
es2019 提案预览

ES2019 提案预览

之 Angular2 特性

摘要

Angular2 特性——pipeline

@link <https://github.com/tc39/proposal-pipeline-operator>

Angular2 特性——Observable

@link <https://github.com/tc39/proposal-observable>

Angular2 特性——export

@link <https://github.com/tc39/proposal-export-default-from>

Angular2 特性——访问控制符

@link <https://github.com/tc39/proposal-private-declarations>

Angular2 特性——Declarations in Conditionals

@link

<https://github.com/tc39/proposal-Declarations-in-Conditionals>

正文

什么是流水线（Pipeline）？

Pipeline 是 Linux 系统 Shell 语言的一个特性，能够将若干处理函数用 vertical bar 相连，并顺序执行。Angular2 在模板引擎中加入 pipeline 特性，比如国际化中的 translate 指令。示例如下：

```
function doubleSay (str) {  
  return str + ", " + str;  
}  
function capitalize (str) {  
  return str[0].toUpperCase() + str.substring(1);  
}  
function exclaim (str) {  
  return str + '!';  
}  
let result = exclaim(capitalize(doubleSay("hello")));  
result //=> "Hello, hello!"  
  
let result = "hello"  
  |> doubleSay  
  |> capitalize  
  |> exclaim;  
  
result //=> "Hello, hello!"
```

什么是 Observable?

Angular2 扩展模块@angular/rxjs 中提供了用于异步调用的 Observable 对象和用于同步调用的 Promise 对象。该提案建议将 Observable 引入 JavaScript, 示例如下:

```
let subscription = commandKeys(inputElement).subscribe({
  next(val) { console.log("Received key command: " + val) },
  error(err) { console.log("Received an error: " + err) },
  complete() { console.log("Stream complete") },
});
```

什么是 export?

该提案建议 export 关键字支持导出 ModuleNameSpace, 示例如下:

```
export someIdentifier from "someModule";
export someIdentifier, { namedIdentifier } from "someModule";
```

访问控制符

TypeScript 支持 private、protected、public 等访问控制符。该提案建议将访问控制符特性引入 JS, 示例如下:

```
// https://github.com/Polymer/lit-html/blob/1a51eb54/src/lib/parts.ts
```

```
private #createPart;
```

```
class AttributeCommitter {
  //...

  [#createPart]() {
    return new AttributePart(this);
  }
}
```

```
class PropertyCommitter extends AttributeCommitter {
  [#createPart]() {
    return new PropertyPart(this);
  }
}
```

Declarations in Conditionals

TypeScript 允许 Declarations in Conditionals 特性, 示例如下:

```
for(let k in valueMap){
  let param = {
    label: valueMap[k],
    value: k
  }
  arr.push(param);
}
```

```
}
```

JavaScript 不允许该特性:

```
class Foo {
  get data() {
    let result = [];
    /* ... do some expensive work ... */
    return result;
  }
}

let foo = new Foo;
if (foo.data) {
  for (let item of foo.data) {
    /* A */
  }
} else {
  /* B */
}
```

JS 添加该特性后, 上述代码可简写成:

```
class Foo {
  get data() {
    let result = [];
    /* ... do some expensive work ... */
    return result;
  }
}

let foo = new Foo;
if (let data = foo.data) {
  for (let item of data) {
    /* A */
  }
} else {
  /* B */
}
```

什么是代理?

JavaScript 作为一门编程语法,对属性(或者说是字段)没有提供原生的 setter & getter 函数.这些功能在其他的,类似于 Java,.Net,Objective-C 里都有提供.

所以,proxy 更像是 JavaScript 提供的一个统一的属性的 getter & setter 的监测接口.

作用:

-
1. 代理可以作为一个“漏斗”，定义一组公共的 getter、setter 方法，动态地附加到需要的对象上
 2. 代理可以用于封装原生的 HTML API.

□

ES2019 提案预览

之 WHATWG URL

摘要

JavaScript 语言特性——URL、URLSearchParams

@link <https://github.com/jasnell/proposal-url>

正文

已经没有 HTML5 的说法了

万维网联盟（W3C）和 Web 超文本应用技术工作组（WHATWG）签署了一项协议，该协议表明两个小组将合作开发单一版本的 HTML 和 DOM 规范。通过制定相同的规范，W3C 认为它对社区有益，因为开发人员在开发产品时可以遵循一套规则。也就是说，从 19 年 5 月起，没有 HTML5、HTML5.1 的说法了，统一名称为 HTML 规范。不加版本号有利于持续演进（反映出业界从瀑布模型向原型模型、敏捷开发的转变）。

URLSearchParams 类

在项目中存在这样的代码：

```
let url = this.getIntegrityDataUrl;  
url += "?data=" + encodeURIComponent(pData);  
url += "&isc_dataFormat=json";  
url += "&pageSize=100";  
url += "&pageNo=1";  
url += "&isc_flag=smartClient";  
url += "&_operationType=fetch";  
url += "&_startRow=0";  
url += "&_endRow=75";  
url += "&_textMatchStyle=exact";  
url += "&_componentId=isc_PageListTable_1";  
url += "&_dataSource=isc_PageRestDataSource_1";  
url += "&isc_metaDataPrefix=_";
```

出于 semantics 考虑，可以使用 WHATWG URL、URLSearchParams 特性，示例如下：

```
const base = new URL('http://example.org/foo');  
const url = new URL('bar', base);
```

```
.....  
var paramsString = "q=URLUtils.searchParams&topic=api"  
var searchParams = new URLSearchParams(paramsString);  
  
for (let p of searchParams) {  
    console.log(p);  
}  
  
searchParams.has("topic") === true; // true  
searchParams.get("topic") === "api"; // true  
searchParams.getAll("topic"); // ["api"]  
searchParams.get("foo") === null; // true  
searchParams.append("topic", "webdev");  
searchParams.toString(); // "q=URLUtils.searchParams&topic=api&topic=webdev"  
searchParams.set("topic", "More webdev");  
searchParams.toString(); // "q=URLUtils.searchParams&topic=More+webdev"  
searchParams.delete("topic");  
searchParams.toString(); // "q=URLUtils.searchParams"
```

□

ES2019 提案预览

之 Immutability（不可变性）

摘要

TypeScript 语言特性——Immutability（不可变性）

正文

什么是 Immutability（不可变性）？

不可变性是一类关键字与 Object 静态方法的总称，包括：

- readonly
- const
- Object.freeze()
- Object.seal().

不可变性的实现方法有什么不同？

const and Object.freeze() serve totally different purposes.

const is there for declaring a variable which has to assigned right away and can't be reassigned. variables declared by const are block scoped and not function scoped like variables declared with var

`Object.freeze()` is a method which accepts an object and returns the same object. Now the object cannot have any of its properties removed or any new properties added.

这个提案讲了什么？

这个提案建议 TypeScript 添加两个语法糖（sugar）：`##`和`||`，分别用于 `freeze` 和 `seal` 对象。示例如下：

```
const foo = {#
  a: {#
    b: {#
      c: {#
        d: {#
          e: [# "some string!" #]
        }#
      }#
    }#
  }#
}
```

```
const foo = {|
  a: {|
    b: {|
      c: {|
        d: {|
          e: [| "some string!" |]
        }|
      }|
    }|
  }|
}
```

怎样通过注解冻结对象？

```
function Frozen(constructor:Function){
  Object.freeze(constructor);
  Object.freeze(constructor.prototype)
}
```

```
@Frozen
export class IceCreamComponent{

}
```

```
console.log(Object.isFrozen(IceCreamComponent))
```

□

ES2019 提案预览

之反射

摘要

JavaScript 语言特性——代理、反射

@link

<https://github.com/caitp/TC39-Proposals/blob/master/tc39-reflect-isconstructor-iscallable.md>

正文

什么是反射？

反射机制指的是程序在运行时能够获取自身的信息。例如一个对象能够在运行时知道自己有哪些方法和属性。ES6 新标准中 `Reflect` 类定义了以下静态方法：

```
Reflect.apply()
Reflect.construct()
Reflect.defineProperty()
Reflect.deleteProperty()
Reflect.get()
Reflect.getOwnPropertyDescriptors()
Reflect.getPrototypeOf()
Reflect.has()
Reflect.isExtensible()
Reflect.ownKeys()
Reflect.set()
Reflect.setPrototypeOf()
```

□

ES2019 提案预览

之宽松语法

摘要

TypeScript 语言特性——宽松语法

@link <https://github.com/tc39/proposal-optional-catch-binding>

@link <https://github.com/tc39/proposal-trailing-function-commas>

正文

宽松语法

TypeScript 依赖于脚本语言 JavaScript 引擎，也因此继承了脚本语言宽松语法的基因。

缺省 catch 参数

此提议引入的语法更改允许省略 catch 绑定及其周围的括号：

```
try {  
  // ...  
} catch {  
  // ...  
}
```

函数尾随逗号

在某些代码库/样式指南中，会出现将函数调用和定义分成多行的情况。在这些情况下，当其他代码贡献者出现并向其中一个参数列表添加另一个参数时，它们必须进行两行更新。在对版本控制系统管理的代码（git, subversion, mercurial 等）进行此更改的过程中，第 3 行和第 9 行的非常规/注释代码历史记录信息将更新为指向添加逗号的人（而不是最初添加参数的人）。

为了帮助缓解此问题，某些其他语言（Python, D, Hack 等……可能还有其他……）添加了语法支持，以允许在这些参数列表中使用逗号结尾。这使代码提供者可以始终在这些每行参数列表之一中以尾随逗号结束参数添加，而不必担心代码归因问题。

□

ES2019 提案预览

之对象操作

摘要

TypeScript 语言特性——对象操作

@link <https://github.com/tc39/proposal-object-values-entries>

@link <https://github.com/tc39/proposal-object-from-entries>

@link

<https://github.com/tc39/proposal-object-getownpropertydescriptors>

正文

Object 与 Map 互转

使用 TypeScript 新特性 `Object.entries()`、`Object.fromEntries()` 可以实现 Object 与 Map 类型的相互转化。这个特性可以用于将 Object 中的属性进行变换，示例如下：

```
obj = { abc: 1, def: 2, ghij: 3 };
res = Object.fromEntries(
  Object.entries(obj)
    .filter(([ key, val ]) => key.length === 3)
    .map(([ key, val ]) => [ key, val * 2 ])
);
```

```
// res is { 'abc': 2, 'def': 4 }
```

一句话浅拷贝

使用 `Object.getOwnPropertyDescriptors()` 特性可以定义一句话浅拷贝函数，示例如下：

```
const shallowClone = (object) => Object.create(
  Object.getPrototypeOf(object),
  Object.getOwnPropertyDescriptors(object)
);

const shallowMerge = (target, source) => Object.defineProperties(
  target,
  Object.getOwnPropertyDescriptors(source)
);
```

□

ES2019 提案预览

之异步交互

摘要

Angular2 特性——异步交互

@link <https://github.com/tc39/proposal-promise-any>

@link <https://github.com/tc39/proposal-promise-allSettled>

@link <https://github.com/tc39/proposal-top-level-await>

正文

Promise.any()

Promise.any 是一个新的异步控制方法。如果一组中有一个 Promise 成功执行，那么返回该 Promise 执行结果。如果所有 Promise 都执行失败，那么返回一组错误原因。

Promise.allSettled()

There are four main combinators in the Promise landscape.

name	description	
Promise.allSettled	does not short-circuit	this proposal
Promise.all	short-circuits when an input value is rejected	added in ES2015 ✓
Promise.race	short-circuits when an input value is settled	added in ES2015 ✓
Promise.any	short-circuits when an input value is fulfilled	separate proposal

These are all commonly available in userland promise libraries, and they're all independently useful, each one serving different use cases.

A common use case for this combinator is wanting to take an action after multiple requests have completed, regardless of their success or failure. Other Promise combinators can short-circuit, discarding the results of input values that lose the race to reach a certain state. Promise.allSettled is unique in always waiting for all of its input values.

Promise.allSettled returns a promise that is fulfilled with an array of promise state snapshots, but only after all the original promises have settled, i.e. become either fulfilled or rejected.

Promise.allSettled() 相当于 Observer.forkJoin().

Top Level await

顶级等待使模块可以充当大型异步功能：使用顶级等待，ECMAScript 模块（ESM）可以等待资源，从而导致其他导入模块的模块在开始执行其主体之前等待。

Java 中可用并发工具类 CountdownLatch（java.util.concurrent）实现类似的功能。

□

ES2019 提案预览

之数字支持

摘要

TypeScript 语言特性——数字支持

@link <https://github.com/tc39/proposal-exponentiation-operator>

@link <https://github.com/tc39/proposal-bigint>

@link <https://github.com/tc39/proposal-numeric-separator>

@link <https://github.com/rwaldron/proposal-math-extensions>

正文

指数运算符

将 Python 语言中的指数运算符**引入 JavaScript，示例如下：

```
let squared = 2 ** 2;  
// same as: 2 * 2
```

```
let cubed = 2 ** 3;  
// same as: 2 * 2 * 2
```

新类型 BigInt

JavaScript 基础类型 Number 能够处理的最大整数为 2^{53} ，该提案提出新的类型 BigInt 来处理超大数字，示例如下：

```
const theBiggestInt = 9007199254740991n;  
  
const alsoHuge = BigInt(9007199254740991);  
// ↪ 9007199254740991n  
  
const hugeButString = BigInt('9007199254740991');  
// ↪ 9007199254740991n
```

数字分隔符

为增强代码中数字的可读性，该提案引入下划线作为数字分隔符。数字分隔符在编译时（对于 TypeScript）被剔除，是一个对于编译器没有意义的符号，示例如下：

```
1_000_000_000           // Ah, so a billion  
101_475_938.38          // And this is hundreds of millions  
  
let fee = 123_00;         // $123 (12300 cents, apparently)  
let fee = 12_300;         // $12,300 (woah, that fee!)  
let amount = 12345_00;    // 12,345 (1234500 cents, apparently)  
let amount = 123_4500;    // 123.45 (4-fixed financial)  
let amount = 1_234_500;   // 1,234,500
```

ES2019 提案预览

之正则表达式

摘要

JavaScript 语言特性——正则表达式

正文

命名捕获组 (Named Capture Groups)

带编号的捕获组使您可以引用正则表达式匹配的字符串的某些部分。每个捕获组都分配有一个唯一的编号，并且可以使用该编号进行引用，但这会使正则表达式难以理解和重构。

Numbered capture groups allow one to refer to certain portions of a string that a regular expression matches. Each capture group is assigned a unique number and can be referenced using that number, but this can make a regular expression hard to grasp and refactor.

正向断言 (Lookahead Assertions)

反向断言 (Lookbehind Assertions)

正向断言的意思是：当前位置后面的字符串应该满足断言，但是并不捕获。反向断言和正向断言的行为一样，只是方向相反。在当前的 JavaScript 正则表达式版本中，只支持正向断言。

Match Indices

示例如下：

```
const re1 = /a+(?<Z>z)?/;
```

```
// indices are relative to start of the input string:
```

```
const s1 = "xaaaz";
const m1 = re1.exec(s1);
m1.indices[0][0] === 1;
m1.indices[0][1] === 5;
s1.slice(...m1.indices[0]) === "aaaz";
```

String.prototype.matchAll

String.prototype.replaceAll

当前，不使用全局正则表达式无法替换字符串中子字符串的所有实例。与字符串参数一起使用时，`String.prototype.replace` 仅影响首次出现。有很多证据表明开发人员正在尝试在 JS 中执行此操作-请以成千上万的票数查看 [StackOverflow](#) 问题。Currently there is no way to replace all instances of a substring in a string without use of a global regexp. `String.prototype.replace` only affects the first occurrence when used

with a string argument. There is a lot of evidence that developers are trying to do this in JS — see the StackOverflow question with thousands of votes.

□

ES2019 提案预览

之生成器函数

摘要

JavaScript 语言特性——生成器函数

@link <https://github.com/tc39/proposal-function.sent>

@link

<https://github.com/tc39/proposal-generator-arrow-functions>

正文

什么是生成器函数？

Generator 函数是协程在 ES6 的实现，最大特点就是可以交出函数的执行权（即暂停执行）。它不同于普通函数，是可以暂停执行的，所以函数名之前要加星号，以示区别。整个 Generator 函数就是一个封装的异步任务，或者说是异步任务的容器。异步操作需要暂停的地方，都用 yield 语句注明。

调用一个生成器函数并不会马上执行它里面的语句，而是返回一个这个生成器的迭代器（iterator）对象。当这个迭代器的 next() 方法被首次（后续）调用时，其内的语句会执行到第一个（后续）出现 yield 的位置为止，yield 后紧跟迭代器要返回的值。或者如果用的是 yield*（多了个星号），则表示将执行权移交给另一个生成器函数（当前生成器暂停执行）。

生成器函数的应用场景——长轮询（Long Polling）

```
var fetch = require('node-fetch');
```

```
function* gen(){
  var url = 'https://api.github.com/users/github';
  var result = yield fetch(url);
  console.log(result.bio);
}
```

```
var g = gen();
var result = g.next();
```

```
result.value.then(function(data){
  return data.json();
}).then(function(data){
  g.next(data);
});
```

});

使用生成器函数实现长轮询与 `setInterval()/clearInterval()` 方式的不同在于：生成器函数在得到上一个请求的返回后，才会继续发出下一个请求。而 `setInterval()/clearInterval()` 定时发出一次请求，弱网络环境下会出现同时发出多个请求的情形，清除定时器的逻辑复杂。

Function.next

问题 Because there the first next call does not correspond to a yield within the generator function body there is currently no way for the code with the body to access the initial next argument.

解决 The value of `function.sent` within the body of a Generator Function is the value passed to the generator by the next method that most recently resumed execution of the generator. In particular, referencing `function.sent` prior to the first evaluation of a yield operator returns the argument value passed by the next call that started evaluation of the GeneratorBody.

□

ES2019 提案预览

之空值处理

摘要

TypeScript 语言特性——空位合并（Nullish Coalescing）

@link <https://github.com/tc39/proposal-nullish-coalescing>

TypeScript 语言特性——可选链（Optional Chaining）

@link <https://github.com/tc39/proposal-optional-chaining>

正文

什么是空值？为什么要处理空值？

在 TypeScript 中，空值包括 `null`、`undefined`、空字符串、空数组等。在项目中，为了兼容特性不完备的后端语言、框架，需要在前端做空值处理。有时为了确保代码的容错能力，不得不写一些垃圾代码，比如：

```
if (response == null) {
  throw ...
  return;
}
if (response.data == null) {
  throw ...
  return;
}
```

```
}  
let pShowDetails = response.data.showDetails;
```

什么是可选链？

当寻找树状结构深处的属性值时，通常需要检查中间节点是否存在，而且许多 API 返回对象或 `null / undefined`，并且可能只想从结果中提取属性 当它不为 `null` 时。可选的链接运算符允许开发人员处理许多情况，而无需重复自己和/或在临时变量中分配中间结果。

上例中，可以用 `short-circuit evaluation` 简化代码为：

```
let pShowDetails = response && response.data && response.data.showDetails;
```

而用可选链可以将代码进一步简化：

```
let pShowDetails = response?.data?.showDetails;
```

什么是空位合并？

执行属性访问时，通常需要提供默认值，如果该属性访问的结果为 `null` 或未定义。当前，在 `JavaScript` 中表达此意图的典型方法是使用 `||`。这在空值和未定义值的常见情况下效果很好，但是存在一些虚假值可能会产生令人惊讶的结果。空值合并运算符旨在更好地处理这些情况，并用作对空值的相等检查（`null` 或未定义）。

有什么体会？

编程语言一大抄，抄来抄去有提高。

□

ES2019 提案预览

之装饰器

摘要

TypeScript 语言特性——装饰器

@link <https://github.com/tc39/proposal-decorators>

正文

实验特性

装饰器（Decorator）是 TypeScript 的实验特性，需要在配置文件 `tsconfig.json` 中使能才可以使用：

```
"experimentalDecorators": true,
```

五类装饰器

装饰器的类型有：类装饰器、访问器装饰器、属性装饰器、方法装饰器、参数装饰器，但是没有函数装饰器(function)。

Angular 中的装饰器

Angular 中内置的装饰器如下：

类装饰器：@Component、@NgModule、@Pipe、@Injectable

属性装饰器：@Input、@Output、@ContentChild、@ContentChildren、@ViewChild、@ViewChildren

方法装饰器：@HostListener

参数装饰器：@Inject、@Optional、@Self、@SkipSelf、@Host

装饰器的执行顺序

有多个参数装饰器时：从最后一个参数依次向前执行

方法和方法参数中参数装饰器先执行。

类装饰器总是最后执行。

方法和属性装饰器，谁在前面谁先执行。因为参数属于方法一部分，所以参数会一直紧紧挨着方法执行。

□

ES2019 提案预览

之迭代器

摘要

JavaScript 语言特性——迭代器

@link <https://github.com/tc39/proposal-iterator-helpers>

JavaScript 新特性——Object 与 Array??相互转换

@link <https://github.com/tc39/proposal-object-map>

正文

什么是迭代器？

处理集合中的每个项是很常见的操作。JavaScript 提供了许多迭代集合的方法，从简单的 for 循环到 map() 和 filter()。迭代器和生成器将迭代的概念直接带入核心语言，并提供了一种机制来自定义 for...of 循环的行为。

在 JavaScript 中，迭代器是一个对象，它定义一个序列，并在终止时可能返回一个返回值。更具体地说，迭代器是通过使用 next() 方法实现 Iterator protocol 的任何对象，该方法返回具有两个属性的对象：value，这是序列中的 next 值；和 done，如果已经迭代到序列中的最后一个值，则它为 true。如果 value 和 done 一起存在，则它是迭代器的返回值。

JS 中有哪些 iterator helper？

forEach 迭代器

every 迭代器
some 迭代器
reduce 迭代器
map 迭代器
fiter 迭代器

怎样使用 iterator helper?

示例如下:

```
binding: {  
  data: this.kpis.map(kpi=>{  
    return {  
      kpi: kpi.name,  
      location: this.selectedPo.shortName  
    }  
  }  
}),  
}
```

这个提案讲了什么?

这个提案建议将 Python、Rust 中的迭代器引进 JS 中, 如 flatMap、findMap、filterMap 等。

Making mapping over Objects more concise

该特性建议 JavaScript 支持 Object 与 Array 类型的相互转换。这样做的优势在于: Object 对象可以充分使用丰富的 iterator helper 函数, 示例如下:

```
// Utility method to get an iterator on an Object  
Iterator.from(obj)  
  // Standard `map` function that operates on iterables  
  .map(([key, value]) => [transform(key), transform(value)])  
  // Collect the iterable data back into an Object, specifying key and value  
  selectors  
  .toObject(([key]) => key, ([, value]) => value);  
Note 这里改为 JSON 类型可能更为合适.
```

□

ES2019 提案预览

之集合操作

摘要

TypeScript 语言特性——集合操作

@link <https://github.com/tc39/Array.prototype.includes>

@link <https://github.com/tc39/proposal-collection-normalization>

@link <https://github.com/tc39/proposal-upsert>

正文

Array.prototype.includes

示例:

```
assert([1, 2, 3].includes(2) === true);
assert([1, 2, 3].includes(4) === false);
assert([1, 2, NaN].includes(NaN) === true);
assert([1, 2, -0].includes(+0) === true);
assert([1, 2, +0].includes(-0) === true);
assert(["a", "b", "c"].includes("a") === true);
assert(["a", "b", "c"].includes("a", 1) === false);
```

Collection {coerceKey, coerceValue}

由于 JavaScript 是弱类型语言, 因此及时 TypeScript 具有强类型特性, 在编译为 JS 后类型限制得不到保证。该提议基于此提出了强键值特性, 即显式声明 Map、Set 对象的键、值变量类型, 在执行 insert()、update()方法时进行类型检查或强制类型转换, 示例如下:

```
const map = new Map([], {
  coerceKey: String
}); // stored using { [[Key]]: "1", [[Value]]: "one" } in
map.[[MapData]]map.set(1, 'one'); // looks for corresponding { [[Key]]: "1" }
in map.[[MapData]]map.has(1); // true // functions directly exposing the
underlying entry list are unaffected
[...map.entries()]; // [["1", "one"]]
const set = new Set([], {coerceValue: JSON.stringify}); // stored using
{ [[Value]]: '{"path": "/foo"}' } in set.[[SetData]]set.add({path: '/foo'}); //
looks for corresponding { [[Value]]: '{"path": "/foo"}' } in
set.[[SetData]]set.has({path: '/foo'}); // functions directly exposing the
underlying entry list are unaffected
[...set]; // ['{"path": "/foo"}']
```

Map.prototype.upsert

We propose the addition of a method that will add a value to a map if the map does not already have something at key, and will also update an existing value at key. It's worthwhile having this API for the average case to cut down on lookups. It is also worthwhile for developer convenience and expression of intent.

