溢出等问题(如果设置StorageLevel包含disk, 则内存存放不下的数据会溢写至disk, 加大延迟)。

可以通过设置参数spark.streaming.receiver.maxRate来限制Receiver的数据接收速率，此举虽然可以通过限制接收速率，来适配当前的处理能力，防止内存溢出，但也会引入其它问题。

比如：producer数据生产高于maxRate，当前集群处理能力也高于maxRate，这就会造成资源利用率下降等问题。为了更好的协调数据接收速率与资源处理能力，动态控制数据接收速率来适配Spark Streaming Backpressure: 根据JobScheduler反馈作业的执行信息来动态调整Receiver数据接收率。

通过属性"spark.streaming.backpressure.enabled"来控制是否启用backpressure机制，默认值false，即不启用。

1. spark.streaming.concurrentJobs=10：提高Job并发数，读过源码的话会发现，这个参数其实是指定了一个线程池的核心线程数而已，没有指定。
2. spark.streaming.kafka.maxRatePerPartition=2000：设置每秒每个分区最大获取日志数，控制处理数据量，保证数据均匀处理。
3. spark.streaming.kafka.maxRetries=50：获取topic分区leaders及其最新offsets时，调大重试次数。
4. 在应用级别配置重试。

   spark.yarn.maxAppAttempts=5

   spark.yarn.am.attemptFailuresValidityInterval=1h

此处需要【注意】：

spark.yarn.maxAppAttempts值不能超过hadoop集群中yarn.resourcemanager.am.max-attempts的值，原因可参照下面的源码或者官网配置。

13. Spark Streaming对接Kafka的两种方式的区别?

spark streaming是基于微批处理的流式计算引擎，通常是利用spark core或者spark core与spark sql一起来处理数据。在企业实时处理架构中，通常将spark streaming和kafka集成作为整个大数据处理架构的核心环节之一。

针对不同的spark、kafka版本，集成处理数据的方式分为两种：Receiver based Approach和Direct Approach，不同集成版本处理方式的支持，可参考下图：

Note: Kafka 0.8 support is deprecated as of Spark 2.3.0.

	spark-streaming-kafka-0-8	spark-streaming-kafka-0-10
Broker Version	0.8.2.1 or higher	0.10.0 or higher
API Maturity	Deprecated	Stable
Language Support	Scala, Java, Python	Scala, Java
Receiver DStream	Yes	No
Direct DStream	Yes	Yes
SSL / TLS Support	No	Yes
Offset Commit API	No	Yes
Dynamic Topic Subscription	No	Yes

Receiver based Approach

基于receiver的方式是使用kafka消费者高阶API实现的。

对于所有的receiver，它通过kafka接收的数据会被存储于spark的executors上，底层是写入BlockManager中，默认200ms生成一个block（通过配置参数spark.streaming.blockInterval决定）。然后由spark streaming提交的job构建BlockRdd，最终以spark core任务的形式运行。

关于receiver方式，有以下几点需要注意：

- receiver作为一个常驻线程调度到executor上运行，占用一个cpu。
- receiver个数由KafkaUtils.createStream调用次数决定，一次一个receiver。
- kafka中的topic分区并不能关联产生在spark streaming中的rdd分区
  增加在KafkaUtils.createStream()中的指定的topic分区数，仅仅增加了单个receiver消费的topic的线程数，它不会增加处理数据中的并行的spark的数量。
  【topicMap[topic,num_threads]map的value对应的数值是每个topic对应的消费线程数】
- receiver默认200ms生成一个block，建议根据数据量大小调整block生成周期。
- receiver接收的数据会放入到BlockManager，每个executor都会有一个BlockManager实例，由于数据本地性，那些存在receiver的executor会被调度执行更多的task，就会导致某些executor比较空闲。

  建议通过参数spark.locality.wait调整数据本地性。该参数设置的不合理，比如设置为10而任务2s就处理结束，就会导致越来越多的任务调度到数据存在的executor上执行，导致任务执行缓慢甚至失败（要和数据倾斜区分开）。
- 多个kafka输入的DStreams可以使用不同的groups、topics创建，使用多个receivers接收处理数据。
- 两种receiver：

  可靠的receiver：可靠的receiver在接收到数据并通过复制机制存储在spark中时准确的向可靠的数据源发送ack确认。

  不可靠的receiver：不可靠的receiver不会向数据源发送数据已接收确认。这适用于用于不支持ack的数据源。
  当然，我们也可以自定义receiver。

- receiver处理数据可靠性默认情况下，receiver是可能丢失数据的
  可以通过设置spark.streaming.receiver.writeAheadLog.enable为true开启预写日志机制，将数据先写入一个可靠地分布式文件系统如hdfs，确保数据不丢失，但会失去一定性能。

- 限制消费者消费的最大速率，涉及三个参数：

  spark.streaming.backpressure.enabled：默认是false，设置为true，就开启了背压机制。

  spark.streaming.backpressure.initialRate：默认没设置初始消费速率，第一次启动时每个receiver接收数据的最大值。

spark.streaming.receiver.maxRate：默认值没设置，每个receiver接收数据的最大速率（每秒记录数）。每个流每秒最多将消费此数量的记录，将此配置设置为0或负数将不会对最大速率进行限制。

- 在产生job时，会将当前job有效范围内的所有block组成一个BlockRDD，一个block对应一个分区。
- kafka082版本消费者高阶API中，有分组的概念，建议使消费者组内的线程数（消费者个数）和kafka分区数保持一致。如果多于分区数，会有部分消费者处于空闲状态。