□

摘要

JavaScript 语言特性——Static class features

@link <https://github.com/tc39/proposal-static-class-features/>

JavaScript 语言特性——Class field declarations for JavaScript

@link <https://github.com/tc39/proposal-class-fields>

JavaScript 语言特性——Private methods and getter/setters for JavaScript classes

@link <https://github.com/tc39/proposal-private-methods>

JavaScript 语言特性——ECMAScript class property access expressions

@link <https://github.com/tc39/proposal-class-access-expressions>

正文

JavaScript 语言面向对象

以前听人这样说，JavaScript 除了名字中包含 Java，其余和 Java 没有关系。那么二者有没有关系呢？有的。JavaScript 最早设计这门语言的作用是作为衔接后端 Java 组件与前端 Web 组件的“胶水语言”。因为是衔接 Java 组件的脚本语言，所以命名为（Java+Script=）JavaScript。因为前后端传值时会由于诸多因素（如弱网络环境等）导致一些错误（如空值、空指针），所以设计为弱类型。因为时代背景与开发时间的限制（第一版 JavaScript 脚本引擎是 3 天??写出来的），所以采用原型链（prototype chain），未原生支持面向对象特性。综上所述，JavaScript 语言中的面向对象特性本身就是一层 PolyFill。

Static class features

以 Hash tag (#) 标记私有属性、私有方法，示例如下：

```
class ColorFinder {
  static #red = "#ff0000";
  static #green = "#00ff00";
  static #blue = "#0000ff";

  static colorName(name) {
    switch (name) {
      case "red": return ColorFinder.#red;
      case "blue": return ColorFinder.#blue;
      case "green": return ColorFinder.#green;
      default: throw new RangeError("unknown color");
    }
  }
}
```

```
// Somehow use colorName
}
```

Class field declarations for JavaScript

与 Static class features 类似

Private methods and getter/setters for JavaScript classes

支持 getter、setter 方法，示例如下：

```
class Counter extends HTMLElement {
  xValue = 0;

  get x() { return this.xValue; }
  set x(value) {
    this.xValue = value;
    window.requestAnimationFrame(this.render.bind(this));
  }
}
```

ECMAScript class property access expressions

这个提案建议支持以下特性：

在静态方法中访问对象中的非静态属性
一个神奇的特性

□

Angular 脱坑记

之 Lambda 表达式

问题描述

Angular 6.1.5

Lambda 表达式是 Typescript 语法集的一个特性，也称为 Arrow Function。

(...) => { ... }

参数列表 lambda 符号 函数体

解决

如下声明可以接收[无名函数]作为参数：

mission.service.ts

```
export interface MAlert {
  type: string;
  title: string;
  pl: string;
  p2?: string;
  confirmCallback: (res?:any)=>void;
  cancelCallback: (res?:any)=>void;
}
```

Angular 脱坑记

之 NGX 扩展库

问题描述

Angular 6.1.5

NGX 全称 Next Generation eXtension，下一代扩展库。NGX 函数库封装了常用的前端组件库、功能库，是 Angular 一组不可或缺的扩展依赖项。

实际项目中用到以下函数库：

- (1) ngx-echarts Apache 图表库
- (2) ngx-codemirror 前端语法渲染库
- (3) ngx-csv json 转 csv

解决

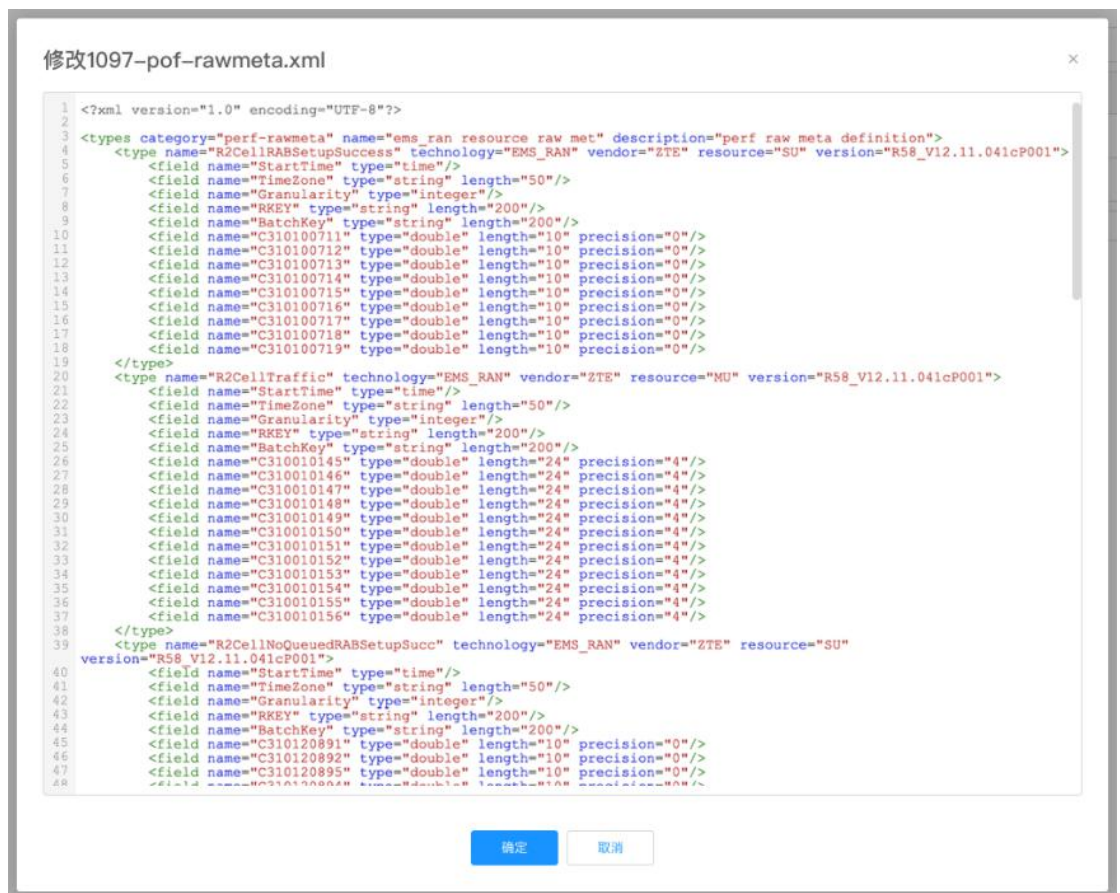
(1) ngx-echarts

效果展示



(2) ngx-codemirror

效果展示



(3) ngx-csv

[threshold-plp.component.ts](#)

```
if (data.isExport()) {  
    console.log("[table_top]opHandler(): EXPORT event triggered.");  
    new NgxCsv(this.data, "csv");  
}
```

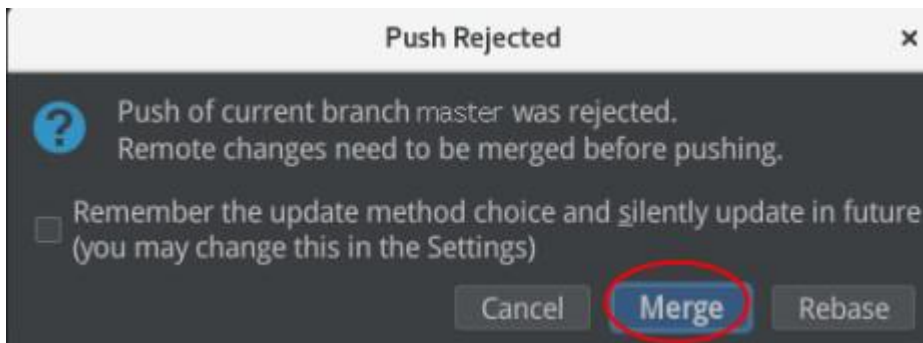
Angular 脱坑记

之修复回滚误操作

问题描述

IntelliJ IDEA	2019.1.3
Git	2.21.0

在团队开发过程中，一处代码可能多人修改；例如当 A 更改代码后提交了，B 在没有实时拉取代码的基础上进行同一文件的修改，然后提交，届时 push 会失败。IDEA 编辑器弹出对话框（如下图所示）：



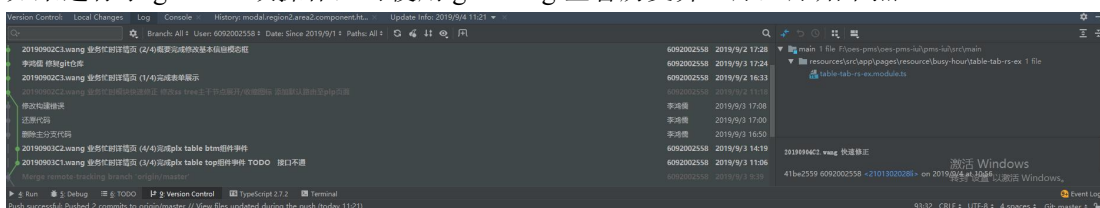
正是在这里，存在埋下“逻辑炸弹”的条件：远程代码库中大量代码段被回滚（删除），IDEA 在执行 Merge 操作时并不会请求确认，而是直接删除本地相应代码段。

解决

由于其不可撤销的特点，Rollback（回滚）是 git 版本控制中的一个危险操作。

最优解法：打开 IDEA Version Control 组件，根据本地仓库与远程仓库差异，过滤出一组本地提交记录。使用 cherry-pick 命令，按照从旧到新的顺序依次提交组内 commit，并 push 到远程仓库

如果进行了 git reset 误操作，可使用 git reflog 查看历史并还原至原始节点（HEAD）



这只是一种临时解法。长期的解决办法：

- （1）对成员账号限制权限
- （2）组内成员建立独立分支，定期（每天、每周、每次发布）合并至 master 分支

中

Angular 脱坑记

之关键字 declare

问题描述

Angular	6.1.5
TypeScript	2.7.2

declare 是 TypeScript 关键字。其作用为声明外部 JavaScript 库中定义的函数，类似于 C 中的函数头。

比如在公式编辑器中需要调用原生 JavaScript 方法选取文本域中的一段文字，如下例示：

解决

item.component.ts

```
declare function selectText(element, startIndex, stopIndex);
```

index.html

```
<script>
...
function selectText(textbox, startIndex, stopIndex) {
  if (textbox.setSelectionRange) {
    textbox.setSelectionRange(startIndex, stopIndex);
  } else if (textbox.createTextRange) {
    var range = textbox.createTextRange();
    range.collapse(true);
    range.moveStart('character', startIndex);
    range.moveEnd('character', stopIndex - startIndex);
    range.select();
  }
  textbox.focus();
}
...
</script>
```

□

Angular 脱坑记

之关键字 keyof

问题描述

Angular	6.1.5
TypeScript	2.7.2

keyof 是 TypeScript 关键字，其作用为返回对象属性名列表（数组），常用于获取、设置对象属性的函数的参数列表中。

解决

jQueryStatic.d.ts

```
proxy<TContext>(context: TContext,
  name: keyof TContext,
  ...additionalArguments: any[]): (...args: any[]) => any;
```

□

Angular 脱坑记

之关键字 `readonly`

问题描述

Angular	6.1.5
TypeScript	2.7.2

`readonly` 为 TypeScript 关键字。其作用为限制变量读取权限为只读，可以看做面向对象模式下 `const`（JavaScript 关键字）的替代品。

解决

`resource.ts`

```
public static readonly RULE_equalityExpression = 4;
```

p.s.可以使用 `Readonly`（Utility Type 的一种）定义常量为 TypeScript 对象类型。下例摘自 TypeScript Docs:

```
interface Todo {
  title: string;
}
const todo: Readonly<Todo> = {
  title: 'Delete inactive users',
};
todo.title = 'Hello'; // Error: cannot reassign a readonly property
```

□

Angular 脱坑记

之内存溢出

问题描述

IntelliJ IDEA	2019.1.3
Angular	6.1.5

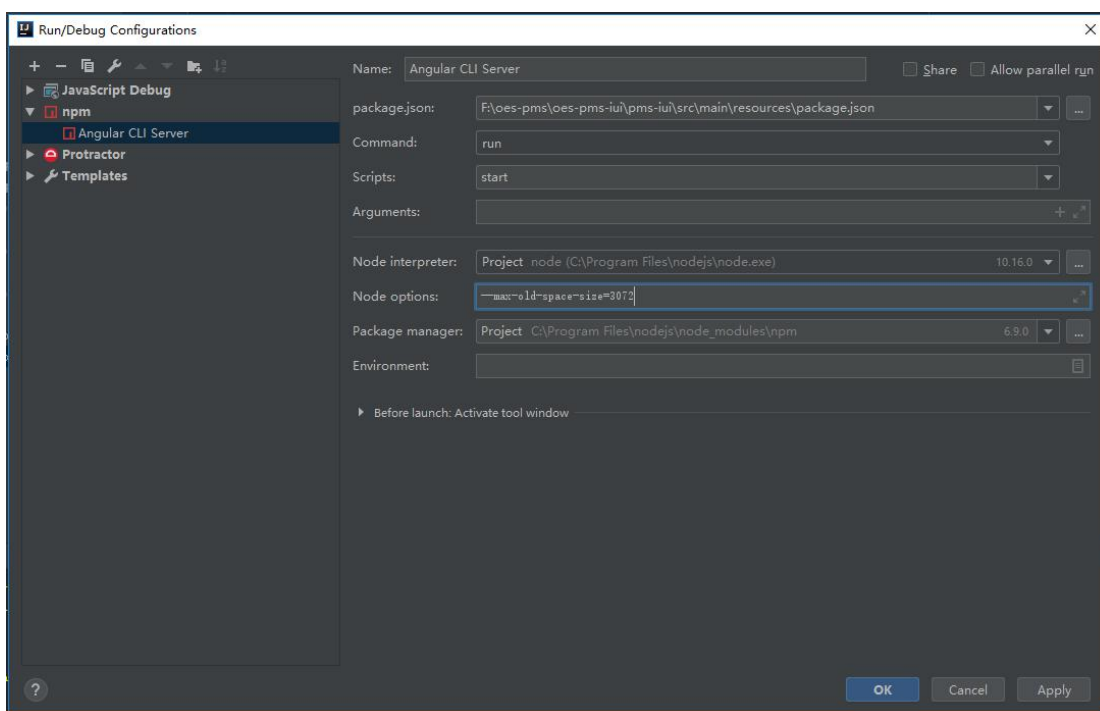
随着页面数的增多，项目文件体积也随之增加。在编译项目的过程中（dev），Node 引擎频繁出现内存溢出问题

```
Exception in thread "main" java.lang.reflect.InvocationTargetException (5 internal calls)
Caused by: java.lang.OutOfMemoryError: PermGen space
    at java.lang.ClassLoader.defineClass(Native Method)
    at java.lang.ClassLoader.defineClass(ClassLoader.java:800)
    at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
    at java.net.URLClassLoader.defineClass(URLClassLoader.java:449)
    at java.net.URLClassLoader.access$100(URLClassLoader.java:71)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:361)
    at java.net.URLClassLoader$1.run(URLClassLoader.java:355) (1 internal call)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:425)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:358)
    at org.infinispan.configuration.cache.UnsafeConfigurationBuilder.<init>(UnsafeConfigurationBuilder.java:29)
    at org.infinispan.configuration.cache.ConfigurationBuilder.<init>(ConfigurationBuilder.java:53)
    at com.tscloud.gisserver.utils.CacheManager.createCacheManagerProgrammatically(CacheManager.java:43)
    at com.tscloud.gisserver.utils.CacheManager.getInstance(CacheManager.java:28)
    at com.tscloud.gisserver.utils.DataGridUtils.getCache(DataGridUtils.java:18)
    at com.tscloud.gisserver.utils.StartJetty.startJetty(StartJetty.java:30)
    at com.tscloud.main.StartGisserver.main(StartGisserver.java:18) (5 internal calls)
    ... 5 more
```

解决

v8 本身有一个默认配置:Currently, by default v8 has a memory limit of 512mb on 32-bit systems, and 1gb on 64-bit systems. The limit can be raised by setting `--max-old-space-size` to a maximum of ~1gb (32-bit) and ~1.7gb (64-bit), but it is recommended that you split your single process into several workers if you are hitting memory limits.

所以我们需要配置一下运行内存



引用

[1] node 内存配, https://blog.csdn.net/qq_20881087/article/details/62428774?utm_source=blogxgwz8

Angular 脱坑记

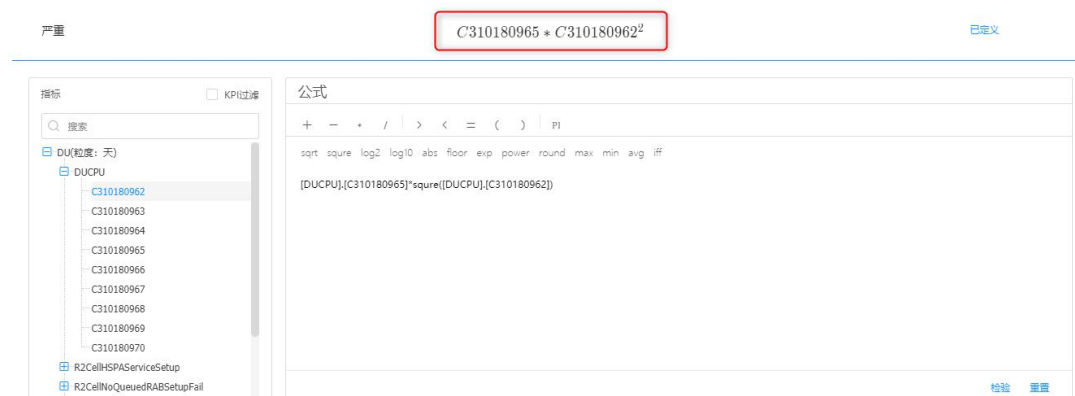
之前端渲染数学表达式

问题描述

Angular 6.1.5

项目中有如下需求：

在公式编辑器中填写 XXX 语法计算式，在公式编辑器标题栏显示经过简化、渲染的 LaTeX 表达式。（如图所示）



XXX 语法是 XX 公司的私有语法集，对变量名的定义如下：

[...] . [...]

指标组名 分隔符 指标名

XXX 语法支持算数运算、逻辑运算、常见数学函数（如指数、对数、幂函数等），复杂度介于简易计算器与 C99 语法之间。

解决

〇、思路

想法是写一个编译器，将 XX 内部计算式编译为 LaTeX 表达式，输入渲染器（不是指 **Renderer**，而是数学表达式渲染器，下同）进行渲染，展示结果。同时，监听文本域（Text Area）事件（Model Change），更新表达式。

一、配置语法解析器 Antlr4

1. 添加依赖

```
npm install --save antlr4ts@0.5.0-alpha.3
npm install --save antlr4ts-cli@0.5.0-alpha.3
```

2. 定义语法 Expr

参考 Antlr4 GitHub 示例 C99 定义语法 Expr 如下：

BasicTypes.g4

```
lexer grammar BasicTypes;
```

```
Constant: Digit+('.'Digit+)?;
Identifier: (Address).'['Prefix(Digit)+']';
WS: [ \r\t\n]+ -> skip;
```

Operators.g4

```
...
Add: '+';
Sub: '-';
Mul: '*';
...
```

Expr.g4

```
...
primaryExpression
: Identifier
| Constant
| '(' expression ')'
| functionDeclaration
;
...
```

3. 添加服务 LaTeX Service

```
...
@Injectable()
export class LatexService {
    convertToLatex(pFormula: string): string {
        ...
    }
}
class CalcExpr implements ExprListener {
    ...
}
```

二、配置公式渲染器 Katex

1. 添加依赖

```
npm install ng-katex --save
```

2. 替换 CDN 服务

参考附录

3. 添加<ng-katex>标签

```
<ng-katex [equation]=",,,"></ng-katex>
```

三、配置 Angular-CLI

1. 添加脚本、样式引用

[index.html](#)

```
<link rel="stylesheet" href="assets/styles/bootstrap.min.css">
<script type="text/javascript" src="assets/scripts/jquery.min.js"></script>
<script type="text/javascript" src="assets/scripts/bootstrap.min.js"></script>
<link rel="stylesheet" href="assets/styles/katex.min.css">
```

2. 注册 angular-cli 命令

package.json

```
"antlr4ts": "node ./node_modules/antlr4ts-cli/antlr4ts
-visitor ./src/app/service/grammar/Expr.g4 -o ./src/app/service/grammar/"
```

3. 更新 angular 配置

(1) 升级 Rx-JS 版本

参考【番外编 记一次排错经历】

(2) 关闭 build optimizer 编译选项

angular.json

```
"buildOptimizer": false,
```

(3) 更新 TS 编译器设置（与 Antlr4ts 兼容）

tsconfig.json

```
"target": "es6",
```

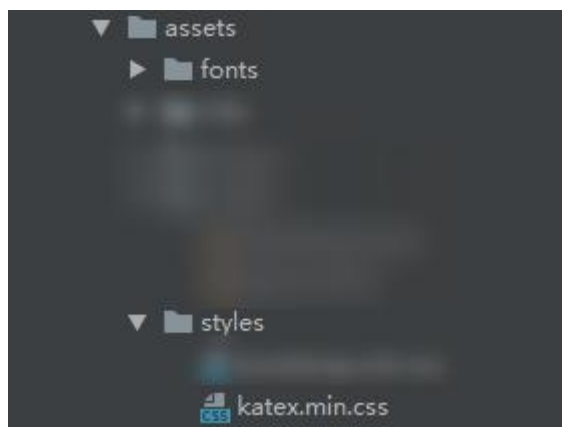
四、测试及效果

如【问题描述】所示

五、展望

随着计算机的更新换代与 H5 技术的发展，浏览器的性能逐步增强。H5 Native API 中定义了 Web Worker 服务。如与本例结合，可实现 C / TS 混编的前端计算，用边缘计算减轻服务器端压力

p.s. 替换 Katex CDN 为本地服务



□

Angular 脱坑记

之动态加载

问题描述

Angular 6.1.5

项目需求：

内部管理后台，页面采用“左树右表”布局。要求根据路由参数动态加载右侧表单组件。同时，保持左侧树结构保持不变

解决

在路由中传递 ID 参数。子页面组件从 URL 获取参数，动态加载对应表单组件。同时，监听路由变化，保证页面初始化完成后，依然能够根据路由变化加载组件

routes.module.ts

```
const routes = [
  {
    path: '',
    component: RsModelComponent
  },
  {
    path: 'extend',
    component: TableTabRsExComponent
  },
  {
    path: 'detail/:type/:resId',
    component: RsModelComponent,
    data: { keep: true, key: "rs-model" }
  }
];
export const routing: ModuleWithProviders = RouterModule.forChild(<Routes>routes);
```

component.html

```
<div class="right-body-inner-div" style="height:100%;">
  <ng-template #viewDetailComponent></ng-template>
</div>
```

component.ts

```
@Component({
  selector: 'demo',
  templateUrl: 'rs-model.component.html',
  styles: [
    `
  `]
})

export class RsModelComponent implements OnInit {
  @ViewChild("treeComponent") treeComponent;
  /*
  * dynamically load components
  * */
  comps: any = [
    RsPlaceholderComponent,
    Area2Component,
    RsDetailRwComponent,
    RsDetailPlpComponent
  ];

  constructor(
    private cfr: ComponentFactoryResolver,
    private router: Router,
    private route: ActivatedRoute
  ) {
    router.events
      .filter((event) => event instanceof NavigationEnd)
      .subscribe((event: NavigationEnd) => {
        // 当路由发生变化，存储在浏览器里面的的用户信息发生变化的时候刷新组件
        this.ngOnInit();
        console.warn("[RsModelComponent] constructor(): router event subscribe.");
      });
  }

  async ngOnInit() {
    let eType = this.route.snapshot.params['type'];
    let eResId = this.route.snapshot.params['resId'];
    console.info("[RsModelComponent] ngOnInit(): type", eType);
    console.info("[RsModelComponent] ngOnInit(): resId", eResId);
    if (isNullOrUndefined(eType)) {
      let comp = this.comps[0];
      let com = this.cfr.resolveComponentFactory(comp);
      this.viewDetailComponent.clear();
      let comRef = this.viewDetailComponent.createComponent(com);
    }
  }
}
```

```

        if (eType == "pdp") {
            ...
        } else if (eType == "plp") {
            /*
             * product list page.
             */
            // dyn load a new plp component HERE.
            let comp = this.comps[3];

            let com = this.cfr.resolveComponentFactory(comp);
            this.viewDetailComponent.clear();
            let comRef = this.viewDetailComponent.createComponent(com);
            this.area2Component = comRef.instance;
        }
    }
}

```

Angular 脱坑记

之同步调用

问题描述

Angular 6.1.5

项目需求：

同步调用因为体验问题，在响应式程序设计中很少采用。但是，在一些特殊的情况下，同步调用方式却可以解决棘手的需求。

比如，一个页面中需要顺序调用若干接口。每个接口的出参，作为下一个接口的入参。最佳解法为采用同步调用方式

解决

resource.ts

```

public postGetExtCategoryList(): Promise<Object> {
    return this.postSimpleAsync(this.getExtCategoryList, {});
}

```

返回参数：Promise 对象（异步请求返回参数为 Observable）

component.ts

```

async caller() {
    await this.resourceService.postGetExtCategoryList()
        .then(res=>{
            ...
        })
}

```



```

    })
    .catch(()=>{
        ...
    });
}

```

await 关键字：加异步锁

async 关键字：声明异步函数

也可以采用 toPromise()方法实现异步调用转同步调用

```

async initPmModelData() {
    await this.pmModelResourceService.getResourceTree().toPromise().then(res => {
        ...
    });
}

```

Angular 脱坑记

之拦截器

问题描述

Angular 6.1.5

HTTP 请求拦截器（HttpInterceptor）是 Angular 6 的一个特性，可用于：

- （1）统一报错拦截（pipelining）
- （2）身份验证
- （3）请求缓存

解决

（4）填充 HttpInterceptor Interface

responseStatusFilter.ts

```

@Injectable()
export class ResponseStatusFilter implements HttpInterceptor {
    constructor(
        private missionService: MissionService
    ) {}

    intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
        if (!environment.enableInterceptors) {
            return next.handle(req);
        }

        if (req.url.includes("/web/rest/dis/") || req.url.includes("/api/vmaster-pms/v1/")) {
            return next.handle(req).do((event: HttpEvent<any>) => {

```

```

        if (event instanceof HttpResponse) {
            if (event.status === 200) {
                // Success.
                console.info("[ResponseStatusFilter] intercept(): url=", req.url);
                console.info("[ResponseStatusFilter] intercept(): method=",
req.method);

                console.info("[ResponseStatusFilter] intercept(): body=", req.body);
            }
        }
    }, (err: any) => {
        if (err instanceof HttpResponse) {
            if (err.status >= 400) {
                this.missionService.alertConfirm({
                    type: "error",
                    message: err.message
                });
            }
        }
    });
}

return next.handle(req);

}
}

```

(5) 在 app module 中挂载拦截器

app.module.ts

```

providers: [
    {
        provide: HTTP_INTERCEPTORS,
        useClass: ResponseStatusFilter,
        multi: true
    }
],

```

(6) 效果



Angular 脱坑记

之数据绑定

问题描述

Angular 6.1.5

数据绑定是 Angular 框架的一大特性，包括：

- (4) [] : 组件外 --> 组件内的单向数据绑定
- (5) () : 组件内 --> 组件外的单向事件绑定
- (6) [](): banana-in-a-plate, 双向数据绑定

解决

(7) 单向数据绑定

reference.html

```
<div class="modal-body">
  <ss-tree [data]="pData"
    (lResources)="addResource($event)"
    [isSelectable]="true"></ss-tree>
</div>
```

ss-tree.component.ts

```
@Component({
  selector: 'ss-tree',
  templateUrl: './tree.html',
  styles: [
```

```

    `]
  })
export class SSTreeComponent {
  @Input() isSelectable: boolean;
}

```

(8) 单向事件绑定

resource-tree.component.ts

```

@Component({
  selector: 'resource-tree',
  templateUrl: './tree.html',
  styles: [

    `]
  })
export class SSTreeComponent implements OnInit {
  @Output() clickEvent = new EventEmitter();
}

```

reference.html

```

<div style="width: 100%;">
  <div class="tree-container" style="height:200px;">
    <resource-tree #plxSelectTree
      (clickEvent)="treeNodeClick($event)"
    >
    </resource-tree>
  </div>
</plx-select>
</div>

```

reference.component.ts

```

export class CustomDropdownComponent {
  treeNodeClick(data) {

  }
}

```

(9) 双向数据绑定 没用过

Angular 脱坑记

之简化项目结构（目录树）

问题描述

Angular	6.1.5
TypeScript	2.7.2

Angular2 采用 MVC（模型、视图、控制器分离）设计模式。因此，理论上应为每个页面建立 model、view、controller 三个文件夹。当页面中需要嵌入子组件时，需要为每个子组件建立 MVC 目录结构。当子组件中又包含子组件时，会使项目结构显得复杂、凌乱。以下总结一些简化项目目录树结构的心得：

解决

1. 将 Model 合并进 View

虽然具有强类型、面向对象的特点，执行 TypeScript 脚本归根结底还是需要依靠 JavaScript。不论具有多复杂、整洁的目录结构，JavaScript 脚本在编译时会被合并为单个文件。因此，【将 Model 合并进 View】在简化项目结构的同时，并不会对 Angular 的设计模式造成副作用。示例如下：

chart.component.ts

```
@Component({
  selector: 'chart',
  templateUrl: 'chart.html',
  styleUrls: ['chart.css']
})
export class ChartComponent implements OnInit {
  mChartModel: ChartModel;
  ...
}
Export class ChartModel {
  var: Type;
}
```

2. 采用 ng-template 实现模态框（Modal）

<ng-template>是 Angular2 项目的一个常用标签。Angular 语法糖*ngIf 就会被模板引擎转换为这个标签：

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

```
<!-- *ngIf 翻译成 ng-template 元素之后 -->
```

```
<ng-template [ngIf]="hero">
  <div class="name">{{hero.name}}</div>
</ng-template>
```

product-list-page.component.html

```
<ng-template #thresholdView let-c="close" let-d="dismiss">
  <div class="modal-header">
    <h4 class="modal-title"></h4>
    <button type="button" class="close" (click)="d('Cross click')">
      <span class="plx-ico-close-l6"></span>
    </button>
  </div>
  <div class="modal-body">
    <div class="div-wrapper">

      </div>
    </div>
  </div>
  <div class="modal-footer" style="margin-top: -16px" *ngIf="!pIsEditMode">
    <div class="form-group w-100">
      <div class="btnGroup modal-btn mx-auto float-none">
        <button type="button" class="plx-btn" (click)="cancel()">关闭</button>
        <button type="button" class="plx-btn plx-btn-primary" (click)="confirm()">修
改</button>
      </div>
    </div>
  </div>
</ng-template>
```

product-list-page.component.ts

```
export class ThresholdPlpComponent implements OnInit {
  modal: any;
  @ViewChild('thresholdView') thresholdView: any;

  constructor(private modalService: PlxModal,
    ...
  ) {
    this.infoRepo = new BasicInfoPlpRepository();
    this.onChange = new EventEmitter<ResourceTypeRepository>();
    this.data = [];
  }

  openModal() {
    const size: 'sm' | 'lg' = 'lg';
    const options = {
      size: size,
    };
    this.modal = this.modalService.open(this.thresholdView, options);
  }
}
```

3. 使用 TypeScript 高级语言特性

Intersection Types <#>

An intersection type combines multiple types into one. This allows you to add together existing types to get a single type that has all the features you need. For example, `Person & Serializable & Loggable` is a `Person` and `Serializable` and `Loggable`. That means an object of this type will have all members of all three types.