Direct Approach

direct approach是spark streaming不使用receiver集成kafka的方式，一般在企业生产环境中使用较多。相较于receiver，有以下特点：

1. 不使用receiver

a. 不需要创建多个kafka streams并聚合它们。

b. 减少不必要的CPU占用。

c. 减少了receiver接收数据写入BlockManager，然后运行时再通过blockId、网络传输、磁盘读取等来获取数据的整个过程，提升了效率。

d. 无需wal，进一步减少磁盘IO操作。

2. direct方式生的rdd是KafkaRDD，它的分区数与kafka分区数保持一致一样多的rdd分区来消费，更方便我们对并行度进行控制。

   注意：在shuffle或者repartition操作后生成的rdd，这种对应关系会失效。

3. 可以手动维护offset，实现exactly once语义。

4. 数据本地性问题。在KafkaRDD在compute函数中，使用SimpleConsumer根据指定的topic、分区、offset去读取kafka数据。

   但在010版本后，又存在假如kafka和spark处于同一集群存在数据本地性的问题。

5. 限制消费者消费的最大速率spark.streaming.kafka.maxRatePerPartition：从每个kafka分区读取数据的最大速率（每秒记录数）。这是针对每个分区进行限速，需要事先知道kafka分区数，来评估系统的吞吐量。

14. 数据质量如何监控?

数据质量管理是对数据从计划、获取、存储、共享、维护、应用、消亡生命周期的每个阶段里可能引发的数据质量问题，进行识别、度量、监控、预警等，通过改善了提高组织的管理水平使数据质量进一步提高。

数据质量管理是一个集方法论、技术、业务和管理为一体的解决方案。放过有效的数据质量控制手段，进行数据的管理和控制，消除数据质量问题，从而提高企业数据变现的能力。

会遇到的数据质量问题：数据真实性、数据准确性、数据一致性、数据完整性、数据唯一性、数据关联性、数据及时性。

算法题

1. 创建bean对象的三种方式

第一种方式：使用默认构造函数创建。
在spring中的配置文件中，使用bean标签，配以id和class属性之后，且没有其他标签时，采用的就是默认构造函数创建bean对象，
此时类中没有默认构造函数，则对象无法创建。

第二种方式：通过静态工厂创建bean对象。工厂类中提供一个静态方法，可以返回要用的bean对象。

第三种方式：通过静态工厂创建bean对象。工厂类中提供一个普通方法，可以返回要用的bean对象。

2. SpringBoot自动装配

先看看SpringBoot的主配置类：

```
@SpringBootApplication
public class DemosbApplication {

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


里面有一个main方法运行了一个run()方法，在run方法中必须要传入一个被@SpringBootApplication注解的类。

@SpringBootApplication

SpringBoot应用标注在某个类上说明这个类是SpringBoot的主配置类，SpringBoot就会运行这个类的main方法来启动SpringBoot项目。

那@SpringBootApplication注解到底是什么呢，点进去看看：

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(
    excludeFilters = {@Filter(
    type = FilterType.CUSTOM,
    classes = {TypeExcludeFilter.class}
), @Filter(
    type = FilterType.CUSTOM,
    classes = {AutoConfigurationExcludeFilter.class}
)}
)
public @interface SpringBootApplication {
    @AliasFor(
```


发现@@SpringBootApplication是一个组合注解。

@SpringBootConfiguration

先看看@SpringBootConfiguration注解：

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {
}
```


这个注解很简单，表名该类是一个Spring的配置类。

再进去看看@Configuration：

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {
    @AliasFor(
        annotation = Component.class
    )
    String value() default "";
}
```


说明Spring的配置类也是Spring的一个组件。

@EnableAutoConfiguration

这个注解是开启自动配置的功能。

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import({AutoConfigurationImportSelector.class})
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

    Class<?>[] exclude() default {};

    String[] excludeName() default {};
}
```


先看看@AutoConfigurationPackage注解：

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Import({Registrar.class})
public @interface AutoConfigurationPackage {
}
```


这个注解是自动配置包，主要是使用的@Import来给Spring容器中导入一个组件 ， 这里导入的是Registrar.class。

来看下这个Registrar：

```java
    static class Registrar implements ImportBeanDefinitionRegistrar, DeterminableImports {
        Registrar() {
        }

        public void registerBeanDefinitions(AnnotationMetadata metadata, BeanDefinitionRegistry registry) {
            AutoConfigurationPackages.register(registry, new String[]{(new AutoConfigurationPackages.PackageImport(metadata)).getPackageName()});
        }

        public Set<Object> determineImports(AnnotationMetadata metadata) {
            return Collections.singleton(new AutoConfigurationPackages.PackageImport(metadata));
        }
    }
}
```


就是通过这个方法获取扫描的包路径，可以debug看看：

在这行代码上打了一个断点：

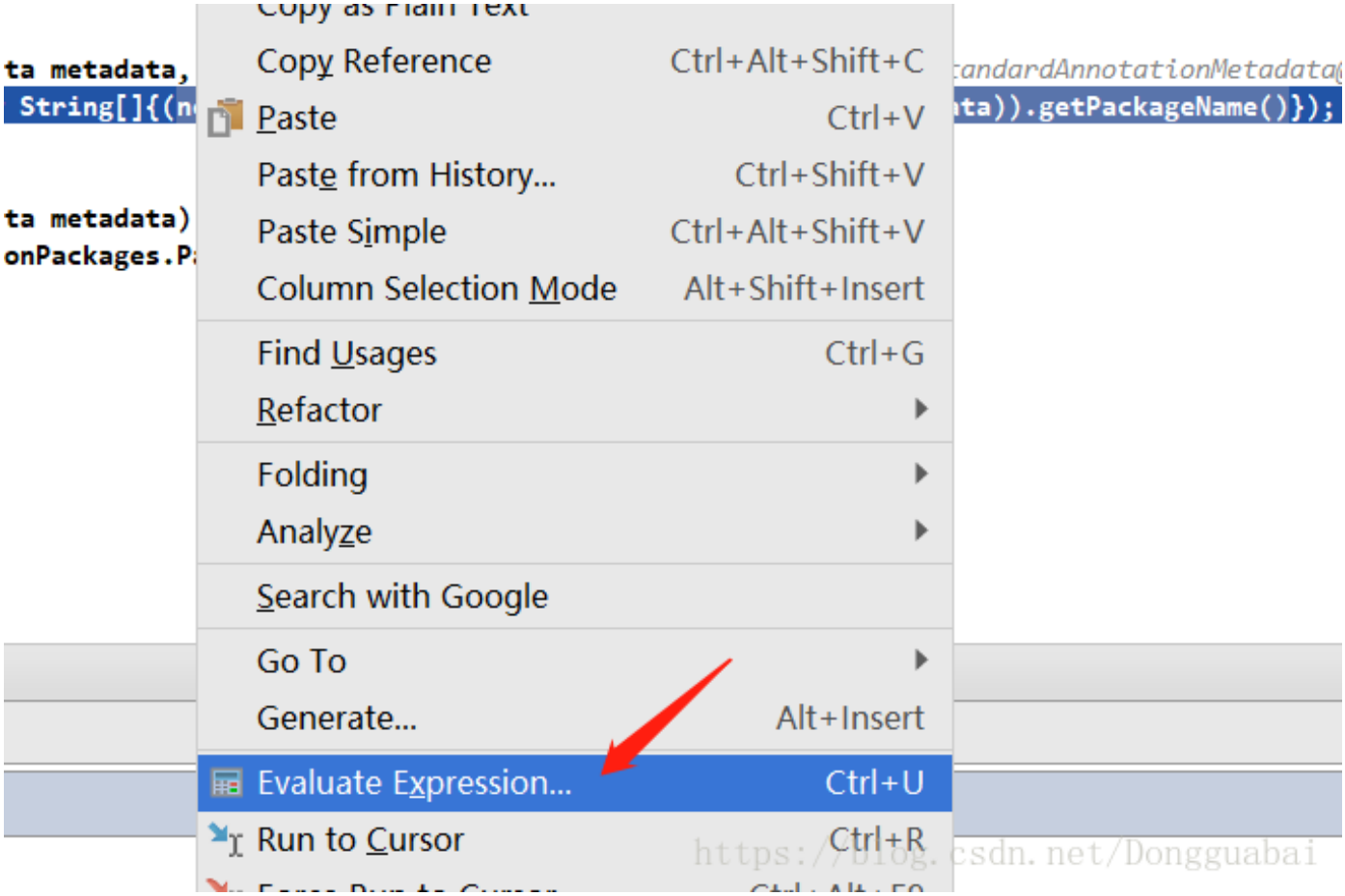
```java
        }

        public void registerBeanDefinitions(AnnotationMetadata metadata, BeanDefinitionRegistry registry) {
            AutoConfigurationPackages.register(registry, new String[]{(new AutoConfigurationPackages.PackageImport(metadata)).getPackageNam
        }

        public Set<Object> determineImports(AnnotationMetadata metadata) {
```


启动项目：

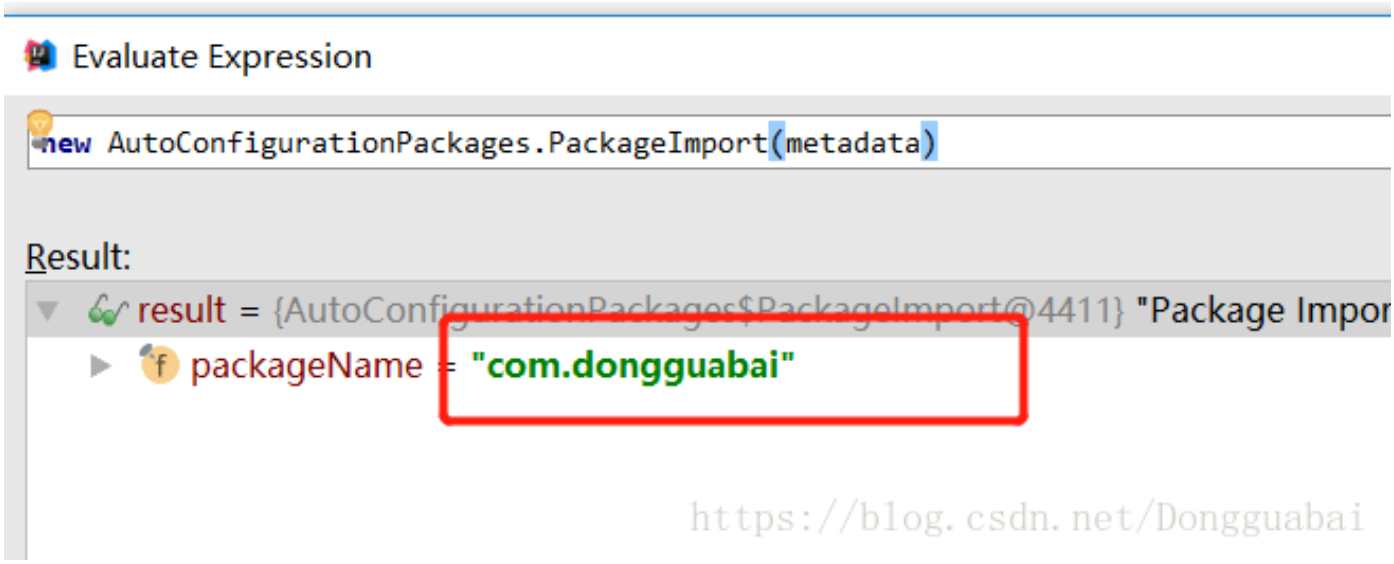
进入断点处：

```java
        }

        public void registerBeanDefinitions(AnnotationMetadata metadata, BeanDefinitionRegistry registry) {  metadata: StandardAnnotationMetadata@3704  registry: "org
            AutoConfigurationPackages.register(registry, new String[]{(new AutoConfigurationPackages.PackageImport(metadata)).getPackageName()});  registry: "org.spri
        }

        public Set<Object> determineImports(AnnotationMetadata metadata) {
            return Collections.singleton(new AutoConfigurationPackages.PackageImport(metadata));
```