[product-list-page.component.ts](#)

```
srcObj = {} as BasicInfo&DateAndTime&FormulaSet&SelectedPo&SelectedResInstance;
```

□

Angular 脱坑记 番外编

之记一次排错经历

问题描述

angular	6.1.5
rx-js	6.1.0
rxjs-compat	6.1.0

rx-js 全称 the React eXtension for JavaScript。一个重要的作用是提供异步调用接口，其定义了对象 `Observable`、`Subscription` 等。

故障现象：

- 测试环境编译、服务正常
- 生产环境编译通过、全称无报错
- 生产环境服务，访问页面时控制台报错 “XXX: Subscription is not defined.”

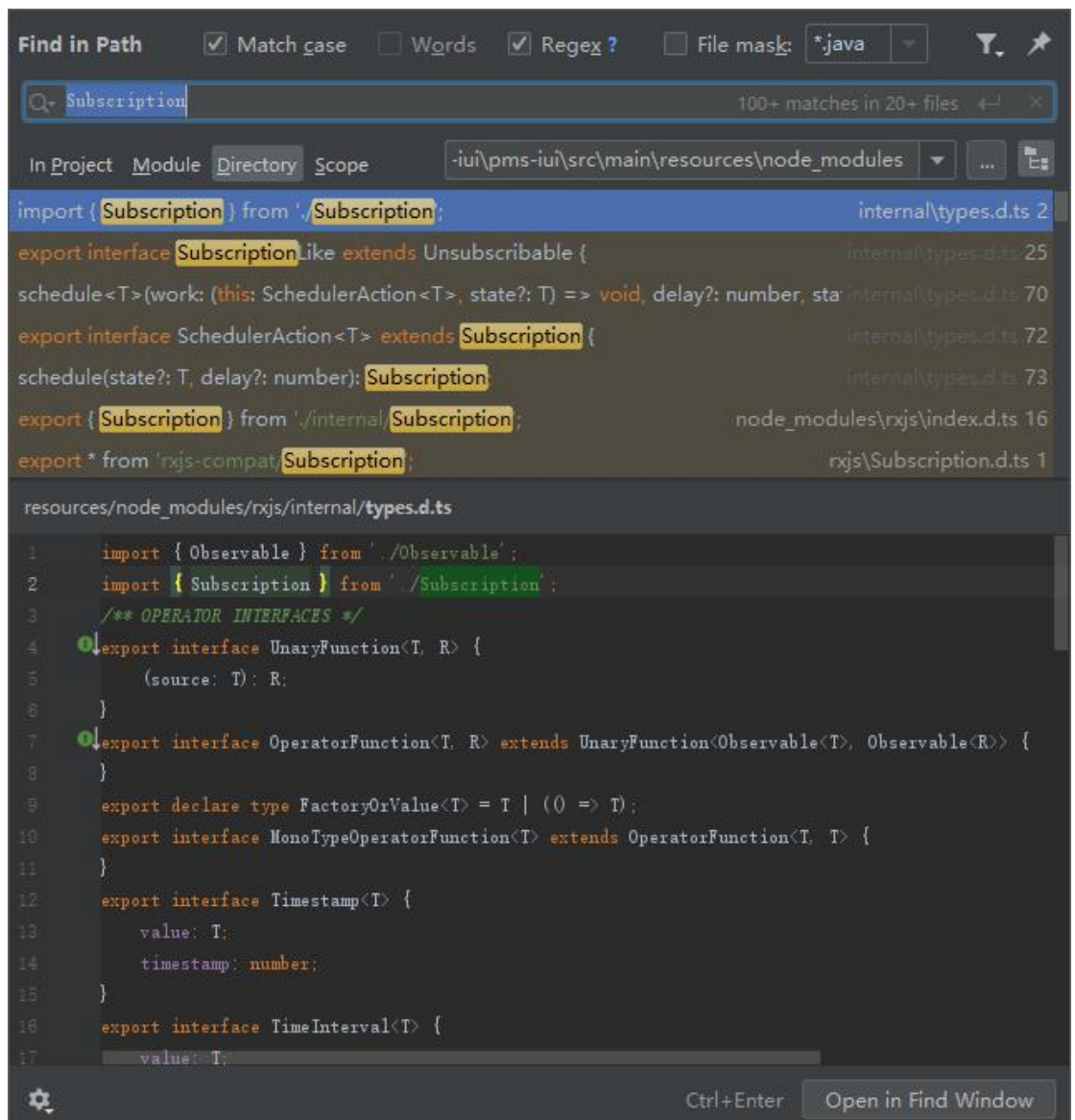
解决

(10) 验证故障现象

(11) 检查页面

在故障页面右键选“检查”，观察到页面渲染至 `<app-root></app-root>`，故推测为依赖项相关错误。

(12) IDEA 编辑器全局搜索 `Subscription` 对象，结果如下：



以上结果说明：

故障是依赖项 rx-js 导致的。

(13) 检查 package.json

package.json

```
"rxjs": "~6.1.0",
"rxjs-compat": "~6.1.0",
```

与 npm 版本比较，可知其版本号低于最新版本。将两个依赖项升级至 6.3.0 即可。

(14) 关闭代码压缩选项

关闭 optimization 选项，在浏览器中可见原始 JS 代码（未混淆、压缩）


```
package.json x angular.json x rs-model.module.ts x styles.css x
5 "node_modules/jquery/dist/jquery.js",
6 "node_modules/zone.js/dist/zone.js",
7 "node_modules/tools.js/tools.js",
8 "node_modules/echarts/dist/echarts.js"
9 ]
10 },
11 "configurations": {
12   "production": {
13     "optimization": true,
14     "outputHashing": "all",
15     "sourceMap": false,
```

(15) 进一步调试

进一步调试发现,angular-cli 的@angular-devkit/build-optimizer 与现有依赖库函数之间存在未知的副作用,故将其关闭:

```
package.json x angular.json x rs-model.module.ts x styles.css
35 "node_modules/jquery/dist/jquery.js",
36 "node_modules/zone.js/dist/zone.js",
37 "node_modules/tools.js/tools.js",
38 "node_modules/echarts/dist/echarts.js"
39 ]
40 },
41 "configurations": {
42   "production": {
43     "optimization": true,
44     "outputHashing": "all",
45     "sourceMap": false,
46     "extractCss": true,
47     "namedChunks": false,
48     "aot": true,
49     "extractLicenses": true,
50     "vendorChunk": false,
51     "buildOptimizer": false,
52     "fileReplacements": [
53       {
```

(16) 验证

部署生产环境 Nginx 服务,经测试,故障已解决。

附录 build-optimizer 扩展选项:

```
export interface BuildOptimizerOptions {  
  content?: string;  
  inputFilePath?: string;  
  outputFilePath?: string;  
  emitSourceMap?: boolean;  
  strict?: boolean;  
  isSideEffectFree?: boolean;  
}
```

【小结】

1. build-optimizer 可能存在副作用 (Side Effect)
2. 关闭 optimization 功能有利于在生产环境调试依赖项
3. 如无特殊需要, 尽量不要使用不成熟的新技术

□

Angular 脱坑记

之路由调试

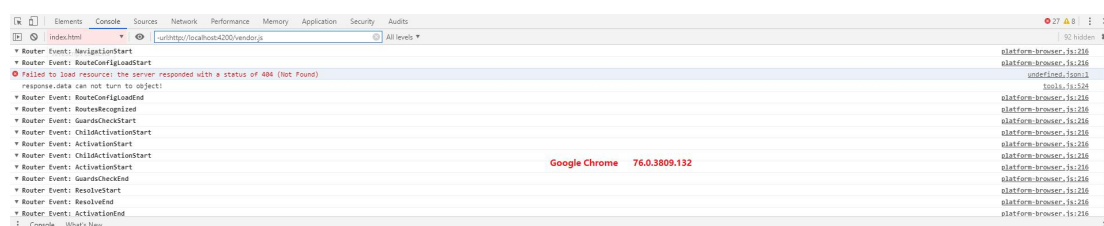
问题描述

Google Chrome 76.0.3809.132

Angular 6.1.5

enableTracing=true

不能正常打印路由调试信息



解决

使用 Edge 浏览器可正常打印路由调试日志。打开调试器会出现页面频繁刷新的现象(可能是内存溢出引起的)



Angular 脱坑记

之路由复用

问题描述

Angular 6.1.5

(7) 在基于 Angular 的 SPA 应用中，应用通过路由在各个页面之间进行导航。默认情况下，用户在离开一个页面时，这个页面(组件)会被 Angular 销毁，用户的输入信息也随之丢失，当用户再次进入这个页面时，看到的是一个新生成的页面(组件)，之前的输入信息都没了。

(8) 配置的前端项目就是基于 Angular 的，工作中遇到了这样的问题，部分页面需要保存用户的输入信息，用户再次进入页面时需要回到上一次离开时的状态，部分页面每次都要刷新页面，不需要保存用户信息。而页面间的导航正是通过路由实现的，Angular 的默认行为不能满足我们的需求！

解决

针对以上问题，通过查阅 Angular 的相关资料可以发现，Angular 提供了 RouteReuseStrategy 接口，通过实现这个接口，可以让开发者自定义路由复用策略。

(1) RouteReuseStrategy 接口

```
export abstract class RouteReuseStrategy {
  abstract shouldDetach(route: ActivatedRouteSnapshot): boolean;

  abstract store(route: ActivatedRouteSnapshot, handle:
  DetachedRouteHandle | null): void;

  abstract shouldAttach(route: ActivatedRouteSnapshot): boolean;

  abstract retrieve(route: ActivatedRouteSnapshot): DetachedRouteHandle |
  null;

  abstract shouldReuseRoute(future: ActivatedRouteSnapshot, curr:
  ActivatedRouteSnapshot): boolean;
}
```

这个接口只定义了 5 个方法，每个方法的作用如下：

① shouldDetach

路由离开时是否需要保存页面，这是实现自定义路由复用策略最重要的一个方法。

其中：

返回值为 true 时，路由离开时保存页面信息，当路由再次激活时，会直接显示保存的页面。

返回值为 false 时，路由离开时直接销毁组件，当路由再次激活时，直接初始化为新页面。

② store

如果 shouldDetach 方法返回 true，会调用这个方法保存页面。

③ shouldAttach

路由进入页面时是否有页面可以重用。 true： 重用页面，false： 生成新的页面

④ retrieve

路由激活时获取保存的页面，如果返回 null，则生成新页面

⑤ shouldReuseRoute

决定跳转后是否可以使用跳转前的路由页面，即跳转前后跳转后使用相同的页面

(2) 实践

① 自定义路由复用策略

```
export class CustomRouteReuseStrategy implements RouteReuseStrategy {
  handlers: { [key: string]: DetachedRouteHandle } = {};

  shouldDetach(route: ActivatedRouteSnapshot): boolean {
    return route.data.reload === false;
  }

  store(route: ActivatedRouteSnapshot, handle: DetachedRouteHandle): void {
    this.handlers[route.routeConfig.path] = handle;
  }

  shouldAttach(route: ActivatedRouteSnapshot): boolean {
    return !!route.routeConfig
    && !!this.handlers[route.routeConfig.path];
  }

  retrieve(route: ActivatedRouteSnapshot): DetachedRouteHandle {
    return this.handlers[route.routeConfig.path];
  }

  shouldReuseRoute(future: ActivatedRouteSnapshot, curr:
  ActivatedRouteSnapshot): boolean {
    return future.routeConfig === curr.routeConfig;
  }
}
```

在这个路由复用策略中，有两个关键点：

1. 我们使用了一个 handlers 对象来保存页面。

2. 通过路由配置的 `data` 对象中的 `reload` 属性来判断一个页面是否需要保存，并且只有 `reload` 属性为 `false` 时，才会保存页面。如果不配置 `reload` 属性，或者 `reload` 属性不为 `false`，则不会保存页面。

② 配置路由重用策略为自定义策略

为了使用自定义的路由复用策略，需要在应用的根路由模块 `providers` 中使用自定义的路由复用策略

```
@NgModule({
  imports: [RouterModule.forRoot(routes, {useHash: true})],
  exports: [RouterModule],
  providers: [
    {
      provide: RouteReuseStrategy,
      useClass: CustomRouteReuseStrategy
    }
  ]
})
export class AppRoutingModule { }
```

③ 配置路由

在路由配置中，按需配置路由的 `data` 属性。如需要保存页面，则设置 `reload` 值为 `false`，如不需要保存页面，不配置该属性。例如：

```
const routes: Routes = [
  {
    path: 'foo',
    component: FooComponent
  },
  {
    path: 'bar',
    component: BarComponent,
    data: {reload: false}
  }
];
```

此路由配置下，访问/`foo` 页面始终会生成一个新的页面，而/`bar` 页面会在路由离开时会被保存，再次进入该页面都会恢复到上一次离开该页面时的状态

(3) 扩展

可以使用 Angular 路由复用策略实现 Tab（选项卡）功能（读写 `cookie`）

引用

[1] 中兴开发者社区，<https://blog.csdn.net/o4dc8ojo7zl6/article/details/79224523>

[2] angular 4 实现的 tab 栏切换，<https://www.cnblogs.com/lslgg/p/7700888.html>

Angular2 技术笔记

Angular 脱坑记

之 Lambda 表达式

问题描述

Angular 6.1.5

Lambda 表达式是 Typescript 语法集的一个特性，也称为 Arrow Function。

(...) => { ... }

参数列表 lambda 符号 函数体

解决

如下声明可以接收[无名函数]作为参数：

mission.service.ts

```
export interface MAlert {  
  type: string;  
  title: string;  
  pl: string;  
  p2?: string;  
  confirmCallback: (res?:any)=>void;  
  cancelCallback: (res?:any)=>void;  
}
```

Angular 脱坑记

之 NGX 扩展库

问题描述

Angular 6.1.5

NGX 全称 Next Generation eXtension，下一代扩展库。NGX 函数库封装了常用的前端组件库、功能库，是 Angular 一组不可或缺的扩展依赖项。

实际项目中用到以下函数库：

- (4) ngx-echarts Apache 图表库
- (5) ngx-codemirror 前端语法渲染库
- (6) ngx-csv json 转 csv

解决

(17) ngx-echarts

效果展示



(18) ngx-codemirror

效果展示

```
修改1097-pof-rawmeta.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <types category="perf-rawmeta" name="ems_ran resource raw met" description="perf raw meta definition">
4   <type name="R2CellRABSetupSuccess" technology="EMS_RAN" vendor="ZTE" resource="SU" version="R58_V12.11.041cP001">
5     <field name="StartTime" type="time"/>
6     <field name="TimeZone" type="string" length="50"/>
7     <field name="Granularity" type="integer"/>
8     <field name="RKEY" type="string" length="200"/>
9     <field name="BatchKey" type="string" length="200"/>
10    <field name="C310100711" type="double" length="10" precision="0"/>
11    <field name="C310100712" type="double" length="10" precision="0"/>
12    <field name="C310100713" type="double" length="10" precision="0"/>
13    <field name="C310100714" type="double" length="10" precision="0"/>
14    <field name="C310100715" type="double" length="10" precision="0"/>
15    <field name="C310100716" type="double" length="10" precision="0"/>
16    <field name="C310100717" type="double" length="10" precision="0"/>
17    <field name="C310100718" type="double" length="10" precision="0"/>
18    <field name="C310100719" type="double" length="10" precision="0"/>
19  </type>
20  <type name="R2CellTraffic" technology="EMS_RAN" vendor="ZTE" resource="MU" version="R58_V12.11.041cP001">
21    <field name="StartTime" type="time"/>
22    <field name="TimeZone" type="string" length="50"/>
23    <field name="Granularity" type="integer"/>
24    <field name="RKEY" type="string" length="200"/>
25    <field name="BatchKey" type="string" length="200"/>
26    <field name="C310010145" type="double" length="24" precision="4"/>
27    <field name="C310010146" type="double" length="24" precision="4"/>
28    <field name="C310010147" type="double" length="24" precision="4"/>
29    <field name="C310010148" type="double" length="24" precision="4"/>
30    <field name="C310010149" type="double" length="24" precision="4"/>
31    <field name="C310010150" type="double" length="24" precision="4"/>
32    <field name="C310010151" type="double" length="24" precision="4"/>
33    <field name="C310010152" type="double" length="24" precision="4"/>
34    <field name="C310010153" type="double" length="24" precision="4"/>
35    <field name="C310010154" type="double" length="24" precision="4"/>
36    <field name="C310010155" type="double" length="24" precision="4"/>
37    <field name="C310010156" type="double" length="24" precision="4"/>
38  </type>
39  <type name="R2CellNoQueuedRABSetupSucc" technology="EMS_RAN" vendor="ZTE" resource="SU"
40  version="R58_V12.11.041cP001">
41    <field name="StartTime" type="time"/>
42    <field name="TimeZone" type="string" length="50"/>
43    <field name="Granularity" type="integer"/>
44    <field name="RKEY" type="string" length="200"/>
45    <field name="BatchKey" type="string" length="200"/>
46    <field name="C310120891" type="double" length="10" precision="0"/>
47    <field name="C310120892" type="double" length="10" precision="0"/>
48    <field name="C310120895" type="double" length="10" precision="0"/>
49  </type>
50  <type name="R2CellNoQueuedRABSetupSucc" technology="EMS_RAN" vendor="ZTE" resource="SU"
51  version="R58_V12.11.041cP001">
52    <field name="StartTime" type="time"/>
53    <field name="TimeZone" type="string" length="50"/>
54    <field name="Granularity" type="integer"/>
55    <field name="RKEY" type="string" length="200"/>
56    <field name="BatchKey" type="string" length="200"/>
57    <field name="C310120891" type="double" length="10" precision="0"/>
58    <field name="C310120892" type="double" length="10" precision="0"/>
59    <field name="C310120895" type="double" length="10" precision="0"/>
60  </type>
61  <type name="R2CellNoQueuedRABSetupSucc" technology="EMS_RAN" vendor="ZTE" resource="SU"
62  version="R58_V12.11.041cP001">
63    <field name="StartTime" type="time"/>
64    <field name="TimeZone" type="string" length="50"/>
65    <field name="Granularity" type="integer"/>
66    <field name="RKEY" type="string" length="200"/>
67    <field name="BatchKey" type="string" length="200"/>
68    <field name="C310120891" type="double" length="10" precision="0"/>
69    <field name="C310120892" type="double" length="10" precision="0"/>
70    <field name="C310120895" type="double" length="10" precision="0"/>
71  </type>
72  <type name="R2CellNoQueuedRABSetupSucc" technology="EMS_RAN" vendor="ZTE" resource="SU"
73  version="R58_V12.11.041cP001">
74    <field name="StartTime" type="time"/>
75    <field name="TimeZone" type="string" length="50"/>
76    <field name="Granularity" type="integer"/>
77    <field name="RKEY" type="string" length="200"/>
78    <field name="BatchKey" type="string" length="200"/>
79    <field name="C310120891" type="double" length="10" precision="0"/>
80    <field name="C310120892" type="double" length="10" precision="0"/>
81    <field name="C310120895" type="double" length="10" precision="0"/>
82  </type>
83  <type name="R2CellNoQueuedRABSetupSucc" technology="EMS_RAN" vendor="ZTE" resource="SU"
84  version="R58_V12.11.041cP001">
85    <field name="StartTime" type="time"/>
86    <field name="TimeZone" type="string" length="50"/>
87    <field name="Granularity" type="integer"/>
88    <field name="RKEY" type="string" length="200"/>
89    <field name="BatchKey" type="string" length="200"/>
90    <field name="C310120891" type="double" length="10" precision="0"/>
91    <field name="C310120892" type="double" length="10" precision="0"/>
92    <field name="C310120895" type="double" length="10" precision="0"/>
93  </type>
94  <type name="R2CellNoQueuedRABSetupSucc" technology="EMS_RAN" vendor="ZTE" resource="SU"
95  version="R58_V12.11.041cP001">
96    <field name="StartTime" type="time"/>
97    <field name="TimeZone" type="string" length="50"/>
98    <field name="Granularity" type="integer"/>
99    <field name="RKEY" type="string" length="200"/>
100   <field name="BatchKey" type="string" length="200"/>
101   <field name="C310120891" type="double" length="10" precision="0"/>
102   <field name="C310120892" type="double" length="10" precision="0"/>
103   <field name="C310120895" type="double" length="10" precision="0"/>
104 </type>
105 </types>
106 </xml>
```

(19) ngx-csv

threshold-plp.component.ts

```
if (data.isExport()) {
  console.log("[table_top]opHandler(): EXPORT event triggered.");
  new ngxCsv(this.data, "csv");
}
```

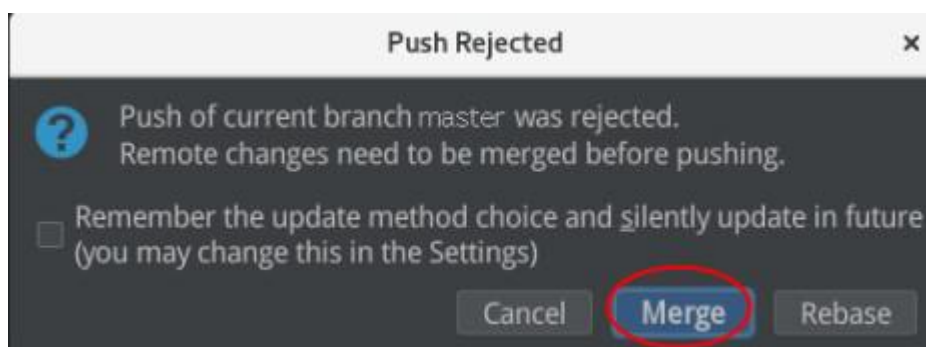
Angular 脱坑记

之修复回滚误操作

问题描述

IntelliJ IDEA 2019.1.3
Git 2.21.0

在团队开发过程中，一处代码可能多人修改；例如当 A 更改代码后提交了，B 在没有实时拉取代码的基础上进行同一文件的修改，然后提交，届时 push 会失败。IDEA 编辑器弹出对话框（如下图所示）：



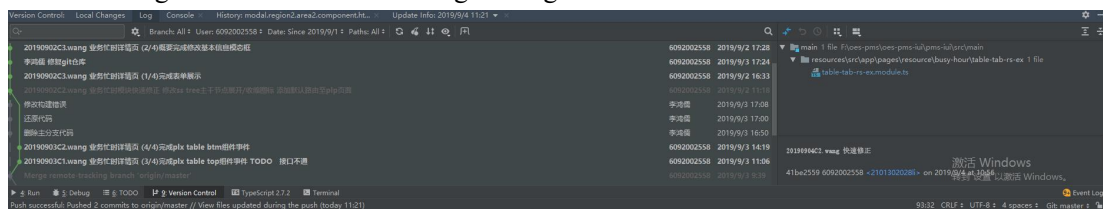
正是在这里，存在埋下“逻辑炸弹”的条件：远程代码库中大量代码段被回滚（删除），IDEA 在执行 Merge 操作时并不会请求确认，而是直接删除本地相应代码段。

解决

由于其不可撤销的特点，Rollback（回滚）是 git 版本控制中的一个危险操作。

最优解法：打开 IDEA Version Control 组件，根据本地仓库与远程仓库差异，过滤出一组本地提交记录。使用 cherry-pick 命令，按照从旧到新的顺序依次提交组内 commit，并 push 到远程仓库

如果进行了 git reset 误操作，可使用 git relog 查看历史并还原至原始节点（HEAD）



这只是一种临时解法。长期的解决办法：

（3）对成员账号限制权限

（4）组内成员建立独立分支，定期（每天、每周、每次发布）合并至 master 分支

中

Angular 脱坑记

之关键字 declare

问题描述

Angular	6.1.5
TypeScript	2.7.2

declare 是 TypeScript 关键字。其作用为声明外部 JavaScript 库中定义的函数，类似于 C 中的函数头。

比如在公式编辑器中需要调用原生 JavaScript 方法选取文本域中的一段文字，如下例示：

解决

item.component.ts

```
declare function selectText(element, startIndex, stopIndex);
```

index.html

```
<script>
  ...
  function selectText(textbox, startIndex, stopIndex) {
    if (textbox.setSelectionRange) {
      textbox.setSelectionRange(startIndex, stopIndex);
    } else if (textbox.createTextRange) {
      var range = textbox.createTextRange();
      range.collapse(true);
      range.moveStart('character', startIndex);
      range.moveEnd('character', stopIndex - startIndex);
      range.select();
    }
    textbox.focus();
  }
  ...
</script>
```

□

Angular 脱坑记

之关键字 keyof

问题描述

Angular	6.1.5
TypeScript	2.7.2

keyof 是 TypeScript 关键字，其作用为返回对象属性名列表（数组），常用于获取、设置对象属性的函数的参数列表中。

解决

jQueryStatic.d.ts

```
proxy<TContext>(context: TContext,  
    name: keyof TContext,  
    ...additionalArguments: any[]): (...args: any[]) => any;
```

□

Angular 脱坑记

之关键字 readonly

问题描述

Angular	6.1.5
TypeScript	2.7.2

readonly 为 TypeScript 关键字。其作用为限制变量读取权限为只读，可以看做面向对象模式下 const（JavaScript 关键字）的替代品。

解决

resource.ts

```
public static readonly RULE_equalityExpression = 4;
```

p.s.可以使用 Readonly（Utility Type 的一种）定义常量为 TypeScript 对象类型。下例摘自 TypeScript Docs:

```
interface Todo {  
    title: string;
```

```

}
const todo: Readonly<Todo> = {
  title: 'Delete inactive users',
};
todo.title = 'Hello'; // Error: cannot reassign a readonly
property

```

□

Angular 脱坑记

之内存溢出

问题描述

IntelliJ IDEA	2019.1.3
Angular	6.1.5

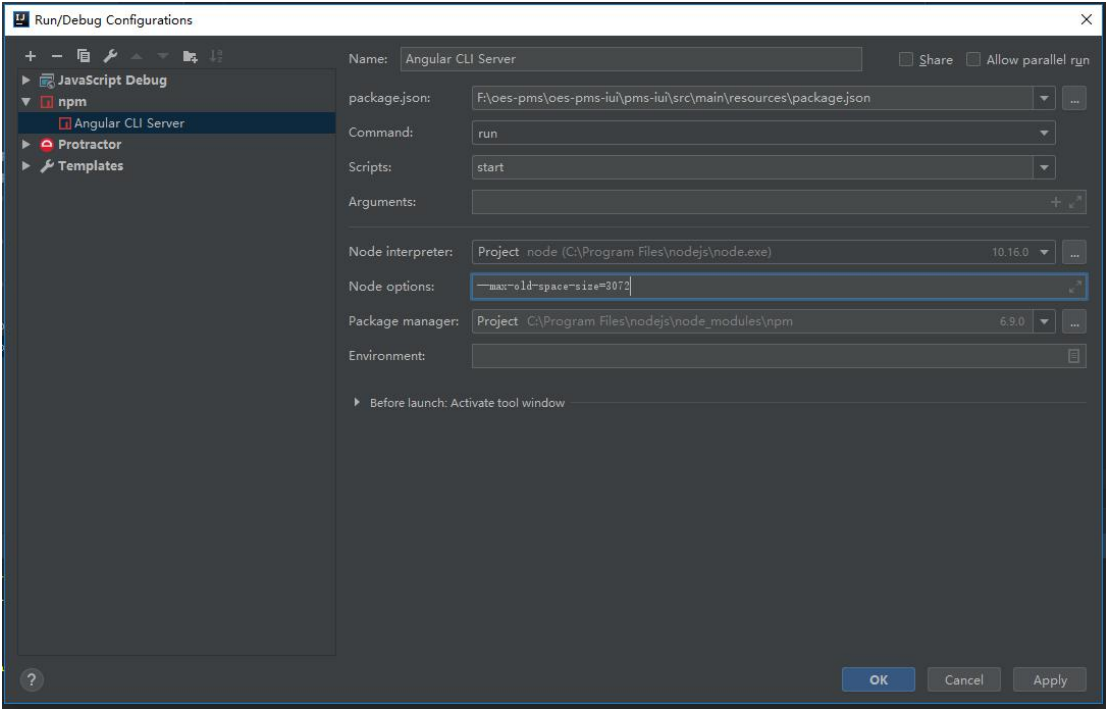
随着页面数的增多，项目文件体积也随之增加。在编译项目的过程中（dev），Node 引擎频繁出现内存溢出问题



解决

v8 本身有一个默认配置:Currently, by default v8 has a memory limit of 512mb on 32-bit systems, and 1gb on 64-bit systems. The limit can be raised by setting `--max-old-space-size` to a maximum of ~1gb (32-bit) and ~1.7gb (64-bit), but it is recommended that you split your single process into several workers if you are hitting memory limits.