看看能否获取扫描的包路径：

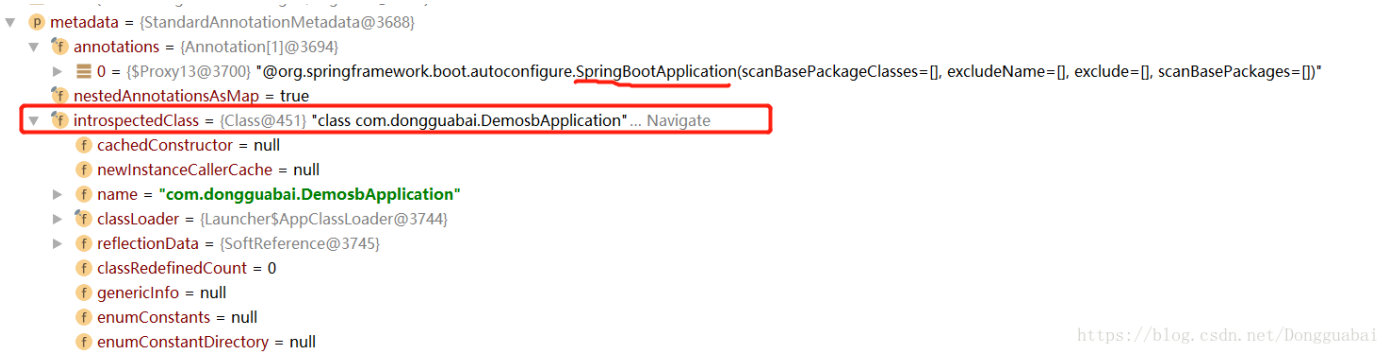


已经获取到了包路径：



那那个metadata是什么呢：

可以看到是标注在@SpringBootApplication注解上的DemosbApplication，也就是我们的主配置类：



说白了就是将主配置类（即@SpringBootApplication标注的类）的所在包及子包里面所有组件扫描加载到Spring容器。所以包名一定要注意。

现在包扫描路径获取到了，那具体加载哪些组件呢，看看下面这个注解。

@Import({AutoConfigurationImportSelector.class})

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import({AutoConfigurationImportSelector.class})
public @interface EnableAutoConfiguration {
    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

    Class<?>[] exclude() default {};

    String[] excludeName() default {};
}
```


@Import注解就是给Spring容器中导入一些组件，这里传入了一个组件的选择器:AutoConfigurationImportSelector。

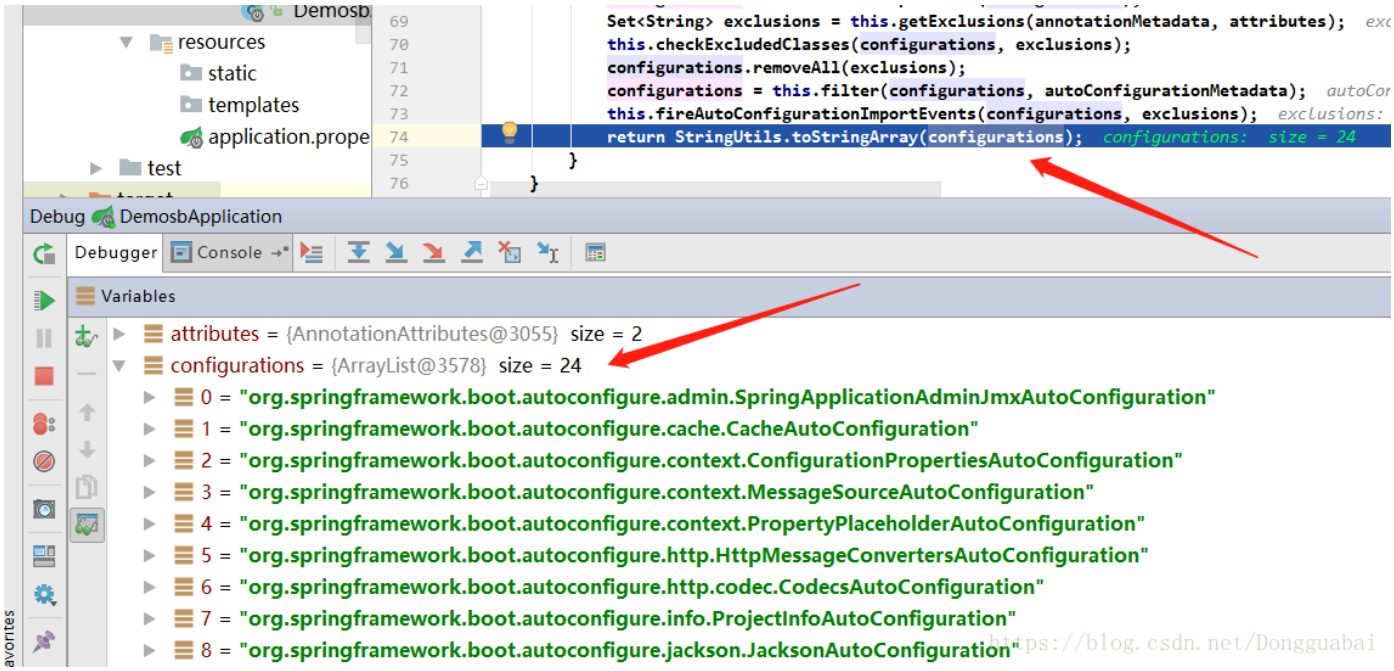
里面有一个selectImports方法，将所有需要导入的组件以全类名的方式返回；这些组件就会被添加到容器中。

```
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if(!this.isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    } else {
        AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader.loadMetadata(this.beanClassLoader);
        AnnotationAttributes attributes = this.getAttributes(annotationMetadata);
        List<String> configurations = this.getCandidateConfigurations(annotationMetadata, attributes);
        configurations = this.removeDuplicates(configurations);
        Set<String> exclusions = this.getExclusions(annotationMetadata, attributes);
        this.checkExcludedClasses(configurations, exclusions);
        configurations.removeAll(exclusions);
        configurations = this.filter(configurations, autoConfigurationMetadata);
        this.fireAutoConfigurationImportEvents(configurations, exclusions);
        return StringUtils.toStringArray(configurations);
    }
}
```


debug运行看看：

会给容器中导入非常多的自动配置类（xxxAutoConfiguration）；就是给容器中导入这个场景需要的所有组件，并配置好这些组件:

```
            configurations.removeAll(exclusions);
            configurations = this.filter(configurations, autoConfigurationMetadata);  autoConfigu
            this.fireAutoConfigurationImportEvents(configurations, exclusions);  exclusions:  size
            return StringUtils.toStringArray(configurations);  configurations:  size = 24
        }
    }
```

有了自动配置类，免去了我们手动编写配置注入功能组件等的工作。

那他是如何获取到这些配置类的呢，看看上面这个方法：

```
public String[] selectImports(AnnotationMetadata annotationMetadata) {  annotationMetadata: StandardAnnotati
    if(!this.isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    } else {
        AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader.loadMetadata(t
        AnnotationAttributes attributes = this.getAttributes(annotationMetadata);  attributes:  size = 2
        List<String> configurations = this.getCandidateConfigurations(annotationMetadata, attributes);
        configurations = this.removeDuplicates(configurations);
        Set<String> exclusions = this.getExclusions(annotationMetadata, attributes);  exclusions:  size = 0
        this.checkExcludedClasses(configurations, exclusions);
        configurations.removeAll(exclusions);
        configurations = this.filter(configurations, autoConfigurationMetadata);  autoConfigurationMetadata:
        this.fireAutoConfigurationImportEvents(configurations, exclusions);  exclusions:  size = 0
        return StringUtils.toStringArray(configurations);  configurations:  size = 24
    }
```



```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(this.getSpringFactoriesLoaderFactoryClass(), this.getBeanClassLoader());
    Assert.notEmpty(configurations,  message: "No auto configuration classes found in META-INF/spring.factories. If you are using a custom packaging
    return configurations;
}
```



```
public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable ClassLoader classLoader) {
    String factoryClassName = factoryClass.getName();
    return (List)loadSpringFactories(classLoader).getOrDefault(factoryClassName, Collections.emptyList());
}
```

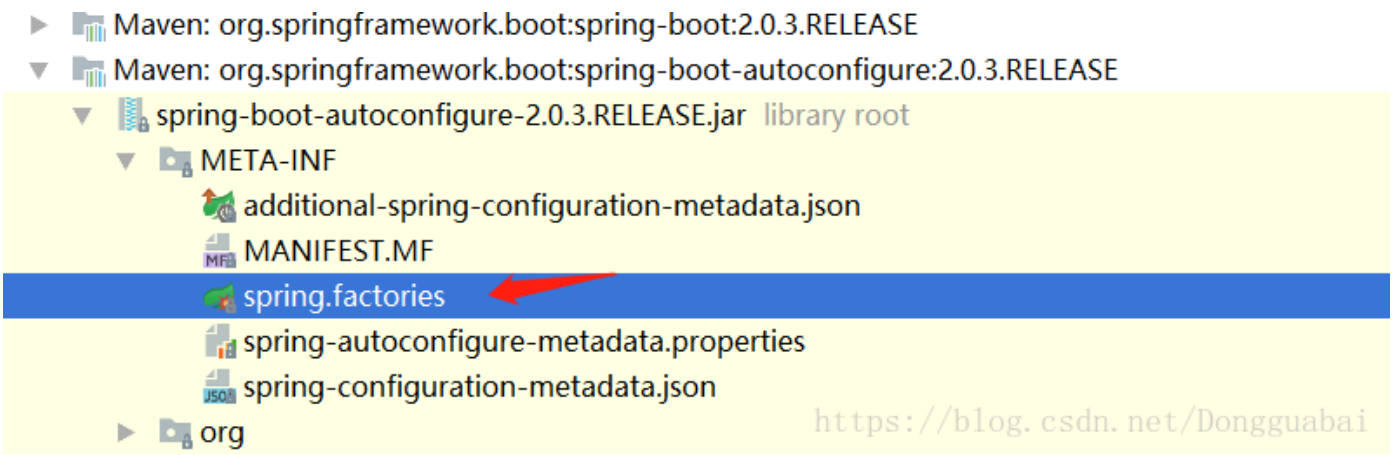

会从META-INF/spring.factories中获取资源，然后通过Properties加载资源：