所以我们需要配置一下运行内存



引用

[1] node 内存配, https://blog.csdn.net/qq_20881087/article/details/62428774?utm_source=blogxgwz8

Angular 脱坑记

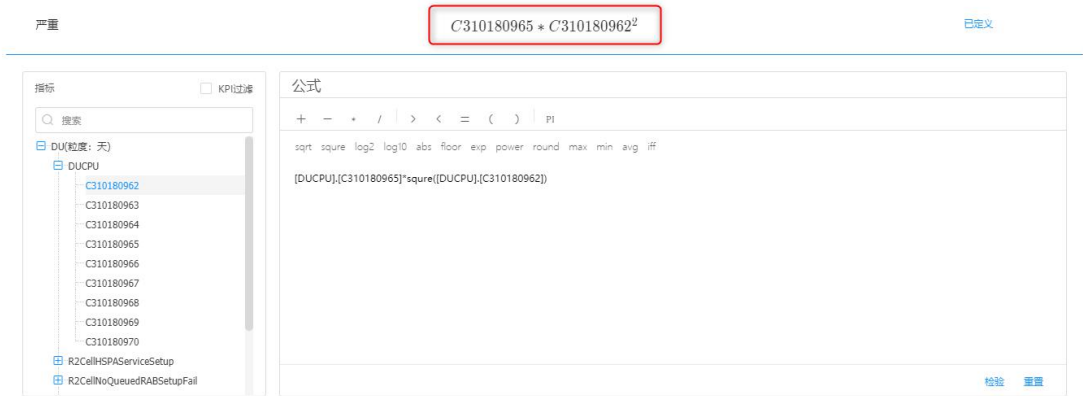
之前端渲染数学表达式

问题描述

Angular 6.1.5

项目中有如下需求：

在公式编辑器中填写 XXX 语法计算式，在公式编辑器标题栏显示经过简化、渲染的 LaTeX 表达式。（如图所示）



XXX 语法是 XX 公司的私有语法集，对变量名的定义如下：

[...] . [...]

.....

指标组名	分隔符	指标名
------	-----	-----

XXX 语法支持算数运算、逻辑运算、常见数学函数（如指数、对数、幂函数等），复杂度介于简易计算器与 C99 语法之间。

解决

六、思路

想法是写一个编译器，将 XX 内部计算式编译为 LaTeX 表达式，输入渲染器（不是指 `Renderer`，而是数学表达式渲染器，下同）进行渲染，展示结果。同时，监听文本域（`Text Area`）事件（`Model Change`），更新表达式。

七、配置语法解析器 Antlr4

4. 添加依赖

```
npm install --save antlr4ts@0.5.0-alpha.3
npm install --save antlr4ts-cli@0.5.0-alpha.3
```

5. 定义语法 Expr

参考 Antlr4 GitHub 示例 C99 定义语法 Expr 如下：

BasicTypes.g4

```
lexer grammar BasicTypes;
...
Constant: Digit+('.'Digit+)?;
Identifier: (Address) '[' Prefix(Digit)+ ']' ;
WS: [ \r\t\n]+ -> skip;
```

Operators.g4

```
...
Add: '+' ;
Sub: '-' ;
Mul: '*' ;
...
```

Expr.g4

```
...
primaryExpression
: Identifier
| Constant
| '(' expression ')'
| functionDeclaration
;
...
```

6. 添加服务 LaTeX Service

```
...
@Injectable()
export class LatexService {
    convertToLatex(pFormula: string): string {
        ...
    }
}
class CalcExpr implements ExprListener {
    ...
}
```

八、配置公式渲染器 Katex

4. 添加依赖

```
npm install ng-katex --save
```

5. 替换 CDN 服务

参考附录

6. 添加<ng-katex>标签

```
<ng-katex [equation]=",,,"></ng-katex>
```

九、配置 Angular-CLI

4. 添加脚本、样式引用

index.html

```
<link rel="stylesheet" href="assets/styles/bootstrap.min.css">
<script type="text/javascript" src="assets/scripts/jquery.min.js"></script>
<script type="text/javascript" src="assets/scripts/bootstrap.min.js"></script>
<link rel="stylesheet" href="assets/styles/katex.min.css">
```

5. 注册 angular-cli 命令

package.json

```
"antlr4ts": "node ./node_modules/antlr4ts-cli/antlr4ts
-visitor ./src/app/service/grammar/Expr.g4 -o ./src/app/service/grammar/"
```

6. 更新 angular 配置

(1) 升级 Rx-JS 版本

参考【番外编 记一次排错经历】

(2) 关闭 build optimizer 编译选项

angular.json

```
"buildOptimizer": false,
```

(3) 更新 TS 编译器设置（与 Antlr4ts 兼容）

tsconfig.json

```
"target": "es6",
```

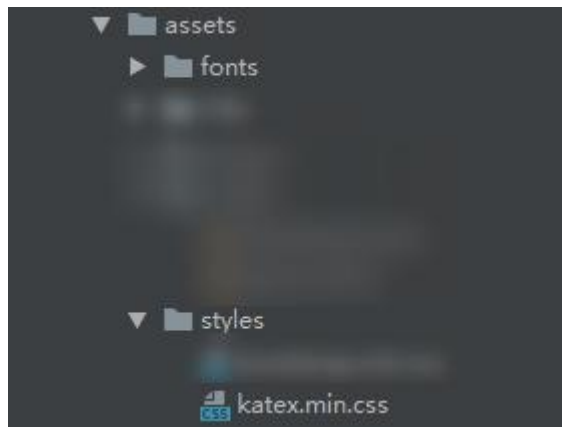
十、测试及效果

如【问题描述】所示

十一、展望

随着计算机的更新换代与 H5 技术的发展，浏览器的性能逐步增强。H5 Native API 中定义了 Web Worker 服务。如与本例结合，可实现 C / TS 混编的前端计算，用边缘计算减轻服务器端压力

p.s. 替换 Katex CDN 为本地服务



□

Angular 脱坑记

之动态加载

问题描述

Angular 6.1.5

项目需求：

内部管理后台，页面采用“左树右表”布局。要求根据路由参数动态加载右侧表单组件。同时，保持左侧树结构保持不变

解决

在路由中传递 ID 参数。子页面组件从 URL 获取参数，动态加载对应表单组件。同时，监听路由变化，保证页面初始化完成后，依然能够根据路由变化加载组件

routes.module.ts

```
const routes = [
  {
    path: '',
    component: RsModelComponent
  },
  {
    path: 'extend',
```

```

        component: TableTabRsExComponent
    },
    {
        path: 'detail/:type/:resId',
        component: RsModelComponent,
        data: { keep: true, key: "rs-model" }
    }
];
export const routing: ModuleWithProviders = RouterModule.forChild(<Routes>routes);

```

component.html

```

<div class="right-body-inner-div" style="height:100%;">
    <ng-template #viewDetailComponent></ng-template>
</div>

```

component.ts

```

@Component({
    selector: 'demo',
    templateUrl: 'rs-model.component.html',
    styles: [
        `
    `]
})
export class RsModelComponent implements OnInit {
    @ViewChild("treeComponent") treeComponent;
    /*
    * dynamically load components
    * */
    comps: any = [
        RsPlaceholderComponent,
        Area2Component,
        RsDetailRwComponent,
        RsDetailPlpComponent
    ];
    constructor(
        private cfr: ComponentFactoryResolver,
        private router: Router,
        private route: ActivatedRoute
    ) {

```



```

router.events
  .filter((event) => event instanceof NavigationEnd)
  .subscribe((event: NavigationEnd) => {
    // 当路由发生变化，存储在浏览器里面的的用户信息发生变化的时候刷新组件
    this.ngOnInit();
    console.warn("[RsModelComponent] constructor(): router event subscribe.");
  });
}

async ngOnInit() {
  let eType = this.route.snapshot.params['type'];
  let eResId = this.route.snapshot.params['resId'];
  console.info("[RsModelComponent] ngOnInit(): type", eType);
  console.info("[RsModelComponent] ngOnInit(): resId", eResId);
  if (isNullOrUndefined(eType)) {
    let comp = this.comps[0];
    let com = this.cfr.resolveComponentFactory(comp);
    this.viewDetailComponent.clear();
    let comRef = this.viewDetailComponent.createComponent(com);
  }
  if (eType == "pdp") {
    ...
  } else if (eType == "plp") {
    /*
    * product list page.
    * */
    // dyn load a new plp component HERE.
    let comp = this.comps[3];

    let com = this.cfr.resolveComponentFactory(comp);
    this.viewDetailComponent.clear();
    let comRef = this.viewDetailComponent.createComponent(com);
    this.area2Component = comRef.instance;
  }
}
}

```

Angular 脱坑记

之同步调用

问题描述

Angular 6.1.5

项目需求：

同步调用因为体验问题，在响应式程序设计中很少采用。但是，在一些特殊的情况下，同步调用方式却可以解决棘手的需求。

比如，一个页面中需要顺序调用若干接口。每个接口的出参，作为下一个接口的入参。最佳解法为采用同步调用方式

解决

resource.ts

```
public postGetExtCategoryList(): Promise<Object> {  
    return this.postSimpleAsync(this.getExtCategoryList, {});  
}
```

返回参数：Promise 对象（异步请求返回参数为 Observable）

component.ts

```
async caller() {  
    await this.resourceService.postGetExtCategoryList()  
        .then(res=>{  
        ...  
    })  
    .catch(()=>{  
        ...  
    });  
}
```

await 关键字：加异步锁

async 关键字：声明异步函数

也可以采用 toPromise()方法实现异步调用转同步调用

```
async initPmModelData() {  
    await this.pmModelResourceService.getResourceTree().toPromise().then(res => {  
        ...  
    });  
}
```

Angular 脱坑记

之拦截器

问题描述

Angular 6.1.5

HTTP 请求拦截器（HttpInterceptor）是 Angular 6 的一个特性，可用于：

（9）统一报错拦截（pipelining）

(10) 身份验证

(11) 请求缓存

解决

(20) 填充 HttpInterceptor Interface

responseStatusFilter.ts

```
@Injectable()
export class ResponseStatusFilter implements HttpInterceptor {
  constructor(
    private missionService: MissionService
  ) {}

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    if (!environment.enableInterceptors) {
      return next.handle(req);
    }
    if (req.url.includes("/web/rest/dis/") || req.url.includes("/api/vmaster-pms/v1/")) {
      return next.handle(req).do((event: HttpEvent<any>) => {
        if (event instanceof HttpResponse) {
          if (event.status === 200) {
            // Success.
            console.info("[ResponseStatusFilter] intercept(): url=", req.url);
            console.info("[ResponseStatusFilter] intercept(): method=",
req.method);
            console.info("[ResponseStatusFilter] intercept(): body=", req.body);
          }
        }
      }, (err: any) => {
        if (err instanceof HttpResponse) {
          if (err.status >= 400) {
            this.missionService.alertConfirm({
              type: "error",
              message: err.message
            });
          }
        }
      });
    }

    return next.handle(req);
  }
}
```

.....

(21) 在 app module 中挂载拦截器
app.module.ts

```
providers: [  
  {  
    provide: HTTP_INTERCEPTORS,  
    useClass: ResponseStatusFilter,  
    multi: true  
  }],
```

(22) 效果



Angular 脱坑记

之数据绑定

问题描述

Angular 6.1.5

数据绑定是 Angular 框架的一大特性，包括：

- (12) []: 组件外 --> 组件内的单向数据绑定
- (13) (): 组件内 --> 组件外的单向事件绑定
- (14) []@: banana-in-a-plate，双向数据绑定

解决

(23) 单向数据绑定

reference.html

```
<div class="modal-body">
  <ss-tree [data]="pData"
    (lResources)="addResource($event)"
    [isSelectable]="true"></ss-tree>
</div>
```

ss-tree.component.ts

```
@Component({
  selector: 'ss-tree',
  templateUrl: './tree.html',
  styles: [
    `
  ]
})
export class SSTreeComponent {
  @Input() isSelectable: boolean;
}
```

(24) 单向事件绑定

resource-tree.component.ts

```
@Component({
  selector: 'resource-tree',
  templateUrl: './tree.html',
  styles: [
    `
  ]
})
export class SSTreeComponent implements OnInit {
  @Output() clickEvent = new EventEmitter();
}
```

reference.html

```
<div style="width: 100%;">
  <div class="tree-container" style="height:200px;">
    <resource-tree #plxSelectTree
      (clickEvent)="treeNodeClick($event)"
    >
  </resource-tree>
</div>
```

```
</plx-select>
</div>
```

reference.component.ts

```
export class CustomDropdownComponent {
    treeNodeClick(data) {

    }
}
```

(25) 双向数据绑定
没用过

Angular 脱坑记

之简化项目结构（目录树）

问题描述

Angular	6.1.5
TypeScript	2.7.2

Angular2 采用 MVC（模型、视图、控制器分离）设计模式。因此，理论上应为每个页面建立 model、view、controller 三个文件夹。当页面中需要嵌入子组件时，需要为每个子组件建立 MVC 目录结构。当子组件中又包含子组件时，会使项目结构显得复杂、凌乱。以下总结一些简化项目目录树结构的心得：

解决

4. 将 Model 合并进 View

虽然具有强类型、面向对象的特点，执行 TypeScript 脚本归根结底还是需要依靠 JavaScript。不论具有多复杂、整洁的目录结构，JavaScript 脚本在编译时会被合并为单个文件。因此，【将 Model 合并进 View】在简化项目结构的同时，并不会对 Angular 的设计模式造成副作用。示例如下：

chart.component.ts

```
@Component({
    selector: 'chart',
    templateUrl: 'chart.html',
    styleUrls: ['chart.css']
})
export class ChartComponent implements OnInit {
    mChartModel: ChartModel;
    ...
}
Export class ChartModel{
```

```
var: Type;
}
```

5. 采用 ng-template 实现模态框 (Modal)

<ng-template>是 Angular2 项目的一个常用标签。Angular 语法糖*ngIf 就会被模板引擎转换为这个标签:

```
<div *ngIf="hero" class="name">{{hero.name}}</div>
```

<!-- *ngIf 翻译成 ng-template 元素之后 -->

```
<ng-template [ngIf]="hero">
  <div class="name">{{hero.name}}</div>
</ng-template>
```

product-list-page.component.html

```
<ng-template #thresholdView let-c="close" let-d="dismiss">
  <div class="modal-header">
    <h4 class="modal-title"></h4>
    <button type="button" class="close" (click)="d('Cross click')">
      <span class="plx-ico-close-16"></span>
    </button>
  </div>
  <div class="modal-body">
    <div class="div-wrapper">

    </div>
  </div>
  <div class="modal-footer" style="margin-top: -16px" *ngIf="!pIsEditMode">
    <div class="form-group w-100">
      <div class="btnGroup modal-btn mx-auto float-none">
        <button type="button" class="plx-btn" (click)="cancel()">关闭</button>
        <button type="button" class="plx-btn plx-btn-primary" (click)="confirm()">修
改</button>
      </div>
    </div>
  </div>
</ng-template>
```

product-list-page.component.ts

```
export class ThresholdPlpComponent implements OnInit {
  modal: any;
  @ViewChild('thresholdView') thresholdView: any;

  constructor(private modalService: PlxModal,
    ...
  ) {
```

```

    this.infoRepo = new BasicInfoPlpRepository();
    this.onChange = new EventEmitter<ResourceTypeRepository>();
    this.data = [];
  }

  openModal() {
    const size: 'sm' | 'lg' = 'lg';
    const options = {
      size: size,
    };
    this.modal = this.modalService.open(this.thresholdView, options);
  }
}

```

6. 使用 TypeScript 高级语言特性

Intersection Types

An intersection type combines multiple types into one. This allows you to add together existing types to get a single type that has all the features you need. For example, Person & Serializable & Loggable is a Person and Serializable and Loggable. That means an object of this type will have all members of all three types.

product-list-page.component.ts

```
srcObj = {} as BasicInfo&DateAndTime&FormulaSet&SelectedPo&SelectedResInstance;
```

□

Angular 脱坑记 番外编

之记一次排错经历

问题描述

angular	6.1.5
rx-js	6.1.0
rxjs-compat	6.1.0

rx-js 全称 the React eXtension for JavaScript。一个重要的作用是提供异步调用接口，其定义了对象 Observable、Subscription 等。

故障现象：

测试环境编译、服务正常

生产环境编译通过、全称无报错

生产环境服务，访问页面时控制台报错 “XXX: Subscription is not defined.”

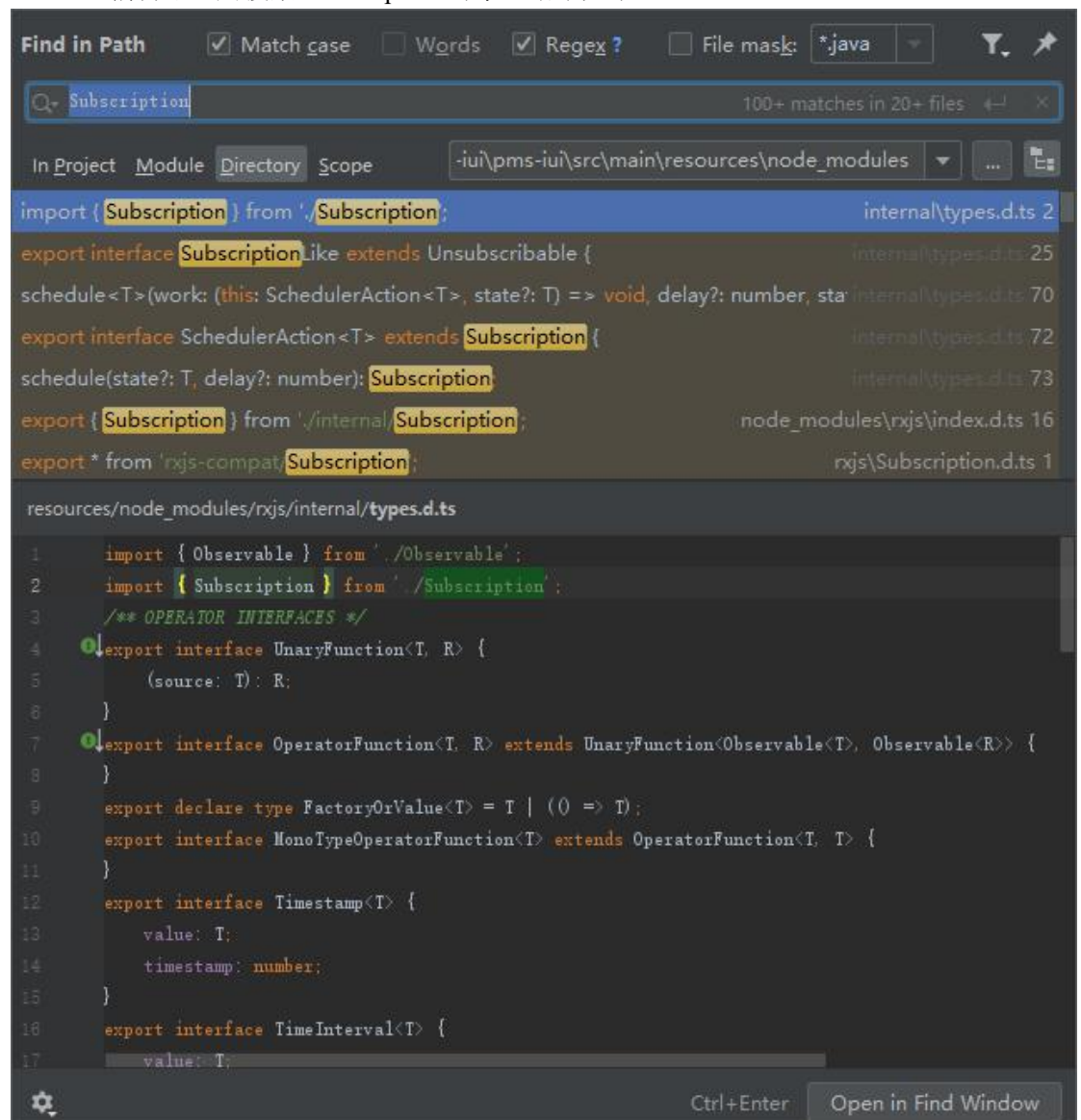
解决

(26) 验证故障现象

(27) 检查页面

在故障页面右键选“检查”，观察到页面渲染至<app-root></app-root>，故推测为依赖项相关错误。

(28) IDEA 编辑器全局搜索 Subscription 对象，结果如下：



The screenshot shows the 'Find in Path' search results in IntelliJ IDEA. The search term is 'Subscription'. The results are listed in a table with columns for the file path and the line number. The first result is 'internal/types.d.ts 2', which contains the line 'import { Subscription } from './Subscription;'. The second result is 'internal/types.d.ts 25', which contains the line 'export interface SubscriptionLike extends Unsubscribable {'. The third result is 'internal/types.d.ts 70', which contains the line 'schedule<T>(work: (this: SchedulerAction<T>, state?: T) => void, delay?: number, sta'. The fourth result is 'internal/types.d.ts 72', which contains the line 'export interface SchedulerAction<T> extends Subscription {'. The fifth result is 'internal/types.d.ts 73', which contains the line 'schedule(state?: T, delay?: number): Subscription'. The sixth result is 'node_modules\rxjs\index.d.ts 16', which contains the line 'export { Subscription } from './internal/Subscription;'. The seventh result is 'rxjs\Subscription.d.ts 1', which contains the line 'export * from 'rxjs-compat/Subscription;'. Below the search results, the content of the file 'resources/node_modules/rxjs/internal/types.d.ts' is displayed. The content shows the following code:

```
1 import { Observable } from './Observable';
2 import { Subscription } from './Subscription';
3 /** OPERATOR INTERFACES */
4 export interface UnaryFunction<T, R> {
5   (source: T): R;
6 }
7 export interface OperatorFunction<T, R> extends UnaryFunction<Observable<T>, Observable<R>> {
8 }
9 export declare type FactoryOrValue<T> = T | (() => T);
10 export interface MonoTypeOperatorFunction<T> extends OperatorFunction<T, T> {
11 }
12 export interface Timestamp<T> {
13   value: T;
14   timestamp: number;
15 }
16 export interface TimeInterval<T> {
17   value: T;
```

以上结果说明：

故障是依赖项 rx-js 导致的。

(29) 检查 package.json

package.json

```
"rxjs": "~6.1.0",  
"rxjs-compat": "~6.1.0",
```

与 npm 版本比较, 可知其版本号低于最新版本。将两个依赖项升级至 6.3.0 即可。

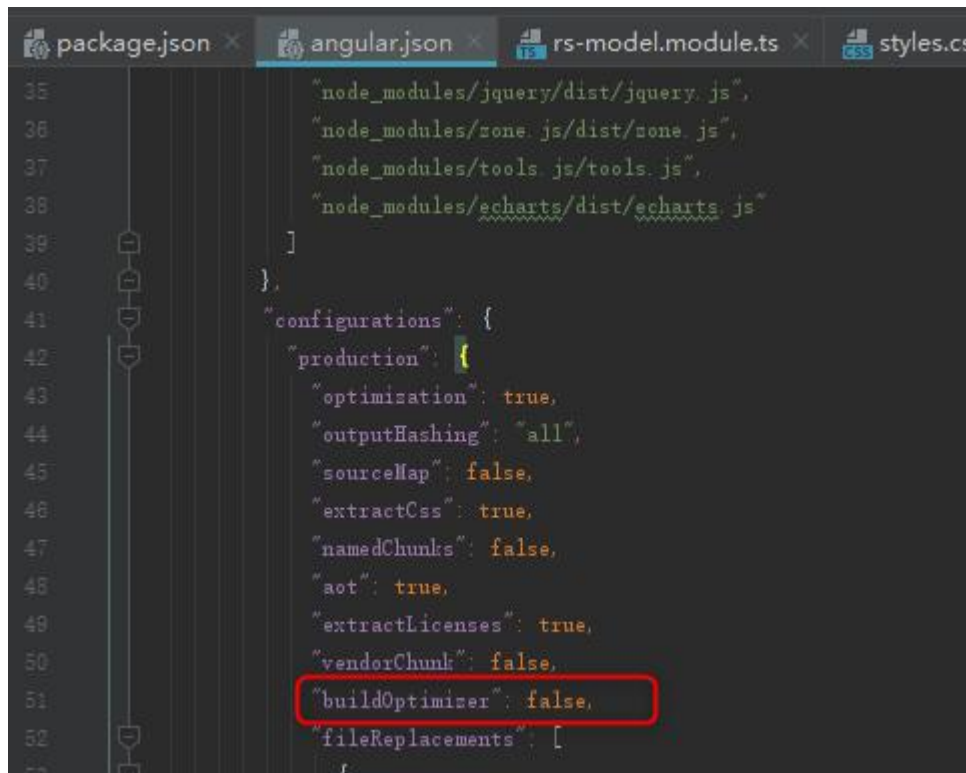
(30) 关闭代码压缩选项

关闭 optimization 选项, 在浏览器中可见原始 JS 代码 (未混淆、压缩)



(31) 进一步调试

进一步调试发现, angular-cli 的 @angular-devkit/build-optimizer 与现有依赖库函数之间存在未知的副作用, 故将其关闭:



(32) 验证

部署生产环境 Nginx 服务, 经测试, 故障已解决。

附录 build-optimizer 扩展选项：

```
export interface BuildOptimizerOptions {  
  content?: string;  
  inputFilePath?: string;  
  outputFilePath?: string;  
  emitSourceMap?: boolean;  
  strict?: boolean;  
  isSideEffectFree?: boolean;  
}
```

【小结】

4. build-optimizer 可能存在副作用（Side Effect）
5. 关闭 optimization 功能有利于在生产环境调试依赖项
6. 如无特殊需要，尽量不要使用不成熟的新技术



Angular 脱坑记

之路由调试

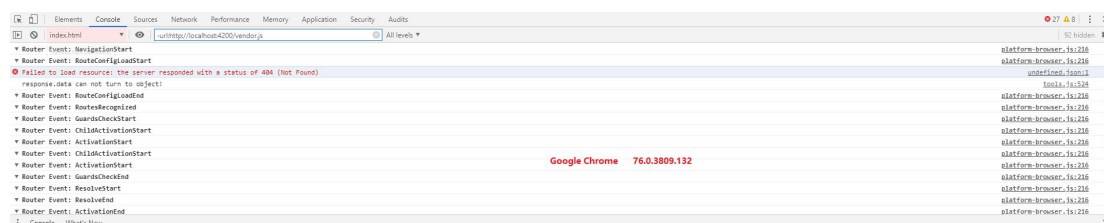
问题描述

Google Chrome 76.0.3809.132

Angular 6.1.5

enableTracing=true

不能正常打印路由调试信息



解决

使用 Edge 浏览器可正常打印路由调试日志。打开调试器会出现页面频繁刷新的现象（可能是内存溢出引起的）



Angular 脱坑记

之路由复用

问题描述

Angular 6.1.5

(15) 在基于 Angular 的 SPA 应用中，应用通过路由在各个页面之间进行导航。默认情况下，用户在离开一个页面时，这个页面(组件)会被 Angular 销毁，用户的输入信息也随之丢失，当用户再次进入这个页面时，看到的是一个新生成的页面(组件)，之前的输入信息都没了。

(16) 配置的前端项目就是基于 Angular 的，工作中遇到了这样的问题，部分页面需要保存用户的输入信息，用户再次进入页面时需要回到上一次离开时的状态，部分页面每次都要刷新页面，不需要保存用户信息。而页面间的导航正是通过路由实现的，Angular 的默认行为不能满足我们的需求！

解决

针对以上问题，通过查阅 Angular 的相关资料可以发现，Angular 提供了 RouteReuseStrategy 接口，通过实现这个接口，可以让开发者自定义路由复用策略。

(4) RouteReuseStrategy 接口

```
export abstract class RouteReuseStrategy {
  abstract shouldDetach(route: ActivatedRouteSnapshot): boolean;

  abstract store(route: ActivatedRouteSnapshot, handle: DetachedRouteHandle | null): void;

  abstract shouldAttach(route: ActivatedRouteSnapshot): boolean;

  abstract retrieve(route: ActivatedRouteSnapshot): DetachedRouteHandle | null;

  abstract shouldReuseRoute(future: ActivatedRouteSnapshot, curr: ActivatedRouteSnapshot): boolean;
}
```