```
private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader classLoader) {
    MultiValueMap<String, String> result = (MultiValueMap)cache.get(classLoader);
    if(result != null) {
        return result;
    } else {
        try {
            Enumeration<URL> urls = classLoader != null?classLoader.getResources( name: "META-INF/spring.factories"):ClassLoader.getSystemResources( name: "META-INF/spring.factories");
            LinkedMultiValueMap result = new LinkedMultiValueMap();

            while(urls.hasMoreElements()) {
                URL url = (URL)urls.nextElement();
                UrlResource resource = new UrlResource(url);
                Properties properties = PropertiesLoaderUtils.loadProperties(resource);
                Iterator var6 = properties.entrySet().iterator();

                while(var6.hasNext()) {
                    Entry<?, ?> entry = (Entry)var6.next();
                    List<String> factoryClassNames = Arrays.asList(StringUtils.commaDelimitedListToStringArray((String)entry.getValue()));
                    result.addAll((String)entry.getKey(), factoryClassNames);
                }
            }

            cache.put(classLoader, result);
            return result;
        } catch (IOException var9) {
            throw new IllegalArgumentException("Unable to load factories from location [META-INF/spring.factories]", var9);
        }
```


Spring Boot在启动的时候从类路径下的META-INF/spring.factories中获取EnableAutoConfiguration指定的值，将这些值作为自动配置类导入到容器中，自动配置类就生效，帮我们进行自动配置工作。以前我们需要自己配置的东西，自动配置类都帮我们完成了。

```
)
@ConditionalOnClass({Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class})
@ConditionalOnMissingBean({WebMvcConfigurationSupport.class})
@AutoConfigureOrder(-2147483638)
@AutoConfigureAfter({DispatcherServletAutoConfiguration.class, ValidationAutoConfiguration.class})
public class WebMvcAutoConfiguration {
    public static final String DEFAULT_PREFIX = "";
    public static final String DEFAULT_SUFFIX = "";
    private static final String[] SERVLET_LOCATIONS = new String[]{"/"};

    public WebMvcAutoConfiguration() {
    }

    @Bean
    @ConditionalOnMissingBean({HiddenHttpMethodFilter.class})
    public OrderedHiddenHttpMethodFilter hiddenHttpMethodFilter() { return new OrderedHiddenHttpMethodFilter(); }

    @Bean
    @ConditionalOnMissingBean({HttpPutFormContentFilter.class})
    @ConditionalOnProperty(
        prefix = "spring.mvc.formcontent.putfilter",
        name = {"enabled"},
        matchIfMissing = true
    )
```


J2EE的整体整合解决方案和自动配置都在spring-boot-autoconfigure-2.0.3.RELEASE.jar：



```
org.springframework.boot.autoconfigure.security.servlet.UserDetailsServiceAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityFilterAutoConfiguration,\
org.springframework.boot.autoconfigure.security.reactive.ReactiveSecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.reactive.ReactiveUserDetailsServiceAutoConfiguration,\
org.springframework.boot.autoconfigure.sendgrid.SendGridAutoConfiguration,\
org.springframework.boot.autoconfigure.session.SessionAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.client.OAuth2ClientAutoConfiguration,\
org.springframework.boot.autoconfigure.solr.SolrAutoConfiguration,\
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.jta.JtaAutoConfiguration,\
org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration,\
org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.web.embedded.EmbeddedWebServerFactoryCustomizerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.HttpHandlerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.ReactiveWebServerFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.error.ErrorWebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.reactive.WebSocketReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketServletAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketMessagingAutoConfiguration,\
org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfiguration

# Failure analyzers
org.springframework.boot.diagnostics.FailureAnalyzer=\
```


比如看看WebMvcAutoConfiguration：

都已经帮我们配置好了，我们不用再单独配置了：

```
)
@ConditionalOnClass({Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class})
@ConditionalOnMissingBean({WebMvcConfigurationSupport.class})
@AutoConfigureOrder(-2147483638)
@AutoConfigureAfter({DispatcherServletAutoConfiguration.class, ValidationAutoConfiguration.class})
public class WebMvcAutoConfiguration {
    public static final String DEFAULT_PREFIX = "";
    public static final String DEFAULT_SUFFIX = "";
    private static final String[] SERVLET_LOCATIONS = new String[]{"/"};

    public WebMvcAutoConfiguration() {
    }

    @Bean
    @ConditionalOnMissingBean({HiddenHttpMethodFilter.class})
    public OrderedHiddenHttpMethodFilter hiddenHttpMethodFilter() { return new OrderedHiddenHttpMethodFilter(); }

    @Bean
    @ConditionalOnMissingBean({HttpPutFormContentFilter.class})
    @ConditionalOnProperty(
        prefix = "spring.mvc.formcontent.putfilter",
        name = {"enabled"},
        matchIfMissing = true
    )
```

3. Aop如何处理全局异常

使用@AfterThrowing异常通知:

注：使用异常通知，不会完全处理异常，异常会向上继续传递给调用者。

1.1自定义注解：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Documented
public @interface ProcessException {
  Class<? extends Throwable>[] value() default {};
}
```