这个接口只定义了 5 个方法，每个方法的作用如下：

⑥ shouldDetach

路由离开时是否需要保存页面，这是实现自定义路由复用策略最重要的一个方法。

其中：

返回值为 `true` 时，路由离开时保存页面信息，当路由再次激活时，会直接显示保存的页面。

返回值为 `false` 时，路由离开时直接销毁组件，当路由再次激活时，直接初始化为新页面。

⑦ `store`

如果 `shouldDetach` 方法返回 `true`，会调用这个方法保存页面。

⑧ `shouldAttach`

路由进入页面时是否有页面可以重用。 `true`： 重用页面， `false`： 生成新的页面

⑨ `retrieve`

路由激活时获取保存的页面，如果返回 `null`，则生成新页面

⑩ `shouldReuseRoute`

决定跳转后是否可以继续使用跳转前的路由页面，即跳转前后跳转后使用相同的页面

（5）实践

① 自定义路由复用策略

```

export class CustomRouteReuseStrategy implements RouteReuseStrategy {
  handlers: { [key: string]: DetachedRouteHandle } = {};

  shouldDetach(route: ActivatedRouteSnapshot): boolean {
    return route.data.reload === false;
  }

  store(route: ActivatedRouteSnapshot, handle: DetachedRouteHandle): void {
    this.handlers[route.routeConfig.path] = handle;
  }

  shouldAttach(route: ActivatedRouteSnapshot): boolean {
    return !!route.routeConfig
    && !!this.handlers[route.routeConfig.path];
  }

  retrieve(route: ActivatedRouteSnapshot): DetachedRouteHandle {
    return this.handlers[route.routeConfig.path];
  }

  shouldReuseRoute(future: ActivatedRouteSnapshot, curr:
  ActivatedRouteSnapshot): boolean {
    return future.routeConfig === curr.routeConfig;
  }
}

```

在这个路由复用策略中，有两个关键点：

1. 我们使用了一个 handlers 对象来保存页面。

2. 通过路由配置的 data 对象中的 reload 属性来判断一个页面是否需要保存，并且只有 reload 属性为 false 时，才会保存页面。如果不配置 reload 属性，或者 reload 属性不为 false，则不会保存页面。

② 配置路由重用策略为自定义策略

为了使用自定义的路由复用策略，需要在应用的根路由模块 providers 中使用自定义的路由复用策略

```

@NgModule ({
  imports: [RouterModule.forRoot(routes, {useHash: true})],
  exports: [RouterModule],
  providers: [
    {
      provide: RouteReuseStrategy,
      useClass: CustomRouteReuseStrategy
    }
  ]
})
export class AppRoutingModule {
}

```

③ 配置路由

在路由配置中，按需配置路由的 data 属性。如需要保存页面，则设置 reload 值为 false，如不需要保存页面，不配置该属性。例如：


```
const routes: Routes = [
  {
    path: 'foo',
    component: FooComponent
  },
  {
    path: 'bar',
    component: BarComponent,
    data: {reload: false}
  }
];
```

此路由配置下，访问/foo 页面始终会生成一个新的页面，而/bar 页面会在路由离开时会被保存，再次进入该页面都会恢复到上一次离开该页面时的状态

(6) 扩展

可以使用 Angular 路由复用策略实现 Tab（选项卡）功能（读写 cookie）

引用

- [1] 中兴开发者社区, <https://blog.csdn.net/o4dc8ojo7zl6/article/details/79224523>
- [2] angular 4 实现的 tab 栏切换, <https://www.cnblogs.com/lslgg/p/7700888.html>

Angular 脱坑记

之通过 service 在组件之间传递数据

问题描述

Angular 6.1.5

数据耦合是程序设计中需要解决的一个常见问题，上一代 jQuery+Bootstrap 前端设计模式中，常使用 window 对象挂载全局变量的方式解决对象间的数据耦合问题，而在 Angular2 框架中，由于 angular2 具有依赖注入的特性，可以通过 service 在组件之间传递数据。

解决

(33) 定义公共服务 CommonService

```
common.service.ts
import { Injectable } from '@angular/core';
@Injectable()
export class DataService {
  data:Object;
}
```

(34) 根组件中将 CommonService 加入 Provider 列表

```
app.component.ts
import { Component } from '@angular/core';
import { DataService } from './iteration-list/service/data.service';//这里在列表跟详情的
```

.....

父组件中引入 DataService

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers:[DataService]//这里添加组件的 providers 元数据
})
export class AppComponent {
  constructor() {
  }
}
```

(35) 在组件中获取 CommonService 实例

```
constructor(
  private dataService:DataService//构造 dataService
){
}
```

组件之间如需传递事件，可在公共服务中声明 EventEmitter，使用消息/订阅者设计模式。

□

Java Web 辅助工具

Java Web 辅助工具

之小辣椒 Lombok

问题描述

JDK

8

J2EE 框架已经逐渐被 Spring 全家桶取代，但是其中 Java Bean 的概念被延续下来。

Java Bean 中的构造器、Getter、Setter 方法是常见的垃圾代码，维护比较繁琐。

Project Lombok 小辣椒应运而生。

解决

@Getter

注解：生成 Getter 方法

@Setter

注解：生成 Setter 方法

@AllArgsConstructor

注解：生成全参构造器

`@Accessors(chain = true)`

注解：支持链式调用（函数式编程）

实例代码

```
@Getter
@Setter
@Accessors(chain = true)
public class GenericAnimalEvent extends ApplicationEvent {
    private String title;
    private String content;

    public GenericAnimalEvent(Object source) {
        super(source);
    }
}
```

```
new GenericAnimalEvent(this)
    .setContent("It s time to feed your animal.")
    .setTitle("Tip")
```

笔记

1. `@Accessors(chain = true)`注解支持函数式编程

□

Java 并发编程

Java 并发编程

之 Executor / Future 框架

问题描述

Spring Boot	2.2.5
JDK	8

Executor / Future 是 Java 语言中一种常用的并发编程框架：Executor 创建线程池、Future 提交线程任务（Runnable、Callable）。

解决

SearchServiceImpl.java

```
private ExecutorService executorService = Executors.newFixedThreadPool(10);

private List<AggrPath> getAllAggrPaths(AggrSearchMode searchMode, List<Identity<PerfObject>>
poIds)
{
    ...
    for (final PerfObject po : availablePos)
    {
        futures.add(executorService.submit(new Callable<Set<AggrPath>>()
        {
            @Override
            public Set<AggrPath> call() throws Exception
            {
                return getAggrPathsContainPo(po);
            }
        }));
    }
    try
    {
        for (Future<Set<AggrPath>> future : futures)
        {
            paths.addAll(future.get());
        }
    }
}
```

```

        catch (Exception e)
        {
            throw new RuntimeException(e);
        }
        return new ArrayList<AggrPath>(paths);
    }

```

□

Java 并发编程

之 ThreadPoolExecutor 工具类

问题描述

Spring Boot	2.2.5
JDK	8

ThreadPoolExecutor 是 Java 并发编程中一个常用的工具类。

解决

CleanTimerTask.java

```

private ThreadPoolExecutor createThreadPoolExecutor()
{
    return new ThreadPoolExecutor(threadPoolSize, threadPoolSize, 1L, TimeUnit.SECONDS,
        new SynchronousQueue<Runnable>(), new NamedPrefixThreadFactory("RECYCLEBIN",
false),
        new RetryRejectExecutionHandler())
    {
        @Override
        protected void beforeExecute(Thread t, Runnable r)
        {
            ...
        }

        @Override
        protected void afterExecute(Runnable r, Throwable t)
        {
            ...
        }
    }
}

```

```
    }  
};  
}  
  
private void cleanAllPoTables()  
{  
    ...  
  
    ThreadPoolExecutor executor = createThreadPoolExecutor();  
  
    while (it.hasNext())  
    {  
        executor.execute(...);  
    }  
  
    executor.shutdown();  
    try  
    {  
        //等待最多 600 秒  
        executor.awaitTermination(600, TimeUnit.SECONDS);  
    }  
    catch (InterruptedException e)  
    {  
        ...  
    }  
    if (!isTimeout)  
    {  
        ...  
    }  
  
    ...  
}
```

□

之单例模式：双重检查加锁

问题描述

JDK

8

Java 中实现单例模式的方法不止一种，较为简单、常见的实现为双重检查加锁。

双重检查加锁在理论上不能严格保证单一实例，但是已经成为了项目的最佳实践

解决

```
/**
 * 同步的初始化
 */
private static void doubleCheckedSynInit()
{
    if (dataTypeToRegexProperties == null)
    {
        synchronized (DataTypeUtil.class)
        {
            if (dataTypeToRegexProperties == null)
            {
                init();
            }
        }
    }
}
```

□

问题描述

JDK

8

Java 中“单例模式”有以下几种实现方式：

1. 饿汉式
2. 懒汉式
3. 双重检查加锁（线程安全）
4. 静态内部类
5. 单元素的枚举类型

由单元素的枚举类型实现的单例模式由 JVM 保证其线程安全性，且可以规避序列化/反序列化操作对单例模式的破坏

解决

```
public class SingletonClass {
    public static void main(String[] args) {
        ProductSingleton singleton = ProductSingleton.INSTANCE
            .setId(1001L)
            .setName("Product A")
            .setDescription("Product A description")
            .setPrice(1.0D);

        System.out.println(JSON.toJSONString(singleton)); // INSTANCE.
    }
}

/**
 * 单元素的枚举类型已经成为实现 Singleton 的最佳方式
 * —— 《Effective Java》
 */
enum ProductSingleton {
    INSTANCE;

    /**
     * Java Bean
     */
    private Long id;
```

```
private String name;
private String description;
private Double price;

/**
 * 机器代码：生成的 Getter、Setter 方法（链式调用）。
 * @return
 */
public Long getId() {
    return id;
}

public ProductSingleton setId(Long id) {
    this.id = id;
    return this;
}

public String getName() {
    return name;
}

public ProductSingleton setName(String name) {
    this.name = name;
    return this;
}

public String getDescription() {
    return description;
}

public ProductSingleton setDescription(String description) {
    this.description = description;
    return this;
}

public Double getPrice() {
    return price;
}

public ProductSingleton setPrice(Double price) {
    this.price = price;
    return this;
}
}
```

笔记

1. 单元素的枚举类型已成为实现单例的最佳方式

□

Java 并发编程

之锁

问题描述

Spring Boot	2.2.5
JDK	8

线程安全性问题出现的条件：

- 多线程环境
- 共享资源
- 非原子性操作

线程安全性问题解决方案：

- 同步监视器 `synchronized`
- 线程局部变量 `ThreadLocal<T>`
- 原子包装类 `Atomic`

同步监视器将同步代码块中的请求串行化，效率较低，因此出现了锁。`ReentrantLock`、`ReentrantReadWriteLock` 是实战中常用的锁。

解决

□

JVM 原理手记

JVM 原理手记

之 static 关键字

问题描述

Jvm 1.8

static 关键字，其作用相当于 C 语言中的全局变量 global。Static 关键字修饰的变量存储在堆内存的静态区

Java 工具类 SimpleDateFormat 不要声明在静态代码段中

解决方案

```
public class AppleClass{
    public static void main(String[] args) {
        AppleClass pw1 = new AppleClass();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        AppleClass pw2 = new AppleClass();
        System.out.println(pw1.VAR );
        System.out.println(pw2.VAR );
    }
    private static final String VAR = new SimpleDateFormat("yyyy_MM_dd_HH_mm_ss").format(new
Date().getTime());
}
```

笔记

1. 基本功：深入理解 JVM 内存模型

□

问题描述

再说空指针

解决方案

怎样处理空指针异常？

- (1) 显式判断空指针异常

```
if (foo != null) {  
    return foo.value;  
} else {  
    throw new NullPointerException();  
}
```

- (2) 隐式捕捉空指针异常

```
try {  
    return foo.value;  
} catch (segment_fault) {  
    uncommon_trap();  
}
```

两种方式的优劣？

虚拟机会注册一个 Segment Fault 信号的异常处理器（伪代码中的 `uncommon_trap()`），这样当 `foo` 不为空的时候，对 `value` 的访问是不会额外消耗一次对 `foo` 判空的开销的。代价就是当 `foo` 真的为空时，必须转入到异常处理器中恢复并抛出 `NullPointerException` 异常，这个过程必须从用户态转到内核态中处理，处理结束后再回到用户态，速度远比一次判空检查慢。当 `foo` 极少为空的时候，隐式异常优化是值得的，但假如 `foo` 经常为空的话，这样的优化反而会让程序更慢

一句话小结？

空值出现频率较低时，使用 `try-catch` 块注册异常处理器；
反之，显式判断空指针。

□

之异常：方法返回值中使用泛型

问题描述

Jvm 1.8

泛型（Generic Type）是 java 语言的一个特性。泛型的使用方法丰富多样，这个问题是在方法返回值中使用泛型时遇到的。

问题再现：

```
public class FAQBean {  
  
    private static int fun(Object o, Integer i, Object... args) {  
        return args.length;  
    }  
  
    public static <T> T parse(String s) {  
        return (T) s;  
    }  
  
    public static void main(String[] args) {  
        fun(null, 0, parse("hello world")); // Ln 1.  
    }  
}
```

说明：

1. <T>的地位等同于 public、static，可以看成 java 语法中一个特殊的关键字：“泛型关键字”。其语义为：声明一个泛型，符号为 T
2. 在方法返回值中使用泛型时，jvm 是根据接收返回值的变量类型对泛型进行推断（deduction）的
3. 执行到【Ln 1】行处会报运行时错误：**java.lang.ClassCastException: java.lang.String cannot be cast to [Ljava.lang.Object**

解决方案

将【Ln 1】修改为：

```
fun(null, 0, (Object) parse("hello world"));
```

在项目升级改造中遇到的问题，可能与 jvm 实现有关。也可能是 jvm 的一个缺陷。

□

问题描述

Jvm 1.8

Spring5 支持全注解配置，将注解（Annotation）这一历史悠久的 Java 特性拉回了时尚的舞台

示例代码演示了一个用于绑定配置项的注解@ParamTemplate

解决方案

```
/*
 * 元注解@Target,@Retention,@Documented,@Inherited
 *
 *      @Target 表示该注解用于什么地方，可能的 ElementType 参数包括：
 *          ElementType.CONSTRUCTOR 构造器声明
 *          ElementType.FIELD 域声明（包括 enum 实例）
 *          ElementType.LOCAL_VARIABLE 局部变量声明
 *          ElementType.METHOD 方法声明
 *          ElementType.PACKAGE 包声明
 *          ElementType.PARAMETER 参数声明
 *          ElementType.TYPE 类，接口（包括注解类型）或 enum 声明
 *
 *      @Retention 表示在什么级别保存该注解信息。可选的 RetentionPolicy 参数包
括：
 *          RetentionPolicy.SOURCE 注解将被编译器丢弃
 *          RetentionPolicy.CLASS 注解在 class 文件中可用，但会被 VM 丢弃
 *          RetentionPolicy.RUNTIME VM 将在运行期也保留注释，因此可以通过反射
机制读取注解的信息。
 *
 *      @Documented 将此注解包含在 javadoc 中
 *
 *      @Inherited 允许子类继承父类中的注解
 */
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
public @interface ParamTemplate {
```

```
.....
public String key();
}
```

笔记

1. 使用 `Class.getDeclaredFields()`方法获取注解
2. 使用 `Field.isAnnotationPresent()`方法检查注解是否存在
3. 使用 `Field.getAnnotation()`方法获得注解

□

Maven 笔记

Maven 笔记

之 Maven 三驾马车——作用域

问题描述

Spring Boot	2.2.5
JDK	8

Maven 中定义了六种作用域：

- ✓ Compile
- ✓ Provided
- ✓ Runtime
- ✓ Test
- ✓ System
- ✓ Import

Jar 包的作用有两种：编译时（`compile`）和运行时（`runtime`）。

若编译时、运行时都需要，则应将作用域定义为 `provided`

若依赖项（通常为 Jar 包）不是从 Maven 仓库获取，而是制定为本地文件，则应将作用域定义为 `System`

作用域 `Import` 与依赖管理（`Dependency Management`）有关

Maven 笔记

之 Maven 三驾马车——依赖管理

问题描述

Spring Boot	2.2.5
JDK	8

Maven 官方文档这样介绍依赖管理（dependency management）：

The dependency management section is a mechanism for centralizing dependency information. When you have a set of projects that inherit from a common parent, it's possible to put all information about the dependency in the common POM and have simpler references to the artifacts in the child POMs.

Maven 中依赖管理（dependency management）的概念可以类比程序设计中“重构（refactor）”的概念，即将公共依赖项的配置信息提取到父节点（POM）中。

依赖管理可以减少重复的依赖项配置，在子节点（POM）中只保留对公共依赖的引用。如同代码重构时，将公共代码提取为函数，放置于父类（抽象类）中，而在子类中仅保留函数句柄（function handle）

作用域（Scope）Import 的作用，正如其表面意义所指，用于导入 POM 中定义的依赖管理（Dependency management）项

Maven 笔记

之 Maven 三驾马车——序

问题描述

Spring Boot	2.2.5
JDK	8

面向对象（Object Oriented）设计理念使得程序设计中的设计、编码过程更为简单、自然。使用面向对象的理论进行程序设计，即为面向对象程序设计（OOP）。使用面向对象的理论进行依赖管理，就产生了一种新的工具——Maven

正如面向对象编程中对继承与多态的支持，面向对象依赖管理工具 Maven 同样支持依赖传递（Dependency mediation）与版本（version）

Maven 项目的目录结构如下所示：

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |       |-- com
    |           |-- mycompany
    |               |-- app
    |                   |-- App.java
    |-- test
    |   |-- java
    |       |-- com
    |           |-- mycompany
    |               |-- app
    |                   |-- AppTest.java
```

□

Maven 笔记

之 Maven 三驾马车——拆分多模块应用

问题描述

Spring Boot	2.2.5
JDK	8

父节点（POM）定义为 pom 类型、子节点（POM）定义为 jar 类型

解决方案

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId></groupId>
  <artifactId></artifactId>
  <packaging>pom</packaging>
  <version></version>
  <name></name>
  <url></url>

  <modules>
    <module>sdk</module>
    <module>api</module>
    <module>docker</module>
  </modules>

  <dependencies>

  </dependencies>
</project>
```

□

Maven 笔记

之 Spring Boot 项目引入外部 jar 包

问题描述

Spring Boot	2.2.5
JDK	8

spring-boot-maven-plugin 是 Spring Boot 项目的打包工具。在项目实践中，会遇到需要引用外部 Jar 包的情况。比如，一些陈旧的、或未使用 Maven 仓库管理的 Jar 包。

在 Spring Boot 项目中引用外部 Jar 包的方式不止一种，这里记录较易实现的解决方案。

解决

1. 添加依赖项 制定 scope 为 system

```
<dependency>
  <groupId>com. supermap</groupId>
  <artifactId>com. supermap. data</artifactId>
  <version>10. 1</version>
  <scope>system</scope>
  <systemPath>F:/supermapJars/com. supermap. data. jar</systemPath>
</dependency>
```

2. 添加插件 spring-boot-maven-plugin 配置

```
<plugin>
  <groupId>org. springframework. boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <includeSystemScope>true</includeSystemScope>
  </configuration>
</plugin>
```

□

Spring & Spring Boot

Spring Boot

之@Controller

问题描述

Spring Boot	2.2.5
JDK	8

The @Controller annotation indicates that a particular class serves the role of a controller. There is no need to extend any controller base class or reference the Servlet API. You are of course still able to reference Servlet-specific features if you need to.

The basic purpose of the @Controller annotation is to act as a stereotype for the annotated class, indicating its role. The dispatcher will scan such annotated classes for mapped methods, detecting @RequestMapping annotations (see the next section).

Annotated controller beans may be defined explicitly, using a standard Spring bean definition in the dispatcher's context. However, the @Controller stereotype also allows for autodetection, aligned with Spring 2.5's general support for detecting component classes in the classpath and auto-registering bean definitions for them.

解决

Spring Boot 中提供新的注解 @RestController，相当于 Spring 注解 @Controller、@ResponseBody 的联合体。

□

Spring Boot

之@RequestMapping

问题描述

Spring Boot	2.2.5
JDK	8

The @RequestMapping annotation is used to map URLs like '/editPet.do' onto an entire class or a particular handler method. Typically the type-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations 'narrowing' the primary mapping for a specific HTTP method request method ("GET"/"POST") or specific HTTP request parameters.

Spring Boot 中对注解@RequestMapping 做了进一步的封装: @GetMapping、@PostMapping 等。

解决

EditPetForm.java

```
@Controller
@RequestMapping("/editPet.do")
@SessionAttributes("pet")
public class EditPetForm {

    private final Clinic clinic;

    @Autowired
    public EditPetForm(Clinic clinic) {
        this.clinic = clinic;
    }

    @ModelAttribute("types")
    public Collection<PetType> populatePetTypes() {
        return this.clinic.getPetTypes();
    }

    @RequestMapping(method = RequestMethod.GET)
    public String setupForm(@RequestParam("petId") int petId, ModelMap model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
    }
}
```

```

        return "petForm";
    }

    @RequestMapping(method = RequestMethod.POST)
    public String processSubmit(
        @ModelAttribute("pet") Pet pet, BindingResult result, SessionStatus status) {

        new PetValidator().validate(pet, result);
        if (result.hasErrors()) {
            return "petForm";
        }
        else {
            this.clinic.storePet(pet);
            status.setComplete();
            return "redirect:owner.do?ownerId=" + pet.getOwner().getId();
        }
    }
}

```

□

Spring Boot

之@RequestParam

问题描述

Spring Boot	2.2.5
JDK	8

The @RequestParam annotation is used to bind request parameters to a method parameter in your controller.

Parameters using this annotation are required by default, but you can specify that a parameter is optional by setting @RequestParam's required attribute to false (e.g., @RequestParam(value="id", required="false")).

解决

EditPetForm.java

```

@Controller
@RequestMapping("/editPet.do")

```

```
.....  
@SessionAttributes("pet")  
public class EditPetForm {  
  
    // ...  
  
    @RequestMapping(method = RequestMethod.GET)  
    public String setupForm(@RequestParam("petId") int petId, ModelMap model) {  
        Pet pet = this.clinic.loadPet(petId);  
        model.addAttribute("pet", pet);  
        return "petForm";  
    }  
  
    // ...  
}
```



Spring Boot

之 application.yml 常用配置

问题描述

Spring Boot	2.2.5
JDK	8

Spring 5 与 Spring 3 是两个重要的发布版本，在配置方式上有以下不同：

1. 使用全注解配置代替基于**`-context.xml`的 IoC 容器配置
(1) 使用`@Service`、`@Autowired`代替 Application Context Utility 帮助类
2. 使用 `application.yml` 代替 `application.properties`

解决

1. 调用环境变量

application.yml

```
server:  
  port: ${PORT:8089}
```

2. 配置 profile

application.yml

```
spring:  
  profiles:  
    active: ${ENV:dev}
```

application-dev.yml

application-pro.yml

□

Spring Boot

之 JMS 消息模型

问题描述

Spring Boot	2.2.5
JDK	8

Java 内的消息传递有两种实现方案：

- JVM 内部消息：使用 JMS 消息模型
- 外部消息：使用 MQ 消息队列（Kafka、ActiveMQ 等）

日出东方，唯我不败。Spring Framework 为我们提供了 Java Web 一站式解决方案。其中就包括通过注解封装的 JMS 消息模型

解决

1. 定义消息体

```
@Getter
@Setter
@Accessors(chain = true)
public class GenericAnimalEvent extends ApplicationEvent {
    private String title;
    private String content;

    public GenericAnimalEvent(Object source) {
        super(source);
    }
}
```

```
}
```

2. 实现、注册消息监听器（Listener）

(1) 实现监听器

```
public class FeedAnimalListener {  
    @EventListener  
    public void handleEvent(GenericAnimalEvent genericAnimalEvent) {  
        System.out.println(genericAnimalEvent.getSource());  
        System.out.println("Feed animal done.");  
    }  
}
```

(2) 注册监听器

```
@Configuration  
@ComponentScan  
public class ApplicationBeanConfig {  
  
    @Bean  
    public FeedAnimalListener getFeedAnimalListener() {  
        return new FeedAnimalListener();  
    }  
}
```

3. 发布消息

```
@Service  
public class AnimalQueryService implements ApplicationEventPublisherAware {  
  
    private ApplicationEventPublisher applicationEventPublisher;  
  
    public void sendMessage() {  
        this.applicationEventPublisher.publishEvent(  
            new GenericAnimalEvent(this)  
                .setContent("It s time to feed your animal.")  
                .setTitle("Tip")  
        );  
    }  
  
    @Override  
    public void setApplicationEventPublisher(ApplicationEventPublisher  
applicationEventPublisher) {  
        this.applicationEventPublisher = applicationEventPublisher;  
    }  
}
```

笔记

1. 消息体中的字段使用 `final` 关键字定义更佳
2. 使用注解 `@EventListener` 声明消息监听器 (Listener)
3. 实现 `ApplicationEventPublisherAware` 接口 (隶属 `Aware` 接口组) 发布消息
4. Spring5 使用全注解配置
 - (1) `applicationContext.xml` 替换为 `@Bean` 注解
 - (2) Java Bean 声明、初始化更为灵活



Spring Boot

之 servlet-api 依赖冲突

问题描述

Spring Boot	2.2.5
JDK	8

```
*****
APPLICATION FAILED TO START
*****

Description:

An attempt was made to call a method that does not exist. The attempt was made from the following
location:

org.apache.catalina.authenticator.AuthenticatorBase.startInternal(AuthenticatorBase.java:132
1)

The following method did not exist:

javax.servlet.ServletContext.getVirtualServerName()Ljava/lang/String;
```


The method's class, `javax.servlet.ServletContext`, is available from the following locations:

```
jar:file:/D:/Program%20Files/ProjectManagement/apache-maven-3.3.9/Repository/javax/servlet/servlet-api/2.5/servlet-api-2.5.jar!/javax/servlet/ServletContext.class
```

```
jar:file:/D:/Program%20Files/ProjectManagement/apache-maven-3.3.9/Repository/org/apache/tomcat/embed/tomcat-embed-core/9.0.33/tomcat-embed-core-9.0.33.jar!/javax/servlet/ServletContext.class
```

It was loaded from the following location:

```
file:/D:/Program%20Files/ProjectManagement/apache-maven-3.3.9/Repository/javax/servlet/servlet-api/2.5/servlet-api-2.5.jar
```

Action:

Correct the classpath of your application so that it contains a single, compatible version of `javax.servlet.ServletContext`

解决

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
    <version>2.2.2.RELEASE</version>
    <exclusions>
      <exclusion>
        <artifactId>servlet-api</artifactId>
        <groupId>javax.servlet</groupId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>
```

□

Spring Boot

之 Spring Cache- Ehcache 实现

问题描述

Spring Boot	2.2.5
JDK	8

Spring Cache 可以由多种 Cache Provider 提供功能支持，常用的有：

1. Jdk ConcurrentMap 实现
2. Ehcache 实现
3. Redis 实现

这篇技术笔记记录了 Ehcache 实现。

Ehcache 具有以下特点：

- 进程内缓存框架
- 支持内存、磁盘二级缓存：应用启动时加载、退出时持久化
- 支持分布式
- 对多节点同步的支持不理想
- Hibernate 框架御用缓存
- 适合单体应用

解决

1. 添加依赖

```
<dependency>
  <groupId>net.sf.ehcache</groupId>
  <artifactId>ehcache</artifactId>
  <version>2.10.6</version>
</dependency>
```

2. 添加配置类

```
@Configuration
public class CacheConfig {

    @Bean
```

```

public CacheManager getCacheManager() {
    ResourceLoader loader = new DefaultResourceLoader();
    Resource ehcacheConfig = loader.getResource("ehcache/ehcache.xml");
    EhCacheManagerFactoryBean factoryBean = new EhCacheManagerFactoryBean();
    factoryBean.setConfigLocation(ehcacheConfig);
    factoryBean.afterPropertiesSet();
    return factoryBean.getObject();
}

@Bean
public EhCacheCacheManager getCacheCacheManager() {
    EhCacheCacheManager cacheCacheManager = new EhCacheCacheManager();
    cacheCacheManager.setCacheManager(getCacheManager());
    cacheCacheManager.afterPropertiesSet();
    return cacheCacheManager;
}
}

```

笔记

1. Spring 5.3 官方文档中是基于 Ehcache 2 进行配置的



Spring Boot

之 Spring Cache- Jdk 实现

问题描述

Spring Boot	2.2.5
JDK	8

Spring Cache 可以由多种 Cache Provider 提供功能支持，常用的有：

4. Jdk ConcurrentMap 实现
5. Ehcache 实现
6. Redis 实现

这篇技术笔记记录了 Jdk ConcurrentMap 实现。

Spring5 全注解配置对 Cache Provider 不大友好。

解决

```
@Configuration
public class CacheConfig {

    @Bean
    public SimpleCacheManager getCacheManager() {
        List<ConcurrentMapCache> caches = new LinkedList<>();
        ConcurrentMapCacheFactoryBean factoryBean = new ConcurrentMapCacheFactoryBean();
        factoryBean.setName("products");
        factoryBean.afterPropertiesSet();
        caches.add(factoryBean.getObject());

        SimpleCacheManager manager = new SimpleCacheManager();
        manager.setCaches(caches);
        return manager;
    }
}
```

□

Spring Boot

之 Spring Cache- Redis 实现

问题描述

Spring Boot	2.2.5
JDK	8

Spring Cache 可以由多种 Cache Provider 提供功能支持，常用的有：

7. Jdk ConcurrentMap 实现
8. Ehcache 实现
9. Redis 实现

这篇技术笔记记录了 Redis 实现。

Redis 是分布式缓存的“老江湖”了。

Redis 采用 Sentinel（哨兵）机制进行投票，探测成员状态（Alive）。

解决

3. 安装 Redis 服务

4. 添加依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-redis</artifactId>
  <version>2.3.0.RELEASE</version>
</dependency>
```

5. 添加配置类

```
@Configuration
public class CacheConfig {
    // Redis Implementation

    @Bean
    public CacheManager cacheManager(RedisConnectionFactory lettuceConnectionFactory) {
        RedisCacheConfiguration defaultCacheConfig =
RedisCacheConfiguration.defaultCacheConfig();
        // 设置缓存管理器管理的缓存的默认过期时间
        defaultCacheConfig = defaultCacheConfig.entryTtl(Duration.ofSeconds(240))
            // 设置 key 为 string 序列化
            .serializeKeysWith(RedisSerializationContext.SerializationPair.fromSerializer(new StringRedisSerializer()))
            // 设置 value 为 json 序列化
            .serializeValuesWith(RedisSerializationContext.SerializationPair.fromSerializer(new GenericJackson2JsonRedisSerializer()))
            // 不缓存空值
            .disableCachingNullValues();

        Set<String> cacheNames = new HashSet<>();
        cacheNames.add("products");

        // 对每个缓存空间应用不同的配置
        Map<String, RedisCacheConfiguration> configMap = new HashMap<>();
        configMap.put("products", defaultCacheConfig.entryTtl(Duration.ofSeconds(180)));

        RedisCacheManager cacheManager = RedisCacheManager.builder(lettuceConnectionFactory)
            .cacheDefaults(defaultCacheConfig)
            .initialCacheNames(cacheNames)
            .withInitialCacheConfigurations(configMap)
```

```

        .build();
        return cacheManager;
    }
}

```

笔记

1. Java Bean Lettuce Connection Factory 是由 spring-boot-autoconfigure 注入的:



□

Spring Boot

之 Spring Cache- 抽象层

问题描述

Spring Boot	2.2.5
JDK	8

日出东方，唯我不败。Spring 框架作为大一统 Java 后端的万能框架，同样提供了缓存（Cache）的封装类。缓存的实现类分散在 spring-context、spring-context-support 两个 Jar 包中。

Spring5 对 Cache 的实现进行了重构，主要由 Cache Manager、Cache Resolver、AbstractValueAdaptingCache 等 Java 类实现其功能。

Spring Cache 的配置遵循“老三样”原则，即添加依赖、使能模块、添加注解。

解决

4. 添加依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-cache</artifactId>
  <version>2.3.0.RELEASE</version>
</dependency>
```

5. 使能模块

```
@SpringBootApplication
@MapperScan("com.bunny.spring.framework.dao.products.ibatis")
@EnableCaching
public class DAOApplication {
    public static void main(String[] args) {
        SpringApplication.run(DAOApplication.class);
    }
}
```

6. 添加注解

```
/**
 *
 * Spring Cache 采用“懒加载”策略
 * 未实现的场景：
 * 由单次查询触发，取出（关联）表中所有数据项（生成项），载入缓存
 *
 */
@CacheConfig(cacheNames = "products")
@Service
public class JdbcDaoService {

    @Resource(name = "jdbcProductDao")
    private ProductsDao jdbcProductDao;

    @Cacheable
    public Product queryById(Long id) throws Exception {
        return jdbcProductDao.queryById(id);
    }

    @CachePut
    public void update(Product product) throws Exception {
        jdbcProductDao.update(product);
    }

    @CacheEvict(allEntries = true)
```

```
public Boolean evict() {  
    return true;  
}
```

笔记

5. Spring Cache 功能尚不完备，不支持以下场景：

- (1) 轻量级锁 Spring Cache 使用重量级锁 `synchronized`，同步机制不可配
- (2) 一次触发，全部取出 即不遵循“懒加载”原则，由单次查询触发，加载表（关联表）中所有数据项（生成项）到缓存中

□

Spring Boot

之 Spring DAO- Hibernate 实现

问题描述

Spring Boot	2.2.5
JDK	8

数据库是海量数据存储、管理的强大工具，是 APP 后台不可或缺的基础设施。Spring 框架提供了面向对象的数据库接口（DAO）、实体-关系映射（ORM）组件的整合方法

这篇技术笔记记录了 Hibernate 实现方式

解决

7. 添加 Maven 依赖

```
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-orm</artifactId>  
    <version>5.2.6.RELEASE</version>
```



```

</dependency>

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.16.Final</version>
</dependency>

```

8. 配置 SessionFactory、HibernateTemplate

```

@Configuration
public class HibernateConfig {

    @Autowired
    private DruidDataSource dataSource;

    @Bean
    public SessionFactory getSessionFactory() {
        LocalSessionFactoryBean factoryBean = new LocalSessionFactoryBean();
        factoryBean.setDataSource(dataSource);
        factoryBean.setMappingResources(
            "hibernate/Product.hbm.xml"
        );
        Properties hibernateProperties = new Properties();
        try {
            hibernateProperties.load(new BufferedInputStream(new FileInputStream(new
File("src/main/resources/hibernate/hibernate.conf"))));
        } catch (IOException e) {
            e.printStackTrace();
        }
        factoryBean.setHibernateProperties(hibernateProperties);
        try {
            factoryBean.afterPropertiesSet();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return factoryBean.getObject();
    }

    @Bean
    public HibernateTemplate getHibernateTemplate() {
        return new HibernateTemplate(getSessionFactory());
    }
}

```

9. 定义映射关系

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- 文件名 Product.hbm.xml 的第一个字母(P)一定要大写,要和数据库对应的实体类保持一致 -->
<!-- 配置文件 Product.hbm.xml, 用于映射 Product 类对应数据库中的 product_表 -->
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.bunny.spring.framework.entity">
    <!-- 表明类 Product 对应数据库中的表 products -->
    <class name="Product" table="products">
        <!-- 属性 id 映射表里面的字段 id -->
        <id name="id" column="id">
            <!-- generator 表示 id 的自增长方式采用数据库的本地方式 -->
            <generator class="native">
            </generator>
        </id>
        <!-- 配置属性映射字段 -->
        <property name="name" />
        <property name="description" />
        <property name="price" />
    </class>
</hibernate-mapping>
```

10. (可选) Hibernate 配置项

hibernate.conf

```
hibernate.dialect=org.hibernate.dialect.MySQL57Dialect
```

笔记

6. Spring5 全注解配置整合 Hibernate 效果不好

7. 常见报错 Unknown table 'system_sequences' in information_schema: 方言 (Dialect) 没配对

8. 使用注解配置映射关系

https://docs.jboss.org/hibernate/orm/5.4/quickstart/html_single/#hibernate-gsg-tutorial-annotations-entity

9. 项目地址

<https://github.com/MariaLikesFish/spring-orm-combo>



Spring Boot

之 Spring DAO- JDBC 实现

问题描述

Spring Boot	2.2.5
JDK	8

数据库是海量数据存储、管理的强大工具，是 APP 后台不可或缺的基础设施。Spring 框架提供了面向对象的数据库接口（DAO）、实体-关系映射（ORM）组件的整合方法

这篇技术笔记记录了 JDBC 实现方式

在 Spring5 全注解配置中，@Repository 注解与@Resource 注解配对使用：@Repository 注解用 value 值在上下文中注册 DAO 对象；@Resource 注解使用 name 值在 Service 类中注入 DAO 对象

解决

11. 标记 DAO

```
@Repository(value = "jdbcProductDao")
public class ProductsDaoI implements ProductsDao {

    @Autowired
    private JdbcTemplate template;

    public Product queryById(Long id) throws Exception {
        // string builder 支持链式调用.
        StringBuilder sb = new StringBuilder("SELECT * FROM products WHERE id=")
            .append(id);
        try {
            return template.query(sb.toString(), new RowMapper<Product>() {

                @Override
```

```

        public Product mapRow(ResultSet resultSet, int i) throws SQLException {
            Long id = resultSet.getLong(1);
            String name = resultSet.getString(2);
            String description = resultSet.getString(3);
            Long price = resultSet.getLong(4);
            return new Product(id, name, description, price);
        }
    }).get(0);
} catch (Exception e) {
    throw new Exception();
}
}
}

```

12. 标记实现类

```

@Service
public class JdbcDaoService {

    @Resource(name = "jdbcProductDao")
    private ProductsDao jdbcProductDao;

    public Product queryById(Long id) throws Exception {
        return jdbcProductDao.queryById(id);
    }
}

```

13. 异常处理

笔记

10. 私有框架可实现 `SqlTemplateManager` 类, 将 SQL 碎块重构到 XML 文件中集中配置

11. The best way to guarantee that your Data Access Objects (DAOs) or repositories provide exception translation is to use the `@Repository` annotation. This annotation also lets the component scanning support find and configure your DAOs and repositories without having to provide XML configuration entries for them.

12. JDBC 实现是项目中可用的方案

□

Spring Boot

之 Spring DAO- MyBatis 实现

问题描述

Spring Boot	2.2.5
JDK	8

数据库是海量数据存储、管理的强大工具,是 APP 后台不可或缺的基础设施。
Spring 框架提供了面向对象的数据库接口 (DAO)、实体-关系映射 (ORM) 组件的整合方法

这篇技术笔记记录了 MyBatis 实现方式

在 Spring5 全注解配置中,使用 @Mapper 注解配置 MyBatis Mapper、使用 @MapperScan 指定 Mapper 扫描路径。Type alias、type handler 类路径写在 YML 配置文件中

解决

14. 添加 Maven 依赖

```
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>2.1.2</version>
</dependency>
```

15. 配置 Mapper 扫描路径

```
@SpringBootApplication
@MapperScan("com.bunny.spring.framework.dao.products.ibatis")
public class DAOApplication {
    public static void main(String[] args) {
        SpringApplication.run(DAOApplication.class);
    }
}
```

16. 实现 Mapper 类

```

@Mapper
public interface ProductsMapper {

    @Select("SELECT * FROM products WHERE id = #{id}")
    public Product queryById(Long id);

}

```

17. （可选）配置 Type Alias、Type Handler 扫描路径

application.yml

```

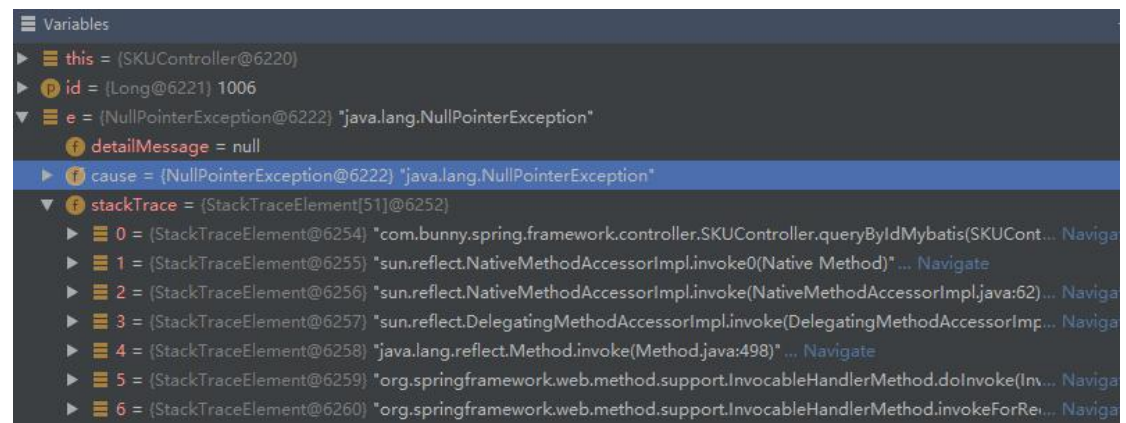
mybatis:
  type-aliases-package: com.bunny.spring.framework.entity

```

笔记

13. 使用全注解配置简洁工整

14. MyBatis 对 Spring 框架的整合没有做 Exception Translation:



```

Variables
this = {SKUController@6220}
id = {Long@6221} 1006
e = {NullPointerException@6222} "java.lang.NullPointerException"
  detailMessage = null
  cause = {NullPointerException@6222} "java.lang.NullPointerException"
  stackTrace = {StackTraceElement[51]@6252}
    0 = {StackTraceElement@6254} "com.bunny.spring.framework.controller.SKUController.queryByIdMybatis(SKUCont... Naviga
    1 = {StackTraceElement@6255} "sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)" ... Naviga
    2 = {StackTraceElement@6256} "sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) ... Naviga
    3 = {StackTraceElement@6257} "sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImp... Naviga
    4 = {StackTraceElement@6258} "java.lang.reflect.Method.invoke(Method.java:498)" ... Naviga
    5 = {StackTraceElement@6259} "org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(In... Naviga
    6 = {StackTraceElement@6260} "org.springframework.web.method.support.InvocableHandlerMethod.invokeForRei... Naviga

```

□

Spring Boot

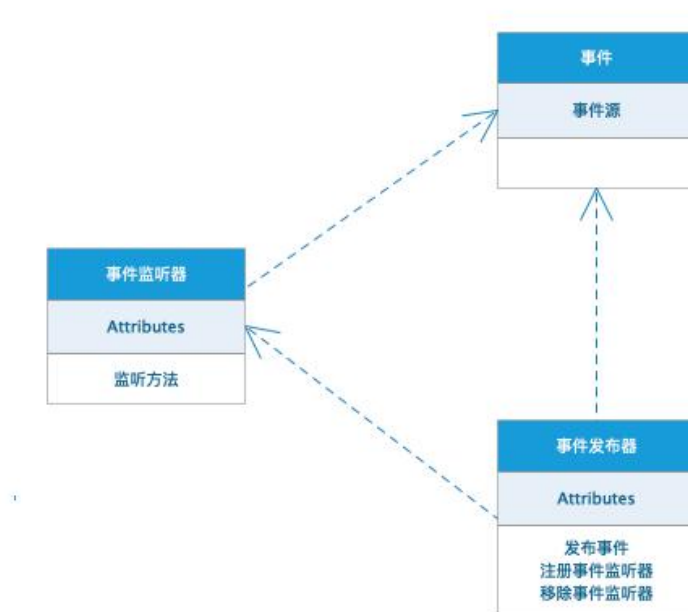
之 Spring 事件发布监听机制

问题描述

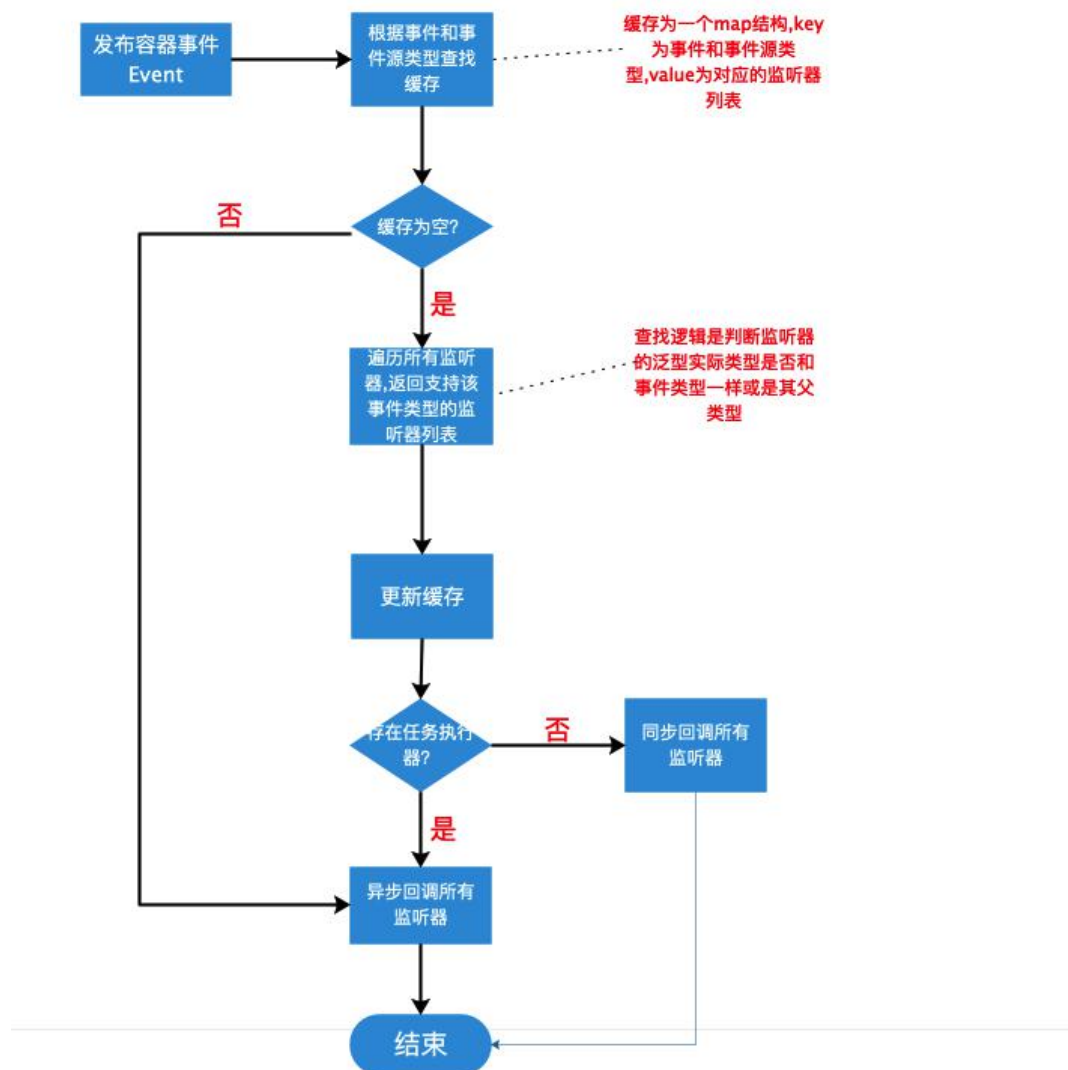
Spring Boot	2.2.5
JDK	8

Spring 事件发布监听机制由以下 Java 类实现：

1. ApplicationEvent Packet
2. EventListener Endpoint
3. ApplicationEventMulticaster Router



解决



□

Spring Boot

之 Spring 配置静态资源服务

问题描述

Spring Boot	2.2.5
JDK	8

Spring 框架提供内置的 Tomcat 服务器，具备提供静态资源服务的能力。这篇笔记记录了通过 application.properties 配置文件启用静态资源服务的实现

```
spring.resources.static-locations
- 默认值
  - classpath:/META-INF/resources/
  - classpath:/resources/
  - classpath:/static/
  - classpath:/public/
- 解释
  - Locations of static resources.
spring.mvc.static-path-pattern
- 默认值
  - /**
- 解释
  - Path pattern used for static resources.
```

解决

application.properties

```
spring.resources.static-locations=file:/${directory}/
spring.mvc.static-path-pattern=${directory}/**
```

□

Spring Boot

之使用 Maven 快速搭建 Spring Boot 框架

问题描述

Spring Boot	2.2.5
JDK	8

Spring 框架是 MVC（模型-视图-控制器）、IoC（控制反转）、AOP（面向切面编程）、Security（安全）等一系列特性组成的框架。boot 作为动词有“to become ready for use especially by booting a program”的含义，即通过配置文件快速启动 Spring 项目。与 Open Stack 的 Blueprint、Ansible 的 Playbook 一样，Spring Boot 通过配置文件省去了编码的烦恼。

解决

pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.5.RELEASE</version>
    </parent>

    <groupId></groupId>
    <artifactId></artifactId>
    <version></version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
            <version>2.2.5.RELEASE</version>
        </dependency>
    </dependencies>
</project>

```

REFERENCE

<https://docs.spring.io/spring-boot/docs/2.2.5.RELEASE/reference/html/using-spring-boot.html#using-boot-maven>



Spring Boot

之修改上传文件大小上限

问题描述

Spring Boot	2.2.5
JDK	8

`spring.servlet.multipart.max-file-size`

- 默认值
 - 1MB
- 描述
 - Max file size.

`spring.servlet.multipart.max-request-size`

- 默认值
 - 10MB
- 描述
 - Max request size.

解决

`application.properties`

```
spring.servlet.multipart.max-file-size=15MB
spring.servlet.multipart.max-request-size=15MB
```



Spring Boot

之函数式接口@FunctionalInterface

问题描述

Spring Boot	2.2.5
JDK	8

@FunctionalInterface 是 JDK 中的接口。顾名思义，“函数式接口”是遵循“函数式编程”规范的接口，即在接口中仅实现一个抽象方法。函数式接口的意义在于可以使用一个类似于 Lambda 表达式的“语法糖”

.....

解决

Runnable 接口使用@FunctionalInterface 注解

@FunctionalInterface

```
public interface Runnable {  
    public abstract void run();  
}
```

Thread 实现 Runnable 接口，使用语法糖实例化接口

```
new Thread(  
    ()->{  
        System.out.println(Thread.currentThread().getName());  
    }  
).start();
```

编译器将语法糖解析为如下表示：

```
new Thread(  
    new Runnable() {  
        @Override  
        public void run() {  
            System.out.println(Thread.currentThread().getName());  
        }  
    }  
).start();
```

□

Spring Boot

之序列化除外关键字 transient

问题描述

Spring Boot	2.2.5
JDK	8

Java POJO 序列化、反序列化功能可由以下接口实现：

1. Serializable
2. Externalizable

POJO 中以下对象属性不会被序列化、反序列化到二进制流中：

1. 被关键字 **static** 修饰的静态域
2. 被关键字 **transient** 修饰的

敏感信息如银行卡密码等需要使用关键字 **transient** 修饰，使其从二进制流中除外

解决

```
public class KeywordTransient {

    @Getter @Setter @AllArgsConstructor @NoArgsConstructor
    static class Apple implements Serializable {
        Integer memory;

        Integer storage;

        transient String password;
    }

    public static void main(String[] args) {
        Apple apple = new Apple(64, 128, "never get public");
        ObjectOutputStream stream = null;
        try {
            stream = new ObjectOutputStream(new FileOutputStream(new
File("src/main/resources/apple.persistence")));
            stream.writeObject(apple);
            stream.flush();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                stream.close();
            } catch (IOException e) {
                // swallow the exception.
            }
        }

        try {
            ObjectInputStream inputStream = new ObjectInputStream(new FileInputStream(new
File("src/main/resources/apple.persistence")));
            Apple appleDeserialized = (Apple) inputStream.readObject();
        }
```

```

        System.out.println("pause");
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
}

```

调试器截图：



□

Spring Boot

之快速使用 Spring IoC 容器

问题描述

Spring Boot	2.2.5
JDK	8

在程序设计语言发展的幼年时期，人们认为能够从实例获取类就足够了。直到组件化编程的出现，接口实现对接口的依赖造成了诸多不便，于是，从类获取实例也变得十分必要。基于这种需求，出现了新的语言特性——反射。在反射特性的基础上，出现了 IoC（控制反转）容器技术。

可以将 Java Bean 注册到 Application Context（看作一张索引表）中，在 Controller 中通过 id 获取 Bean 实例，从而搭建起连接 Model 层和 Controller 层的桥梁。Spring 框架需要，但是缺少一个通用的 Application Context 工具类，在这篇笔记中摘录一个比较好用的实现。

在 Application 的 start 方法中读取 ctx 配置文件并加载应用上下文至工具类中，即可实现 IoC 容器的快速使用

解决

SpringBeanFactory.java

```
public class SpringBeanFactory {
    private static final SpringBeanFactory instance = new SpringBeanFactory();
    private ApplicationContext ctx = null;

    protected ApplicationContext getCtx() {
        return ctx;
    }

    /**
     * 私有构造函数
     */
    private SpringBeanFactory()
    {

    }

    /**
     * 获取单实例
     * @return
     */
    public static SpringBeanFactory getInstance()
    {
        return instance;
    }

    /**
     * SPRING 是否初始化完成
     * @return
     */
    public static boolean isContextReady(){
        return getInstance().getCtx()!=null;
    }
}
```

```
*****
    * 获取指定 ID 的 Bean 实例
    * @param id Spring Bean 的唯一 ID 串
    * @return Bean 实例
    */
    @SuppressWarnings("unchecked")
    public static <T> T getSpringBean(String id)
    {
        return (T) getInstance().getBean(id);
    }

    /**
    * 获取指定 ID 和类型的 Bean 实例列表
    * @param type Spring Bean 的唯一 ID 串
    * @return Bean 实例
    */
    public static <T> List<T> getSpringBeans(Class<T> type)
    {
        return getInstance().getBean(type);
    }

    /**
    * 初始化 ApplicationContext
    * @param ctx 上下文实例
    */
    public synchronized static void setApplicationContext(ApplicationContext ctx)
    {
        getInstance().setContext(ctx);
    }

    /**
    * 初始化 ApplicationContext
    * @param ctx 上下文实例
    */
    private void setContext(ApplicationContext ctx)
    {
        this.ctx = ctx;
    }

    /**
    * 从 ApplicationContext 中获取 Bean
    * @param id Spring Bean 的唯一 ID 串
    * @return Bean 实例
    */
    private Object getBean(String id)
```



```

    {
        return ctx.getBean(id);
    }

    /**
     * 从 ApplicationContext 中获取 Bean
     * @param type Spring Bean 的唯一 ID 串
     * @return Bean 实例
     */
    private <T> List<T> getBean(Class<T> type)
    {
        String[] beanNames = ctx.getBeanNamesForType(type);
        if (beanNames == null)
        {
            return new ArrayList<T>();
        }
        List<T> beans = new ArrayList<T>();
        for (String beanName : beanNames)
        {
            beans.add(ctx.getBean(beanName, type));
        }
        return beans;
    }
}

```

□

Spring Boot

之数据库版本控制工具 Liquibase

问题描述

Spring Boot	2.2.5
JDK	8

Spring Boot “老三样” 包括：

- Controller
- Service
- DAO

三部分组件的设计均遵循 Spring 全家桶 “约定 > 配置 > 编码” 的设计理念。

.....