1.2编写切面

```
@Component
@Aspect
public class UserAspect {
        @Pointcut("@within(org.springframework.stereotype.Controller)")
  private void exceptionProcesser() {}
      @AfterThrowing(value = "exceptionProcesser()", throwing = "e")
  public void afterThrowingMethod(JoinPoint point, Throwable e) {
    Class<?> clazz = point.getTarget().getClass();
    Method[] methods = clazz.getMethods();
    for (Method m : methods) {
      ProcessException anno = m.getAnnotation(ProcessException.class);
      if (anno != null) {
        Class<? extends Throwable>[] exArr = anno.value();
        if (exArr.length == 0) {
          if (e instanceof RuntimeException) {
            try {
              m.invoke(clazz.newInstance(), e);
            } catch(Exception ex) {
              e.printStackTrace();
            }
          }
        } else {
          for (Class<? extends Throwable> exClass : exArr) {
            if (exClass.isInstance(e)) {
              try {
                m.invoke(clazz.newInstance(), e);
              } catch(Exception ex) {
                e.printStackTrace();
              }
            }
          }
```

```
        }
        break;
      }
    }
  }
}
```

1.3写全局异常处理方法：

```
public abstract class BaseController {
    @ProcessException(ServiceException.class)
    public ResponseResult<Void> handleException(Throwable e) {
      System.out.println(e.getMessage());
        ResponseResult<Void> result = new ResponseResult<>();
        result.setMessage(e.getMessage());
        if (e instanceof UsernameDuplicateKeyException) {
            result.setState(4001);
        } else if (e instanceof InsertException) {
            result.setState(4002);
        } else if (e instanceof UserNotFoundException) {
            result.setState(4003);
        } else if (e instanceof PasswordNotMatchException) {
            result.setState(4004);
        } else if (e instanceof UpdateException) {
            result.setState(4005);
        }
        return result;
    }
}
```

```
@Controller
@RequestMapping("/users")
public class UserController extends BaseController {

@Autowired
private IUserService userService;

@PostMapping("/login")
@ResponseBody
public ResponseResult<Void> login(String username, String password, HttpSession
session) {
  User user = userService.login(username, password);
  session.setAttribute("uid", user.getUid());
          session.setAttribute("username", user.getUsername());
```

```
  return new ResponseResult<>(SUCCESS);
}
}
```

```
@Data
public class ResponseResult<T> implements Serializable {
    private static final long serialVersionUID = 8011176026667744133L;
private Integer state;
private String message;
private T data;

public ResponseResult() {
}

public ResponseResult(Integer state) {
    this.state = state;
}
public ResponseResult(Integer state, T data) {
    this.state = state;
    this.data = data;
}
```

2、使用Around环绕通知：

1.1 自定义注解：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@Documented
public @interface ProcessException {
  Class<? extends Throwable>[] value() default {};
}
```

1.2 编写切面

```
@Component
@Aspect
public class UserAspect {
        @Pointcut("@within(org.springframework.stereotype.Controller)")
  private void exceptionProcesser() {}
        @Around(value="exceptionProcesser()")
  public Object around(ProceedingJoinPoint jp) throws Throwable {
    try {
      return jp.proceed();
    } catch (Throwable e) {
      try {
```

```
        Class<?> clazz = jp.getTarget().getClass();
        Method[] methods = clazz.getMethods();
        for (Method m : methods) {
          ProcessException anno = m.getAnnotation(ProcessException.class);
          if (anno != null) {
            Class<? extends Throwable>[] exArr = anno.value();
            if (exArr.length == 0) {
              if (e instanceof RuntimeException) {
                return m.invoke(clazz.newInstance(), e);
              }
            } else {
              for (Class<? extends Throwable> exClass : exArr) {
                if (exClass.isInstance(e)) {
                  return m.invoke(clazz.newInstance(), e);
                }
              }
            }
          }
        }
        throw e;
      } catch (Exception ex) {
        throw ex;
      }
    }
  }
}
```

1.3 写全局异常处理方法：

```
public abstract class BaseController {
    @ProcessException(ServiceException.class)
    public ResponseResult<Void> handleException(Throwable e) {
      System.out.println(e.getMessage());
        ResponseResult<Void> result = new ResponseResult<>();
        result.setMessage(e.getMessage());
        if (e instanceof UsernameDuplicateKeyException) {
            result.setState(4001);
        } else if (e instanceof InsertException) {
            result.setState(4002);
        } else if (e instanceof UserNotFoundException) {
            result.setState(4003);
        } else if (e instanceof PasswordNotMatchException) {
            result.setState(4004);
        } else if (e instanceof UpdateException) {
            result.setState(4005);
        }
        return result;
```

```
    }
}
```

```
@Controller
@RequestMapping("/users")
public class UserController extends BaseController {

@Autowired
private IUserService userService;

@PostMapping("/login")
@ResponseBody
public ResponseResult<Void> login(String username, String password, HttpSession
session) {
  User user = userService.login(username, password);
  session.setAttribute("uid", user.getUid());
          session.setAttribute("username", user.getUsername());
  return new ResponseResult<>(SUCCESS);
}
```