数据库的建表操作可以纳入这一理念中来吗？

可以的。一种落地实现即为数据库版本控制工具 Liquibase

解决

1. 引入 Maven 依赖项

```
<dependency>
  <groupId>org.liquibase</groupId>
  <artifactId>liquibase-core</artifactId>
</dependency>
```

2. 声明 Bean

```
<bean id="liquibase" class="liquibase.integration.spring.SpringLiquibase">
  <property name="dataSource" ref="dataSource" />
  <property name="changeLog"
value="classpath:db-ups-create-table-changelog.xml" />

  <!--
  contexts specifies the runtime contexts to use.
  -->
  <property name="contexts" value="test, production" />
</bean>
```

3. 转写数据库配置文件

如果已有数据库建表脚本，需要转写为 Liquibase 格式的 xml 定义文件

笔记

1. SpringLiquibaseBean 实现了 InitializingBean 接口。会在初始化 Bean 阶段调用其中的 afterPropertiesSet() 方法，初始化数据库

2. Liquibase 会在数据库中新建两张表

- 版本控制
- 同步锁

□

Spring Boot

之重构依赖项最佳实践

问题描述

Spring Boot	2.2.5
JDK	8

注入常用 *class* 的帮助类, 把常用的几种放一起, 子类就不用总注入那么多类了

解决

```
<bean id="classInjectionHelper"
    class=""
    abstract="true">
    <property name="" ref="" />
</bean>

public abstract class ClassInjectionHelper {
    private PerfObjectDao perfObjectDao;

    public PerfObjectDao getPerfObjectDao() {
        return perfObjectDao;
    }

    public void setPerfObjectDao(PerfObjectDao perfObjectDao)
    {
        this.perfObjectDao = perfObjectDao;
    }
}

<bean id=""
    class=""
    parent="classInjectionHelper">
    <property name="" ref="" />
</bean>

public class UninstallPerfMetadataManager extends ClassInjectionHelper {
    private JoinPerfMetadataManager joinPerfMetadataManager;

    public void setJoinPerfMetadataManager(JoinPerfMetadataManager joinPerfMetadataManager)
```

```

    {
        this.joinPerfMetadataManager = joinPerfMetadataManager;
    }
}

```

□

Spring Boot

之面向切面编程（AOP） -Profiling

问题描述

Spring Boot	2.2.5
JDK	8

面向切面编程（AOP）可用于性能统计（Profiling）

解决

```

@Aspect
@Component
public class ProfilingAspect {

    private ProfRepo repo = ProfRepo.getInstance();

    @Pointcut("@annotation(com.bunny.aop.annotation.Timed)")
    public void profiling() {

    }

    @Around("profiling()")
    public Object doProfiling(ProceedingJoinPoint joinPoint) throws Throwable {
        long start = System.nanoTime();
        try {
            return joinPoint.proceed();
        } finally {
            long end = System.nanoTime();
            System.out.println(joinPoint.getSignature().toShortString() + " takes " + (end - start) + "nanoseconds");
            repo.add(joinPoint.getSignature().toShortString() + " takes " + (end - start) +

```

```
        "nanoseconds");  
    }  
}  
}
```



Spring Boot

之面向切面编程（AOP） -Tracing

问题描述

Spring Boot	2.2.5
JDK	8

面向切面编程（AOP）可用于代码追踪（Tracing）

解决

```
@Aspect  
@Component  
public class TraceAspect {  
  
    private int level = 0;  
  
    @Pointcut("within(com.bunny.aop.*)")  
    public void trace() {  
  
    }  
  
    @Before("trace()")  
    public void doTrace(JoinPoint point) {  
        for (int i = 0; i < level; i++) {
```

```

        System.out.print("\t");
    }
    System.out.println(point.getSignature() + Arrays.toString(point.getArgs()));
    level++;
}

@After("trace()")
public void doneTrace(JoinPoint point) {
    level--;
}
}

```

效果预览

```

2020-05-22 14:26:18.325 INFO 11068 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 15 ms
String com.bunny.aop.controller.FeedAnimalController.query() []
void com.bunny.aop.service.FeedAnimalService.query() []
void com.bunny.aop.worker.Pot.queryAnimal() []

```

笔记

15. 极大地方便了 legacy code（陈旧代码）的维护工作：使用 IDE 进行静态分析费时费力。Tracing 动态打印调用栈清晰明了地观察程序的执行逻辑。通过调整函数入参，观察调用栈的变化，方便了解函数功能

16. within: Limits matching to join points within certain types (the execution of a method declared within a matching type when using Spring AOP).

□

Spring Boot

之面向切面编程（AOP）

问题描述

Spring Boot	2.2.5
JDK	8

一些看上去不可思议的特性往往与 JVM 有着紧密的联系，比如重量级锁 `synchronized`、又如这边笔记中提及的面向切面编程 AOP

在 Spring Boot 项目中使用 AOP 功能的配置是“老三样儿”：

- ✓ 添加 Maven 依赖
- ✓ 通过注解开启功能
- ✓ 定义切面

解决

18. 添加 Maven 依赖

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
  <version>2.3.0.RELEASE</version>
</dependency>

<dependency>
  <groupId>org.aspectj</groupId>
  <artifactId>aspectjweaver</artifactId>
  <version>1.9.5</version>
</dependency>
```

19. 通过注解开启功能

```
@Configuration
@EnableAspectJAutoProxy
@ComponentScan
public class ApplicationConfig {
}
```

20. 定义切面

```
@Aspect
@Component
public class LogExecutor {
    @Around("execution(public void com.bunny.aop.worker.Pot.feedAnimal())")
    public Object log(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
        System.out.println("before feed animal");
        return proceedingJoinPoint.proceed();
    }
}
```

笔记

17. 切面类需要注册成 Java Bean

18. AspectJ's dynamic crosscutting support comes in the form of advice. Advice is the code executed at a join point selected by a pointcut. Advice can execute before, after, or around the join point. The body of advice is much like a method body—it encapsulates the logic to be executed upon reaching a join point.

19. Spring AOP 应用场景

- (1) 打印日志
- (2) 调优：打印方法执行时间
- (3) 鉴权

□

Spring Boot

之面向切面编程（AOP）-事务管理 背景

问题描述

Spring Boot	2.2.5
JDK	8

接下来的一篇技术笔记将讨论利用 AOP 特性实现事务管理注解 `@Transactional`。这篇笔记将对 `@Transactional` 注解做一个简单的介绍。

解决

21. 建表并导入测试数据

```
/*
Navicat MySQL Data Transfer

Source Server          : localdb
Source Server Version  : 50727
Source Host            : localhost:3306
Source Database        : transactional
```



```
Target Server Type      : MYSQL
Target Server Version : 50727
File Encoding           : 65001
```

Date: 2020-05-22 08:19:53

*/

SET FOREIGN_KEY_CHECKS=0;

-- Table structure for products

```
DROP TABLE IF EXISTS `products`;
CREATE TABLE `products` (
  `id` int(11) NOT NULL,
  `name` varchar(255) NOT NULL,
  `description` varchar(255) DEFAULT NULL,
  `price` decimal(10,2) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

-- Records of products

```
INSERT INTO `products` VALUES ('1001', 'ProductA', 'Product A description', '1.00');
INSERT INTO `products` VALUES ('1002', 'ProductB', 'Product B description', '2.00');
INSERT INTO `products` VALUES ('1003', 'ProductC', 'Product C description', '3.00');
INSERT INTO `products` VALUES ('1004', 'ProductD', 'Product D description', '4.00');
INSERT INTO `products` VALUES ('1005', 'ProductE', 'Product E description', '5.00');
```

22. 添加@Transactional 注解

```
@Transactional(transactionManager = "getTx")
public void queryAnimal() {
//    可以正常执行
    jdbcTemplate.execute("insert into products (id, name, description, price)
values(1006, 'ProductE', 'Product E description', 5)");
//    抛出异常 Duplicate entry '1005' for key 'PRIMARY'
    jdbcTemplate.execute("insert into products (id, name, description, price)
values(1005, 'ProductF', 'Product F description', 6)");
}
```

23. 观察结果

```
java.sql.SQLIntegrityConstraintViolationException: Duplicate entry '1005' for key 'PRIMARY'
```

打印的异常堆栈从顶到底大致分为三块：

```
{ jdbc }
{ 动态代理}
{ Tomcat }
```

笔记

20. Transactional 注解加在直接调用的方法上面才可以，否则不会回滚。原因在于 Spring Framework 会过滤@Transactional 注解，并自动生成代理类。只有调用到代理类中的相应方法才会进入事务（Transaction）

□

Spring Boot

之面向切面编程（AOP）-事务管理

问题描述

Spring Boot	2.2.5
JDK	8

Spring Framework 中的面向切面编程（AOP）特性可以打破传统面向过程、面向对象程序设计的代码执行流程，实现一些“神奇”的功能。事务管理是应用场景之一。

解决

24. 定义@Transactional 注解

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface Transactional {
    @AliasFor("transactionManager")
    String value() default "";

    @AliasFor("value")
```

```

String transactionManager() default "";

Propagation propagation() default Propagation.REQUIRED;

Isolation isolation() default Isolation.DEFAULT;

int timeout() default -1;

boolean readOnly() default false;

Class<? extends Throwable>[] rollbackFor() default {};

String[] rollbackForClassName() default {};

Class<? extends Throwable>[] noRollbackFor() default {};

String[] noRollbackForClassName() default {};
}

```

自定义注解是为了剥离 Spring Tx 自带的事务管理器行为。为了方便，照抄 Spring Tx 中的同名注解@Transactional。

25. 实现 TransactionDefinition 接口

```

public class TransactionAttributeWithRollbackRules extends DefaultTransactionAttribute {
    Collection<Class<? extends Throwable>> rollbackFor = new ArrayList<Class<? extends
Throwable>>();
    Collection<Class<? extends Throwable>> noRollbackFor = new ArrayList<Class<? extends
Throwable>>();

    public void addRollbackFor(Class<? extends Throwable>[] rollbackFor) {
        Collections.addAll(this.rollbackFor, rollbackFor);
    }

    public void addNoRollbackFor(Class<? extends Throwable>[] noRollbackFor) {
        Collections.addAll(this.noRollbackFor, noRollbackFor);
    }

    @Override
    public boolean rollbackOn(Throwable ex) {
        if (ex instanceof RuntimeException || ex instanceof Error) {
            for (Class<? extends Throwable> t : this.noRollbackFor) {
                if (t.isAssignableFrom(ex.getClass())) {
                    return false;
                }
            }
        }
    }
}

```

```

    }
    return true;
} else {
    for (Class<? extends Throwable> t : this.rollbackFor) {
        if (t.isAssignableFrom(ex.getClass())) {
            return true;
        }
    }
    return false;
}
}
}
}

```

26. 定义切面

```

@Aspect
public abstract class AbstractTransactionManagementAspect {

    @Autowired
    private TransactionManager transactionManager;

    @Pointcut
    protected abstract void transactionalOp();

    public abstract TransactionAttributeWithRollbackRules getTransactionAttribute(
        JoinPoint jp);

    @Around("transactionalOp()")
    public Object transact(ProceedingJoinPoint pjp) throws Throwable {
        TransactionAttributeWithRollbackRules txAttribute = getTransactionAttribute(pjp);
        TransactionStatus status = ((DataSourceTransactionManager) transactionManager)
            .getTransaction(txAttribute);
        try {
            Object ret = pjp.proceed();
            ((DataSourceTransactionManager) transactionManager).commit(status);
            return ret;
        } catch (Throwable ex) {
            if (txAttribute.rollbackOn(ex)) {
                ((DataSourceTransactionManager) transactionManager).rollback(status);
            } else {
                ((DataSourceTransactionManager) transactionManager).commit(status);
            }
        }
        throw ex;
    }
}

```

```
}  
}  
}
```

笔记

1. Then, by using the TransactionDefinition and TransactionStatus objects, you can initiate transactions, roll back, and commit.
2. 切面的定义同样遵循面向对象程序设计的继承特性
3. 并发场景下的@Transactional 如何实现
4. 事务管理器是由 Spring 框架实现的，而不是数据库的一部分逻辑
5. AOP 核心概念
 - (1) Aspect
 - (2) Point cut Expression（九种）
 - (3) Point cut Signature
 - (4) Advice（五种）
6. 完整代码：

<https://github.com/MariaLikesFish/spring-aop-transaction>

□

Spring Boot

之面向切面编程（AOP）-失败重试

问题描述

Spring Boot	2.2.5
JDK	8

面向切面编程（AOP）特性是 Spring 框架的一大“脑洞”。AOP 的设计理念超出了传统的面向过程、面向对象程序设计方法遵循的代码执行流程。正因如此，AOP 可以在一些应用场景下显示出“奇效”

依赖于技术笔记《面向切面编程（AOP）》

解决

LogExecutor.java

```

@Aspect
@Component
public class LogExecutor {

    private int maxRetries = 5;

    @Around("execution(public void com.bunny.aop.worker.Pot.feedAnimal()) || execution(public void com.bunny.aop.worker.Pot.petAnimal())")
    public Object log(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
        int numAttempts = 0;
        Exception exception;
        do {
            numAttempts++;
            System.out.println("trying " + numAttempts + " times");
            try {
                return proceedingJoinPoint.proceed();
            }
            catch(Exception ex) {
                exception = ex;
            }
        } while(numAttempts <= this.maxRetries);
        throw exception;
    }
}

```

笔记

21. 示例改编自 Spring 官方文档
22. 在@Around 注解中使用 “||” 可以将一个 Advice 连接至多个 Pin Point
23. 适用于并发场景

□

Spring Boot

之面向切面编程（AOP）-读写锁

问题描述

Spring Boot	2.2.5
JDK	8

面向切面编程打破了面向过程（函数式）、面向对象程序设计的代码流程，在一些场景下可以发挥意向不到的妙用。例如：

1. 失败重试
2. 事务管理
3. 【读写锁】

心得 面向切面编程类似于虚拟化中的“横向扩展”，可以起到非侵入式改造、代码去重的效果

解决

AbstractReadWriteLockAspect.java

```
@Aspect
public abstract class AbstractReadWriteLockAspect {

    private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

    @Pointcut
    protected abstract void read();

    @Pointcut
    protected abstract void write();

    @Around("read()")
    public void doRead(ProceedingJoinPoint joinPoint) {
        try {
            lock.readLock().lock();
            joinPoint.proceed();
        } catch (Throwable throwable) {
            throwable.printStackTrace();
        } finally {
            lock.readLock().unlock();
        }
    }
}
```

```
@Around("write()")
public void doWrite(ProceedingJoinPoint joinPoint) {
    try {
        lock.writeLock().lock();
        joinPoint.proceed();
    } catch (Throwable throwable) {
        throwable.printStackTrace();
    } finally {
        lock.writeLock().unlock();
    }
}
```

笔记

24. 面向切面编程 “三件套”

(1) 新建注解

(2) 定义切面

① Aspect

② Pin point

③ Advice

(3) 添加注解

25. @target: Limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type.

26. 完成项目地址:

<https://github.com/MariaLikesFish/spring-aop-read-write-lock>

□

Spring Boot

之配置 Druid 高效数据库连接池

问题描述

Spring Boot

2.2.5

JDBC 操作数据库分四步：

- 建立连接
- 执行事务
- 关闭连接
- 释放连接

其中只有【执行事务】为必要开销，其余三步均为 **overhead**。为了解决开销，数据库连接池（中间件）应运而生。

常见的数据连接池包括：

- DBCP
- c3p0
- Druid

其中效率最高的属 Alibaba 捐赠的 Apache Druid 项目，所以又称高效数据库连接池

解决

4. 引入 Maven 依赖项

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.22</version>
</dependency>

<dependency>
  <groupId>com.sybase.jdbc</groupId>
  <artifactId>jconn</artifactId>
</dependency>
```

5. 声明 Data Source

druid-ds.properties

```
jdbc_url=jdbc:sybase:Tds:10.74.170.6:3639?ServiceName=pmsdboes
jdbc_user=dis
jdbc_password=dis
```

druid-ds-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```

.....
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
https://www.springframework.org/schema/context/spring-context.xsd">

    <context:property-placeholder location="classpath:druid-ds.properties"/>

    <bean      id="dataSource"      class="com.alibaba.druid.pool.DruidDataSource"
init-method="init" destroy-method="close">
        <property name="driverClassName" value="com.sybase.jdbc3.jdbc.SybDriver"/>
        <property name="url" value="{jdbc_url}" />
        <property name="username" value="{jdbc_user}" />
        <property name="password" value="{jdbc_password}" />

        <property name="filters" value="stat" />

        <property name="maxActive" value="20" />
        <property name="initialSize" value="1" />
        <property name="maxWait" value="60000" />
        <property name="minIdle" value="1" />

        <property name="timeBetweenEvictionRunsMillis" value="60000" />
        <property name="minEvictableIdleTimeMillis" value="300000" />

        <property name="testWhileIdle" value="true" />
        <property name="testOnBorrow" value="false" />
        <property name="testOnReturn" value="false" />

        <property name="poolPreparedStatements" value="true" />
        <property name="maxOpenPreparedStatements" value="20" />

        <property name="asyncInit" value="true" />
    </bean>
</beans>

```

笔记

1. jconn 为 Sybase IQ 数据库驱动.

Spring Cloud

Spring Cloud

之客户端负载均衡 Ribbon

问题描述

Spring Boot	2.2.5
Spring Cloud	Hoxton SR4
JDK	14

Ribbon 是 Spring Cloud 全家桶中的一个中间件，特点包括：

- 软负载均衡（相对于硬件实现的负载均衡，如 F5）
- 客户端负载均衡

解决

Ribbon 配置：

1. 导入 Maven 依赖

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
  <version>2.2.2.RELEASE</version>
</dependency>
```

2. 配置 application.yml

```
spring:
  application:
    name: spring-cloud-ribbon

server:
  port: 8888

ping-server:
  ribbon:
    eureka:
      enabled: false
    listOfServers: localhost:9092,localhost:9999
    ServerListRefreshInterval: 15000
```

3. 开启注解

```
.....  
@SpringBootApplication  
@RestController  
@RibbonClient(  
name = "ping-a-server",  
configuration = RibbonConfiguration.class)  
public class ServerLocationApp {  
  
    @LoadBalanced  
    @Bean  
    RestTemplate getRestTemplate() {  
        return new RestTemplate();  
    }  
  
    @Autowired  
    RestTemplate restTemplate;  
  
    @RequestMapping("/server-location")  
    public String serverLocation() {  
        return this.restTemplate.getForObject(  
            "http://ping-server/locaus", String.class);  
    }  
  
    public static void main(String[] args) {  
        SpringApplication.run(ServerLocationApp.class, args);  
    }  
}
```

笔记

1. SC Ribbon 依赖于 SC Eureka
2. Ribbon 对应注解@LoadBalanced



Spring Cloud

之服务注册中心 Eureka

问题描述

Spring Boot	2.2.5
Spring Cloud	Hoxton SR4
JDK	14

Spring 全家桶提供了 J2EE 的一站式替代方案，包括 Spring Framework、Spring Cloud、Spring Cloud Data Flow 三个品类，分别对应 Java Web 微服务场景、云原生（Cloud Native）场景、大数据场景

“负载均衡”是“高可用”理念的一种落地实现（其它实现包括服务降级等），比较流行的解决方案有以下几种：

1. Nginx
 - (1) 老牌 web 服务器
 - (2) 稳定、高效
2. Dubbo / Zookeeper
 - (1) 中间件
 - (2) 一个成熟的分布式互联网架构
 - (3) 基于 RPC（远程过程调用）
3. SC(Spring Cloud) Eureka
 - (1) 中间件
 - (2) 新兴架构
 - (3) 基于 REST 接口

SC Eureka 项目经历了反复的闭源、开源调整。可用 SC Consul 代替？

Spring Boot 的理念是“约定大于配置”

解决

Eureka Server 配置：

4. 导入 Maven 依赖

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
  <version>2.2.2.RELEASE</version>
</dependency>
```

5. 配置 application.yml

```
server:
  port: 8081 #服务注册中心端口号
```

```
eureka:
  instance:
    hostname: 127.0.0.1 #服务注册中心 IP 地址
  client:
    registerWithEureka: false #是否向服务注册中心注册自己
    fetchRegistry: false #是否检索服务
    serviceUrl: #服务注册中心的配置内容，指定服务注册中心的位置
    defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

6. 开启注解

```
@EnableEurekaServer
@SpringBootApplication
public class ServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ServerApplication.class);
    }
}
```

Eureka Provider 配置:

1. 导入 Maven 依赖

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
  <version>2.2.2.RELEASE</version>
</dependency>
```

2. 配置 application.yml

```
eureka:
  client:
    serviceUrl: #注册中心的注册地址
    defaultZone: http://127.0.0.1:8081/eureka/
  server:
    port: 8082 #服务端口号
  spring:
    application:
      name: service-provider #服务名称--调用的时候根据名称来调用该服务的方法
```

3. 开启注解

```
@EnableEurekaClient
@SpringBootApplication
public class StarterApplication {
    public static void main(String[] args) {
        SpringApplication.run(StarterApplication.class);
    }
}
```

Eureka Consumer 配置:

1. 添加 Maven 依赖

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
  <version>2.2.2.RELEASE</version>
</dependency>
```

2. 配置 application.yml

```
eureka:
  client:
    serviceUrl: #注册中心的注册地址
    defaultZone: http://127.0.0.1:8081/eureka/
  server:
    port: 8080 #服务端口号
  spring:
    application:
      name: service-consumer #服务名称--调用的时候根据名称来调用该服务的方法
```

3. 开启注解

```
@EnableEurekaClient
@SpringBootApplication
public class ConsumerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class);
    }
}
```

4. 使用 Rest Template 调用接口

```
@Controller
public class LoadBalanceController {

    @Autowired
    private RestTemplate restTemplate;

    @Bean
    public RestTemplate rest() {
        return new RestTemplate();
    }

    @RequestMapping(path = "/rest/query", method = RequestMethod.GET, produces = "application/json")
    @ResponseBody
    public String queryAnimal() {
        return restTemplate.getForObject("http://service-provider/rest/query", String.class);
    }
}
```

}

笔记

1. 配置 Eureka 集群以确保高可用
2. 使用 Ribbon 配置负载均衡



Spring Cloud

之配置中心 Config

问题描述

Spring Boot	2.2.5
Spring Cloud	Hoxton SR4
JDK	14

SC Config 是 Spring Cloud 全家桶提供的配置中心中间件

SC Config 配置文件可以存放在：

- Git 服务器（如 GitHub）
- 文件系统
- 内存

解决

SC Config Server 配置：

7. 导入 Maven 依赖

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
  <version>2.2.2.RELEASE</version>
</dependency>
```

8. 配置 application.yml


```

server:
  port: 8888

spring:
  application:
    name: microservice-cloud-config
  cloud:
    config:
      server:
        git:
          uri: https://github.com/MariaLikesFish/sc-config.git

```

9. 开启注解

```

@EnableConfigServer
@SpringBootApplication
public class ConfigServer {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServer.class, args);
    }
}

```

SC Config Client 配置:

4. 导入 Maven 依赖

```

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-client</artifactId>
  <version>2.2.2.RELEASE</version>
</dependency>

```

5. 配置 application.yml

```

spring:
  cloud:
    config:
      label: master
      name: app1 #需要从 github 上读取的资源名称，注意没有 yml 后缀名
      profile: test #本次访问的配置项
      uri: http://localhost:8888 #本微服务启动后先去找 3344 号服务，通过
SpringCloudConfig 获取 GitHub 的服务地址

```

6. 开启注解

```

@Getter
@Setter
@Service
public class AnimalService {
    @Value("${server.port}")
    private String port;
}

```

```
@Value("${server.id}")
private String id;

}
```

笔记

- 3. 与 SC Bus 联合使用，可实现“热部署”
 - 4. 总结 Spring Cloud 中间件配置的一般流程：
 - (1) 添加 Maven 依赖项
 - (2) 配置 application.yml
 - (3) 开启注解
- 这正印证了 Spring Cloud “约定 > 配置 > 编码”的设计理念

□

Spring Framework 源码阅读

Spring Framework 源码阅读

之 Application 初始化：ApplicationRunner

问题描述

Spring Boot	2.2.5
Spring Framework	5.2.6
JDK	1.8

Spring Boot 的设计理念为“约定优于配置”。这也给 Spring 框架染上了许多神秘主义色彩。

比如，在 SpringApplication 启动阶段，会调用实现了 ApplicationRunner、CommandLineRunner 接口的组件（@Component）钩子，完成应用启动后的自定义逻辑。

.....

解决

```
@Component
public class ThirdPartyRunner implements ApplicationRunner {
    @Override
    public void run(ApplicationArguments args) throws Exception {
        System.out.println("application hook triggered.");
    }
}
```

□

Spring Framework 源码阅读

之 Application 初始化：Spring Factories

问题描述

Spring Boot	2.2.5
Spring Framework	5.2.6
JDK	1.8

Spring Boot、Spring Framework IoC 容器实现逻辑复杂，其中相当大一部分工作在解耦合。

Spring Factories 机制就是 Spring “解耦合”设计理念的一个落地实现。Spring Factories 设计思路借鉴了 Java SPI（Service Provider Interface）。

解决

Spring Boot 应用初始化阶段，会加载 Spring Factories，摘录关键代码如下：

```

private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader classLoader)
{
    MultiValueMap<String, String> result = cache.get(classLoader);
    if (result != null) {
        return result;
    }

    try {
        Enumeration<URL> urls = (classLoader != null ?
            classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
            ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
        result = new LinkedMultiValueMap<>();
        while (urls.hasMoreElements()) {
            URL url = urls.nextElement();
            UrlResource resource = new UrlResource(url);
            Properties properties = PropertiesLoaderUtils.loadProperties(resource);
            for (Map.Entry<?, ?> entry : properties.entrySet()) {
                String factoryTypeName = ((String) entry.getKey()).trim();
                for (String factoryImplementationName :
StringUtils.commaDelimitedListToStringArray((String) entry.getValue())) {
                    result.add(factoryTypeName, factoryImplementationName.trim());
                }
            }
        }
        cache.put(classLoader, result);
        return result;
    }
    catch (IOException ex) {
        throw new IllegalArgumentException("Unable to load factories from location [" +
            FACTORIES_RESOURCE_LOCATION + "]", ex);
    }
}

```

默认的加载路径为类路径"META-INF/spring.factories"。类似于 ATG 配置文件的加载方式，spring.factories 配置文件采用叠加的方式进行加载。但是，由于采用 MultiValueMap 实现，同名工厂之间不会相互覆盖。

默认的 spring.factories 文件位于
org/springframework/boot/spring-boot/2.3.0.RELEASE/spring-boot-2.3.0.RELEASE.jar!/META-INF/spring.factories

□

Spring Framework 源码阅读

之 Application 初始化：代码逻辑

问题描述

Spring Boot 2.2.5
Spring Framework 5.2.6
JDK 1.8

SpringApplication.run(..)依次完成以下步骤：

1. 初始化阶段
 - (1) 读取 spring.factories
2. 运行阶段
 - (1) 配置 Environment：读取 application.yml
 - (2) createApplicationContext(..)：初始化应用上下文
 - (3) prepareContext(..)：解析 JavaConfig、XML 为 BeanDefinition
 - (4) refreshContext(..)：装配 Java Bean
 - (5) afterRefresh(..)：自定义扩展
 - (6) callRunners(..)：应用启动后扩展
 - (7) 上述步骤间，穿插事件发布监听机制（另文介绍）

解决

```
@SpringBootApplication
public class App {
    public static void main(String[] args) {
        SpringApplication.run(App.class);
    }
}
```

□

Spring Framework 源码阅读

之 Application 初始化：获取 ClassLoader

问题描述

Spring Boot	2.2.5
Spring Framework	5.2.6
JDK	1.8

SpringBoot 提供了一个工具类 `SpringApplication`，用于傻瓜式拉起 Spring Boot 应用。

在初始化阶段，有一步获取 `ClassLoader` 的源码。还没有找到应用场景，但这段逻辑写得机巧，在这篇笔记中摘录下来

Java 中类加载器（`ClassLoader`）的作用是维护类名到它相应的二进制字节码文件的映射关系，进而加载系统资源（`SystemResource`）

使用 `ClassLoader::getSystemClassLoader` 方法获得类加载器（APP 内部类）

在 web 应用中，则应获取 Tomcat 内含类加载器，即使用 `Thread::getContextClassLoader` 方法

解决

```
@Nullable
public static ClassLoader getDefaultClassLoader() {
    ClassLoader cl = null;
    try {
        cl = Thread.currentThread().getContextClassLoader();
    }
```

```

    }
    catch (Throwable ex) {
        // Cannot access thread context ClassLoader - falling back...
    }
    if (cl == null) {
        // No thread context class loader -> use class loader of this class.
        cl = ClassUtils.class.getClassLoader();
        if (cl == null) {
            // getClassLoader() returning null indicates the bootstrap ClassLoader
            try {
                cl = ClassLoader.getSystemClassLoader();
            }
            catch (Throwable ex) {
                // Cannot access system ClassLoader - oh well, maybe the caller can live with null...
            }
        }
    }
    return cl;
}

```

□

Spring Framework 源码阅读

之 Bean 加载：加载流程

问题描述

Spring Boot	2.2.5
Spring Framework	5.2.6
JDK	1.8

Spring Bean 的加载流程是深入理解 Spring IoC 容器的一个基础问题。

Spring Bean 的加载流程主要包含以下步骤:

- ① Spring 启动, 查找并加载需要被 Spring 管理的 bean, 进行 Bean 的实例化
- ② Bean 实例化后对将 Bean 的引入和值注入到 Bean 的属性中
- ③ 如果 Bean 实现了 BeanNameAware 接口的话, Spring 将 Bean 的 Id 传递给 setName()方法
- ④ 如果 Bean 实现了 BeanFactoryAware 接口的话, Spring 将调用 setBeanFactory()方法, 将 BeanFactory 容器实例传入
- ⑤ 如果 Bean 实现了 ApplicationContextAware 接口的话, Spring 将调用 Bean 的 setApplicationContext()方法, 将 bean 所在应用上下文引用传入进来。
- ⑥ 如果 Bean 实现了 BeanPostProcessor 接口, Spring 就将调用他们的 postProcessBeforeInitialization()方法。
- ⑦ 如果 Bean 实现了 InitializingBean 接口, Spring 将调用他们的 afterPropertiesSet()方法。类似的, 如果 bean 使用 init-method 声明了初始化方法, 该方法也会被调用
- ⑧ 如果 Bean 实现了 BeanPostProcessor 接口, Spring 就将调用他们的 postProcessAfterInitialization()方法。
- ⑨ 此时, Bean 已经准备就绪, 可以被应用程序使用了。他们将一直驻留在应用上下文中, 直到应用上下文被销毁。
- ⑩ 如果 bean 实现了 DisposableBean 接口, Spring 将调用它的 destroy()接口方法, 同样, 如果 bean 使用了 destroy-method 声明销毁方法, 该方法也会被调用。

解决

【调用栈】postProcessBeforeInitialization 系统类一

```
initializeBean:1776, AbstractAutowireCapableBeanFactory
(org.springframework.beans.factory.support)
doCreateBean:595, AbstractAutowireCapableBeanFactory
(org.springframework.beans.factory.support)
createBean:517, AbstractAutowireCapableBeanFactory
(org.springframework.beans.factory.support)
lambda$doGetBean$0:323, AbstractBeanFactory (org.springframework.beans.factory.support)
getObject:-1, 23197359
(org.springframework.beans.factory.support.AbstractBeanFactory$$Lambda$32)
getSingleton:226, DefaultSingletonBeanRegistry (org.springframework.beans.factory.support)
doGetBean:321, AbstractBeanFactory (org.springframework.beans.factory.support)
getBean:207, AbstractBeanFactory (org.springframework.beans.factory.support)
invokeBeanFactoryPostProcessors:90, PostProcessorRegistrationDelegate
(org.springframework.context.support)
```



```

.....
invokeBeanFactoryPostProcessors:706,                                AbstractApplicationContext
(org.springframework.context.support)
refresh:532, AbstractApplicationContext (org.springframework.context.support)
<init>:89, AnnotationConfigApplicationContext (org.springframework.context.annotation)
main:9, App (org.bunny.spring.source)
;;
org.springframework.context.annotation.internalConfigurationAnnotationProcessor
org.springframework.context.event.internalEventListenerProcessor
org.springframework.context.event.internalEventListenerFactory
[]

```

【调用栈】postProcessBeforeInitialization 系统类二

```

initializeBean:1776,                                                AbstractAutowireCapableBeanFactory
(org.springframework.beans.factory.support)
doCreateBean:595,                                                  AbstractAutowireCapableBeanFactory
(org.springframework.beans.factory.support)
createBean:517,                                                    AbstractAutowireCapableBeanFactory
(org.springframework.beans.factory.support)
lambda$doGetBean$0:323, AbstractBeanFactory (org.springframework.beans.factory.support)
getObject:-1,                                                       23197359
(org.springframework.beans.factory.support.AbstractBeanFactory$$Lambda$32)
getSingleton:226, DefaultSingletonBeanRegistry (org.springframework.beans.factory.support)
doGetBean:321, AbstractBeanFactory (org.springframework.beans.factory.support)
getBean:207, AbstractBeanFactory (org.springframework.beans.factory.support)
registerBeanPostProcessors:208,                                     PostProcessorRegistrationDelegate
(org.springframework.context.support)
registerBeanPostProcessors:722,                                     AbstractApplicationContext
(org.springframework.context.support)
refresh:535, AbstractApplicationContext (org.springframework.context.support)
<init>:89, AnnotationConfigApplicationContext (org.springframework.context.annotation)
main:9, App (org.bunny.spring.source)
;;
org.springframework.context.annotation.internalAutowiredAnnotationProcessor
org.springframework.context.annotation.internalCommonAnnotationProcessor
org.springframework.context.annotation.internalPersistenceAnnotationProcessor
[]

```

【调用栈】postProcessBeforeInitialization 用户类

```

initializeBean:1776,                                                AbstractAutowireCapableBeanFactory
(org.springframework.beans.factory.support)
doCreateBean:595,                                                  AbstractAutowireCapableBeanFactory
(org.springframework.beans.factory.support)

```

```
.....
createBean:517,                                     AbstractAutowireCapableBeanFactory
(org.springframework.beans.factory.support)
lambda$doGetBean$0:323, AbstractBeanFactory (org.springframework.beans.factory.support)
getObject:-1,                                       23197359
(org.springframework.beans.factory.support.AbstractBeanFactory$$Lambda$32)
getSingleton:226, DefaultSingletonBeanRegistry (org.springframework.beans.factory.support)
doGetBean:321, AbstractBeanFactory (org.springframework.beans.factory.support)
getBean:202, AbstractBeanFactory (org.springframework.beans.factory.support)
preInstantiateSingletons:895,                      DefaultListableBeanFactory
(org.springframework.beans.factory.support)
finishBeanFactoryInitialization:878,               AbstractApplicationContext
(org.springframework.context.support)
refresh:550, AbstractApplicationContext (org.springframework.context.support)
<init>:89, AnnotationConfigApplicationContext (org.springframework.context.annotation)
main:9, App (org.bunny.spring.source)
;;
containerConfig
apple
pen
pineapple
[]
```

调用栈图解

```

509  * Return the list of statically specified ApplicationListeners.
510  */
511  public Collection<ApplicationListener<?>> getApplicationListeners() { return this.applicationListeners; }
514
515  @Override
516  public void refresh() throws BeansException, IllegalStateException {
517      synchronized (this.startupShutdownMonitor) {
518          // Prepare this context for refreshing.
519          prepareRefresh();
520
521          // Tell the subclass to refresh the internal bean factory.
522          ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
523
524          // Prepare the bean factory for use in this context.
525          prepareBeanFactory(beanFactory);
526
527          try {
528              // Allows post-processing of the bean factory in context subclasses
529              postProcessBeanFactory(beanFactory);
530
531              // Invoke factory processors registered as beans in the context.
532              invokeBeanFactoryPostProcessors(beanFactory);
533
534              // Register bean processors that intercept bean creation
535              registerBeanPostProcessors(beanFactory);
536
537              // Initialize message source for this context
538              initMessageSource();
539
540              // Initialize event multicaster for this context
541              initApplicationEventMulticaster();
542
543              // Initialize other special beans in specific context subclasses.
544              onRefresh();
545
546              // Check for listener beans and register them
547              registerListeners();
548
549              // Instantiate all remaining (non-lazy-init) singletons.
550              finishBeanFactoryInitialization(beanFactory);
551
552              // Last step: publish corresponding event
553              finishRefresh();
554          }
555      }
556  }

```

Annotations and classes associated with the code:

- `InternalConfigurationAnnotationProcessor`
- `InternalEventListenerProcessor`
- `InternalEventListenerFactory`
- `InternalAutowiredAnnotationProcessor`
- `InternalCommonAnnotationProcessor`
- `InternalPersistenceAnnotationProcessor`
- `containerConfig`
- `apple`
- `pen`
- `pineapple`

Spring Bean 加载的关键类:

`org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory`

关键方法:

```

protected Object initializeBean(final String beanName, final Object bean, @Nullable
RootBeanDefinition mbd) {
    if (System.getSecurityManager() != null) {
        AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
            invokeAwareMethods(beanName, bean);
            return null;
        }, getAccessControlContext());
    }
    else {
        invokeAwareMethods(beanName, bean);
    }
}

```

```

    }

    Object wrappedBean = bean;
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
    }

    try {
        invokeInitMethods(beanName, wrappedBean, mbd);
    }
    catch (Throwable ex) {
        throw new BeanCreationException(
            (mbd != null ? mbd.getResourceDescription() : null),
            beanName, "Invocation of init method failed", ex);
    }
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
    }

    return wrappedBean;
}

```

【问题描述】中 Spring Bean 的加载流程即是根据这个函数的执行流程总结出来的。

□

Spring Framework 源码阅读

之 Bean 加载：特殊注解

问题描述

Spring Boot	2.2.5
Spring Framework	5.2.6
JDK	1.8

Spring Bean 加载过程涉及以下特殊注解：

-
1. @Lazy
 2. @Primary
 3. @DependsOn
 4. @Role
 5. @Description

Spring 源码中对这些特殊注解做了详尽的解释，这篇笔记摘录了其中的重点

解决

27. @Lazy

Spring Bean 加载方式有两种：

- Eager Initialization
- Lazy Initialization

规则<一>：若 Java Bean 被@Lazy 注释，则该类在被其它 Bean 引用之前不加载

@Configuration 类中，@Bean 注解的 Java Bean 默认行为是懒加载(lazy initialization)

规则<二>：@Lazy 注解可以改变@Configuration 类中，@Bean 注解的加载行为

28. @Primary

规则<一>：当多个 Java Bean 符合@Autowired 条件时，被@Primary 注解的 Bean 优先注入

29. @DependsOn

规则<一>：被@DependsOn 注解的类迟于依赖项加载、先于依赖项销毁

30. @Role

@Role 注解包含以下取值：

- BeanDefinition.ROLE_APPLICATION
- BeanDefinition.ROLE_INFRASTRUCTURE
- BeanDefinition.ROLE_SUPPORT

31. @Description

对 Java Bean 的文字说明

□

Spring Framework 源码阅读

之 Bean 加载：重名加载

问题描述

Spring Boot 2.2.5
Spring Framework 5.2.6
JDK 1.8

Spring Bean 加载的本质是将 Bean Name 字符串（key）和 Bean Definition 对象（value）存进 beanDefinitionMap（ConcurrentHashMap）中

Spring 框架对 beanDefinitionMap 做了缓存，增加了代码逻辑的复杂度

当两个 Java Bean 重名时，Spring 框架会根据不同情况抛出异常或打印日志

解决

```
BeanDefinition existingDefinition = this.beanDefinitionMap.get(beanName);
if (existingDefinition != null) {
    if (!isAllowBeanDefinitionOverriding()) {
        throw new BeanDefinitionOverrideException(beanName, beanDefinition, existingDefinition);
    }
    else if (existingDefinition.getRole() < beanDefinition.getRole()) {
        // e.g. was ROLE_APPLICATION, now overriding with ROLE_SUPPORT or ROLE_INFRASTRUCTURE
        if (logger.isInfoEnabled()) {
            logger.info("Overriding user-defined bean definition for bean '" + beanName +
                "' with a framework-generated bean definition: replacing [" +
                    existingDefinition + "] with [" + beanDefinition + "]");
        }
    }
}
else if (!beanDefinition.equals(existingDefinition)) {
    if (logger.isDebugEnabled()) {
        logger.debug("Overriding bean definition for bean '" + beanName +
            "' with a different definition: replacing [" + existingDefinition +
                "] with [" + beanDefinition + "]");
    }
}
```

```

else {
    if (logger.isTraceEnabled()) {
        logger.trace("Overriding bean definition for bean '" + beanName +
            "' with an equivalent definition: replacing [" + existingDefinition +
            "] with [" + beanDefinition + "]");
    }
}
this.beanDefinitionMap.put(beanName, beanDefinition);
}

```

规则<一>: 当不允许覆盖重名 Bean 时, 抛出 BeanDefinitionOverrideException 异常
注 可以在 application.yml 文件中使用配置项 spring.main.allow-bean-definition-overriding 配置

规则<二>: 当以低@Role 优先级覆盖高@Role 优先级时, 打印 info 日志

规则<三>: 当 Bean 名字相同, 定义不同时, 打印 debug 日志

规则<四>: 当 Bean 重复定义时, 打印 trace 日志

□

Spring Framework 源码阅读

之 Bean 的取名规则

问题描述

Spring Boot	2.2.5
Spring Framework	5.2.6
JDK	1.8

Spring Framework IoC 部分文档并未包含框架的全部细节, 有些神秘主义。
这篇技术笔记记录了 Spring IoC 容器对 Java Bean 的命名规则

命名方式包括两种:

1. 通过注解命名

2. 默认命名

解决

32. 通过注解命名

```
protected boolean isStereotypeWithNameValue(String annotationType,
    Set<String> metaAnnotationTypes, @Nullable Map<String, Object> attributes) {

    boolean isStereotype = annotationType.equals(COMPONENT_ANNOTATION_CLASSNAME) ||
        metaAnnotationTypes.contains(COMPONENT_ANNOTATION_CLASSNAME) ||
        annotationType.equals("javax.annotation.ManagedBean") ||
        annotationType.equals("javax.inject.Named");

    return (isStereotype && attributes != null && attributes.containsKey("value"));
}
```

规则<一>: 当 Java Bean 包含如下注解时, 取其 value 值作为 Bean 的名字:

1. @Component
2. @ManagedBean
3. @Named

33. 默认命名

```
public static String getShortNameAsProperty(Class<?> clazz) {
    String shortName = getShortName(clazz);
    int dotIndex = shortName.lastIndexOf(PACKAGE_SEPARATOR);
    shortName = (dotIndex != -1 ? shortName.substring(dotIndex + 1) : shortName);
    return Introspector.decapitalize(shortName);
}
```

规则<二>: 将纯粹的短类名首字母小写作为 Java Bean 的名字

例如: AnimalBean ----> animalBean

□

Spring Framework 源码阅读

之推断主类

问题描述

Spring Boot 2.2.5
Spring Framework 5.2.6
JDK 1.8

Spring Boot 重要的启动类 `SpringApplication` 中包含一个的方法 `deduceMainApplicationClass(...)`，使用运行时异常（`RuntimeException`）获取调用栈，进而推断主类。尽管没有找到应用场景，却是一个机巧的实现

在这篇技术笔记中做一个记录

解决

34. 源码

```
private Class<?> deduceMainApplicationClass() {  
    try {  
        StackTraceElement[] stackTrace = (new RuntimeException()).getStackTrace();  
        StackTraceElement[] var2 = stackTrace;  
        int var3 = stackTrace.length;  
  
        for(int var4 = 0; var4 < var3; ++var4) {  
            StackTraceElement stackTraceElement = var2[var4];  
            if ("main".equals(stackTraceElement.getMethodName())) {  
                return Class.forName(stackTraceElement.getClassName());  
            }  
        }  
    } catch (ClassNotFoundException var6) {  
    }  
  
    return null;  
}
```

35. 测试类

```
public class Deduce {  
    public static void main(String[] args) {
```

```

        new DDeduce().deduce();
        System.out.println("pause");
    }

    static class DDeduce {
        public void deduce() {
            StackTraceElement[] stackTraceElements = (new RuntimeException()).getStackTrace();
            System.out.println("pause");
        }
    }
}

```

□

SSM 框架源码解读

Spring 脱坑记

之 MyBatis 动态 SQL 处理过程

问题描述

MyBatis 本质上是一个数据库中间件，重点是模板引擎。了解 MyBatis 动态 SQL 处理流程有助于避开书写 Mapper 中的一些错误。

解决方案

按处理流程顺序列举如下：

处理 selectKey 标签 SelectKey 需要注意 order 属性，像 MySQL 一类支持自动增长类型的数据库中，order 需要设置为 after 才会取到正确的值，像 Oracle 这样取序列的情况，需要设置为 before。

替换\${}变量 迭代解析 SQL 节点，将 context 中键值对的值替换\${}中的变量

处理标签 依次处理以下标签

Trim
Where
Set
Foreach
If
Choose
When
Otherwise
Bind

替换#{ }变量 不经过 jdbc 关系映射，直接替换进 SQL Context。可使用内部变量

_parameter
_databaseId

遍历 AST 送入 SQLSession 中执行，并完成后续 resultMap 流程（包括嵌套映射）

□

Spring 脱坑记

之 MyBatis 嵌套查询

问题描述

在配置 resultMap 时，会用到 association 标签。若设置其 select 属性（Attribute），则有框架完成 join 语句拼接工作；否则，进行嵌套查询：

```
<association property="vendor"
    resultMap="com.zte.ums.u32.pms.dis.busipack.osf.dao.VendorDao.resultProxy" />
```

解决方案

举例说明 MyBatis 嵌套查询过程：

```
SELECT l.name course_name, u.id uid, u.user_name
FROM jdams_school_live l LEFT
JOIN jdams_school_live_users u ON l.id = u.live_id
WHERE l.yn = 1
AND l.id = 121
ORDER BY course_start_time
```

第一步 处理第一行的值，创建 LiveCourse 对象，同时创建 User 对象，并放到 List 中，然后设置 LiveCourse 的 users 属性

第二步 处理第二行的值，因为在第一行已经创建了 LiveCourse 对象，所以这一次不会再创建 LiveCourse 对象，根据 rowKey 来判断创建没创建 LiveCourse 对象
第三步 创建 User 对象，然后放到 List 中

□

Spring 脱坑记

之 MyBatis 映射关系

问题描述

ORM（Object / Relation Mapping，对象 / 模型映射）是 MyBatis 的核心功能。关系有三种：一对一、一对多、多对多。MyBatis 使用 `associate` 标签表示一对一关系、`collection` 标签表示一对多关系。但是，MyBatis 不支持级联映射，即不支持多对多映射关系

解决方案

示例代码如下：

```
<resultMap type="ThresholdObject" id="ThresholdObjectMap">
  <result property="id" column="ID" typeHandler="typeHandlerLong2Identity"/>
  ...
  <association property="timeMasks" column="ID"

select="com.zte.ums.u32.pms.dis.pmalarm.common.dao.mapper.IMapperThresholdTimeMasks.selectByThresholdObjectId" />

  <collection property="formulas" column="ID"
    javaType="ArrayList" ofType="ThresholdFormula"

select="com.zte.ums.u32.pms.dis.pmalarm.common.dao.mapper.IMapperThresholdFormula.selectByThresholdObjectId" />

</resultMap>
```

注一 多对一映射可通过对称关系转换为一对多映射关系.

注二 多对多映射需额外定义一张关系表.

□

Spring 脱坑记

之 MyBatis 栅栏问题

问题描述

栅栏问题（Fence-post）是对接工程数据库中的一个常见问题。若使用 Java 编码的方式进行处理,会产生许多垃圾代码。MyBatis 框架通过 trim 标签中的 suffixOverride 属性（Attribute）提供了一个便捷的解决方案。

解决方案

示例代码如下：

```
<update id="updateRecords2Start" parameterType="java.util.List">
    UPDATE <include refid="tableName"/>
    <trim prefix="set" suffixOverrides=",">
        <foreach collection="list" item="item" index="index">
            STATUS=#{item.status}
        </foreach>
        <foreach collection="list" item="item" index="index">
            START_TIME=#{item.startTime}
        </foreach>
    </trim>
    WHERE
    <foreach collection="list" separator="or" item="item" index="index">
        SOURCE_ID=#{item.sourceId} AND FILEPATH=#{item.filePath}
    </foreach>
</update>
```

□

Spring 脱坑记

之 MyBatis 遍历逗号分隔字符串

问题描述

在项目实践中,有时不会对表关系做完全分解,即部分一对多映射关系以逗号分隔字符串的方式直接存储到数据表中。那么,在 MyBatis Mapper 配置文件中,处理

这种映射关系是一个常见的错误点。

解决方案

配置代码如下：

```
<select id="selectResNameById" parameterType="String"
    resultType="String">
    SELECT
    SHORT_NAME AS NAME
    FROM
    TM_RES_TYPE
    WHERE
    ID in
    <foreach item="item" index="index" collection='_parameter.split(",")' open="("
separator="," close=")">
        ${item}
    </foreach>
</select>
```

注一 在属性（Attribute）值中，使用 `_parameter` 指代入参。

注二 `#{}` 中的变量会进行 jdbc 映射，而 `${}` 中的变量直接进行字符串拼接。

注三 非数字型（long、integer）变量需要用引号包裹（Wrap）。

□

Spring 脱坑记

之 Spring IoC: Introduction

问题描述

Spring IoC

解决方案

什么是 IoC(控制反转)?

控制反转是面向对象编程中的一种设计原则，可以用来降低计算机代码之间的耦合度。控制反转容器可以帮助避免接口和实现的耦合，即接口的实现不能硬编码，避免耦合。编码的时候只关注接口，接口的具体实现依赖容器完成。

常用的 IoC 容器有：Spring Beans / Context、EJB、Pico Container、Avalon、JBoss、HiveMind 等。

Spring IoC 容器有哪些特点？

有 ApplicationContext 并能按照类型获取 Bean

被@Component 标记的类会被自动加载到容器中

被@Autowired 标记的属性会自动注入

Spring IoC 采用两级级联容器：DispatcherServlet 创建的 IoC 容器的父容器就是 ContextLoaderListener 创建的 IoC 容器。父容器管理的 Bean 可以动态注入到子容器中；同时，保证子容器中的 Bean 相互隔离。这种设计思想曾用于 Ali WebX 框架(Deprecated)

项目中是如何使用 Spring IoC 容器的？

```
private void createApplicationContext() {
    File[] applicationContextFiles = PmFileUtils.getFiles(new File(PmFileUtils.getAppHome() +
"/conf/xml"), Pattern.compile("appctx.*\\.xml"));
    //转换为路径字符串
    List<String> applicationContextFileLocations = new ArrayList<String>();
    if (applicationContextFiles != null) {
        for (File acf : applicationContextFiles) {
            if (acf == null) {
                continue;
            }
            //为解决 unix 系统上 spring 认为以"/"开头的路径是相对路径，所以在文件路径前再加一个
            "file:".
            applicationContextFileLocations.add("file:" + acf.getAbsolutePath());
        }
    }
    //使用 FileSystemXmlApplicationContext 建立上下文对象
    ApplicationContext ctx = new
    FileSystemXmlApplicationContext(applicationContextFileLocations.toArray(new String[0]));
    //添加到 SpringBeanFactory 中，使得句柄被一直持有
    SpringBeanFactory.setApplicationContext(ctx);
}
```

项目采用单一 IoC 容器，一次读出并加载加载所有 Java Bean。小型项目避免使用 Spring 默认配置（级联容器）的成功实践。

□

问题描述

Spring IoC

解决方案

什么是 IoC(控制反转)?

控制反转并不是 Spring 所特有的，Spring 为我们所做的更多的可以说是为我们创建并管理 Bean，然后在我们需要的时候再依赖注入（Dependency Injection）给我们。这也是为什么后来 Spring IoC 更多又叫 Spring DI。Spring IoC 是一种思想，而 DI 依赖注入是这种思想的具体实现方案。

Web 容器是如何启动的？

一般来讲，Web 容器（Tomcat 等）启动 WebApplication 会有以下步骤：

1. Web 容器（Tomcat 等）在启动 Web 应用程序的时候会为每个 WebApplication 创建唯一的 ServletContext 对象。可以把 ServletContext 看作一个 Web 应用的服务器端组件的共享内存。Web 应用的所有部分都可以使用该上下文 ServletContext。
2. Web 容器将 `<context-param></context-param>` 解析为 key-value 对，并交给 ServletContext。
3. Web 容器根据 `<listener></listener>` 中的类创建 ServletContextListener 监听实例，即启动监听。
4. 在 ServletContext 初始化的时候 Web 容器会回调 ServletContextListener 监听类中 `contextInitialized(ServletContextEvent servletContextEvent)` 初始化方法。这里就是我们的 SpringWebApplication 的 Spring 工厂开始创建的地方。
5. Web 容器解析 `<filters></filters>`，并启动拦截器。拦截器开始起作用，当有请求进入时，执行 Filter 的 `doFilter` 方法。
6. 初始化 Servlet 等

Web 容器启动 WebApplication 的时候主要做了 2 件事：

1. Web 容器会帮我们创建启动 XMLWebApplicationContext 工厂
2. 将我们配置的 `contextConfigLocation`（即 `spring-application.xml`）设置给我们的 Spring 工厂 ApplicationContext。

ApplicationContext 是 Spring 提供的一个高级的 IoC 容器，它除了能够提供 IoC 容器的基本功能外，还为用户提供了国际化等附加服务。

Spring IoC 容器有哪些特点?

Import

Alias (别名)

Bean

Beans

FactoryBean 如果要获取 FactoryBean 对象, 需要在 bean 的 id 或者 name 前面加一个&符号来获取

项目中是如何使用 Spring IoC 容器的?

项目中使用私有 SpringBeanFactory 代理 Spring 框架的 ApplicationContext 工厂类。

ModuleInitializer.java

```
private void createApplicationContext() {  
    SpringBeanFactory.setApplicationContext(ctx);  
}
```

```
cleanMetedataToolView = (CleanMetedataToolView)  
SpringBeanFactory.getSpringBean("cleanMetedataToolView");
```

□

Spring 脱坑记

之分布式锁

问题描述

分布式锁

解决方案

什么是分布式锁? 有什么作用?

分布式锁是控制分布式系统之间同步访问共享资源的一种方式。在单机或者单进程环境下, 多线程并发的情况下, 使用锁来保证一个代码块在同一时间内只能由一个线程执行。比如 Java 的 Synchronized 关键字和 Reentrantlock 类。

多进程或者分布式集群环境下, 如何保证不同节点的线程同步执行呢? 这就是分布式锁。

分布式锁可以通过什么来实现?

1. Memcached 分布式锁

Memcached 提供了原子性操作命令 add, 才能 add 成功, 线程获取到锁。

key 已存在的情况下, 则 add 失败, 获取锁也失败。

2. Redis 分布式锁

Redis 的 `setnx` 命令为原子性操作命令。只有在 `key` 不存在的情况下，才能 `set` 成功。和 Memcached 的 `add` 方法比较类似。

3. ZooKeeper 分布式锁

利用 ZooKeeper 的顺序临时节点，来实现分布式锁和等待队列。

4. Chubby 分布式锁

Chubby 底层利用了 Paxos 一致性算法，实现粗粒度分布式锁服务。

介绍一下分布式锁实现需要注意的事项？

分布式锁实现要保证几个基本点

1. 互斥性 任意时刻，只有一个资源能够获取到锁
2. 容灾性 在未成功释放锁的情况下，一定时限内能够恢复锁的正常功能
3. 统一性 加锁和解锁保证统一资源来进行操作

Redis 怎么实现分布式锁？

简单方案

最简单的方法是使用 `setnx` 命令。释放锁的最简单方法是执行 `del` 指令

问题

锁超时 如果一个得到锁的线程在执行任务的过程中挂掉，来不及显式地释放锁，这块资源将会永远被锁住（死锁），别的资源再也别想进来。

优化方案

`Setnx` 没办法设置超时时间，如果利用 `expire` 来设置超时时间，那么这两步操作不是原子性操作

利用 `set` 指令增加了可选参数方法来替代 `setnx`。`Set` 指令可以设置超时时间。

□

运维养成记

运维养成记

之记一次镜像调试

问题描述

项目组采用 Windows、Red Hat Linux 之间的多级“串流”访问远程虚机，在调试镜像的过程中报以下错误：

1. Error: Unable to access jar file
2. /home/run.sh: unable to find
3. /home/project\r is not a directory

说明

1. shell 脚本空行处报错误【2】
2. 在 shell 脚本每行命令结尾添加 “;” 强制截断后，逻辑正常进行

解决方案

简要分析

这是由不可见字符 “\r” 导致的。当 Windows Terminal 中键入 “Enter” 时，串流不可见字符串 “\r\n” 至 Linux Terminal。Linux Terminal 识别 “\n” 为换行符，残留的不可见字符 “\r” 则跟随在每行命令结尾。而且，在 shell 报错时，并不会打印控制字符。从而出现这种常见的 “幽灵报错” 现象。

解决方案

使用 sed、awk 过滤掉控制字符 “\r”

```
cat run.sh | sed 's/\r//' | out.sh
```

```
mv out.sh run.sh
```

□