```
@Data
public class ResponseResult<T> implements Serializable {
    private static final long serialVersionUID = 8011176026667744133L;
private Integer state;
private String message;
private T data;

public ResponseResult() {
}

public ResponseResult(Integer state) {
    this.state = state;
}
public ResponseResult(Integer state, T data) {
    this.state = state;
    this.data = data;
}
}
```

4. 堆

堆是计算机科学中一类特殊的数据结构的统称，堆通常可以被看做是一棵**完全二叉树**的**数组**对象。

- 如果一个结点的位置为**k**，则它的父结点的位置为[**k/2**],而它的两个子结点的位置则分别为**2k**和**2k+1**。
- 每个结点都大于等于它的两个子结点。

5. 页面置换算法

首先看一下什么是页面置换算法：地址映射过程中，若在页面中发现所要访问的页面不在内存中，则产生缺页中断。当发生缺页中断时，如果操作系统内存中没有空闲页面，则操作系统必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。而用来选择淘汰哪一页的规则叫做页面置换算法。

1．最佳置换算法（OPT）（理想置换算法）： 从主存中移出永远不再需要的页面；如无这样的页面存在，则选择最长时间不需要访问的页面。于所选择的被淘汰页面将是以后永不使用的，或者是在最长时间内不再被访问的页面，这样可以保证获得最低的缺页率。 即被淘汰页面是以后永不使用或最长时间内不再访问的页面。

2．先进先出置换算法（FIFO）：是最简单的页面置换算法。这种算法的基本思想是：当需要淘汰一个页面时，总是选择驻留主存时间最长的页面进行淘汰，即先进入主存的页面先淘汰。其理由是：最早调入主存的页面不再被使用的可能性最大。 即优先淘汰最早进入内存的页面。

3．最近最久未使用（LRU）算法：这种算法的基本思想是：利用局部性原理，根据一个作业在执行过程中过去的页面访问历史来推测未来的行为。它认为过去一段时间里不曾被访问过的页面，在最近的将来可能也不会再被访问。所以，这种算法的实质是：当需要淘汰一个页面时，总是选择在最近一段时间内最久不用的页面予以淘汰。即淘汰最近最长时间未访问过的页面。

4. 时钟(CLOCK)置换算法：

LRU算法的性能接近于OPT,但是实现起来比较困难，且开销大；FIFO算法实现简单，但性能差。所以操作系统的设计者尝试了很多算法，试图用比较小的开销接近LRU的性能，这类算法都是CLOCK算法的变体。
简单的CLOCK算法是给每一帧关联一个附加位，称为使用位。当某一页首次装入主存时，该帧的使用位设置为1;当该页随后再被访问到时，它的使用位也被置为1。对于页替换算法，用于替换的候选帧集合看做一个循环缓冲区，并且有一个指针与之相关联。当某一页被替换时，该指针被设置成指向缓冲区中的下一帧。当需要替换一页时，操作系统扫描缓冲区，以查找使用位被置为0的一帧。每当遇到一个使用位为1的帧时，操作系统就将该位重新置为0；如果在这个过程开始时，缓冲区中所有帧的使用位均为0，则选择遇到的第一个帧替换；如果所有帧的使用位均为1,则指针在缓冲区中完整地循环一周，把所有使用位都置为0，并且停留在最初的位置上，替换该帧中的页。由于该算法循环地检查各页面的情况，故称为CLOCK算法，又称为最近未用(Not Recently Used, NRU)算法。

6. 银行家算法

银行家算法（Banker's Algorithm）是一个避免死锁（Deadlock）的著名算法，是由艾兹格·迪杰斯特拉在1965年为T.H.E系统设计的一种避免死锁产生的算法。它以银行借贷系统的分配策略为基础，判断并保证系统的安全运行。

在银行中，客户申请贷款的数量是有限的，每个客户在第一次申请贷款时要声明完成该项目所需的最大资金量，在满足所有贷款要求时，客户应及时归还。银行家在客户申请的贷款数量不超过自己拥有的最大值时，都应尽量满足客户的需要。在这样的描述中，银行家就好比操作系统，资金就是资源，客户就相当于要申请资源的进程。

银行家算法是一种最有代表性的避免死锁的算法。在避免死锁方法中允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次分配资源的安全性，若分配不会导致系统进入不安全状态，则分配，否则等待。为实现银行家算法，系统必须设置若干数据结构。

银行家算法中的数据结构

为了实现银行家算法，在系统中必须设置这样四个数据结构，分别用来描述系统中可利用的资源、所有进程对资源的最大需求、系统中的资源分配，以及所有进程还需要多少资源的情况。

(1) 可利用资源向量 Available。这是一个含有 m 个元素的数组，其中的每一个元素代表一类可利用的资源数目，其初始值是系统中所配置的该类全部可用资源的数目，其数值随该类资源的分配和回收而动态地改变。如果Available[j] = K，则表示系统中现Rj类资源K个。

(2) 最大需求矩阵Max。这是一个n x m的矩阵，它定义了系统中n个进程中的每个进程对m类资源的最大需求。如果Max[i,j] = K，则表示进程i需要Rj 类资源的最大数目为K。

(3) 分配矩阵 Allocation。这也是一个n x m的矩阵，它定义了系统中每一类资源当前已分配给每一进程的资源数。如果 Allocation[i,jl = K，则表示进程i当前己分得Rj类资源的数目为K。

(4) 需求矩阵Need.这也是一个n×m的矩阵，用以表示每一个进程尚需的各类资源数。如果Need[i,j] = K，则表示进程i还需要Rj类资源K个方能完成其任务。

上述三个矩阵间存在下述关系:

Need[i,j] = Max[i,j] - allocation[i, j]

注：资料来源于网络。