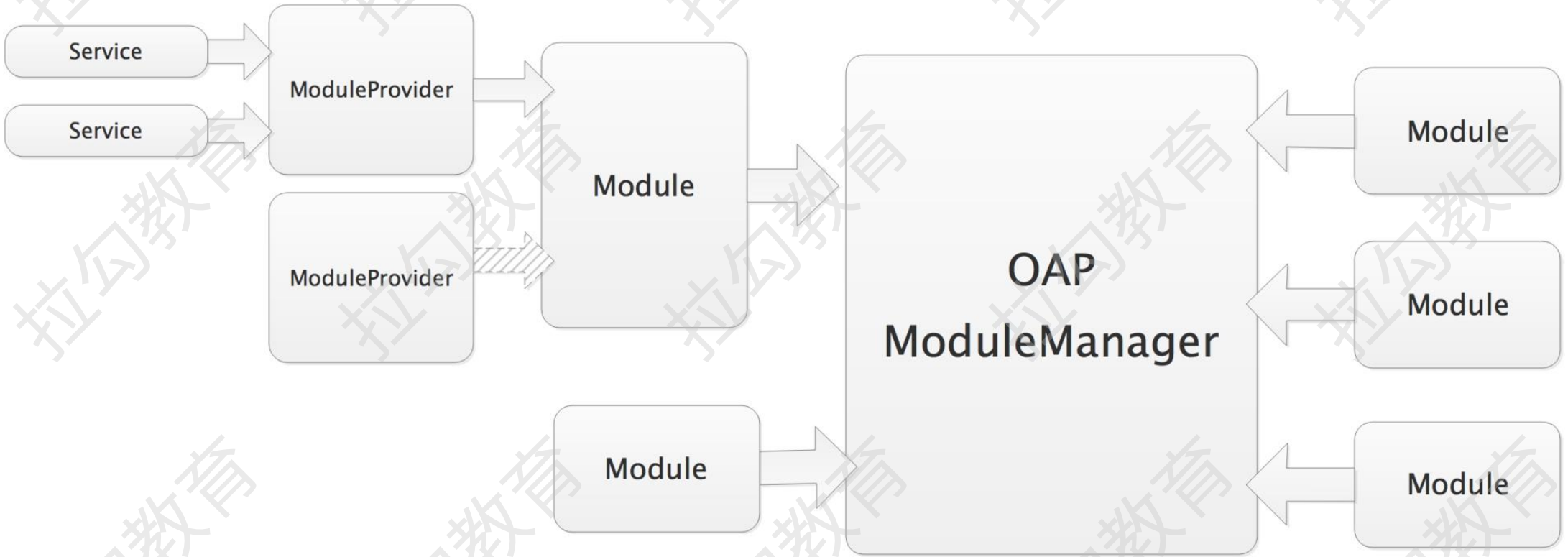
《31 讲带你搞懂 SkyWalking》

徐郡明 资深技术专家

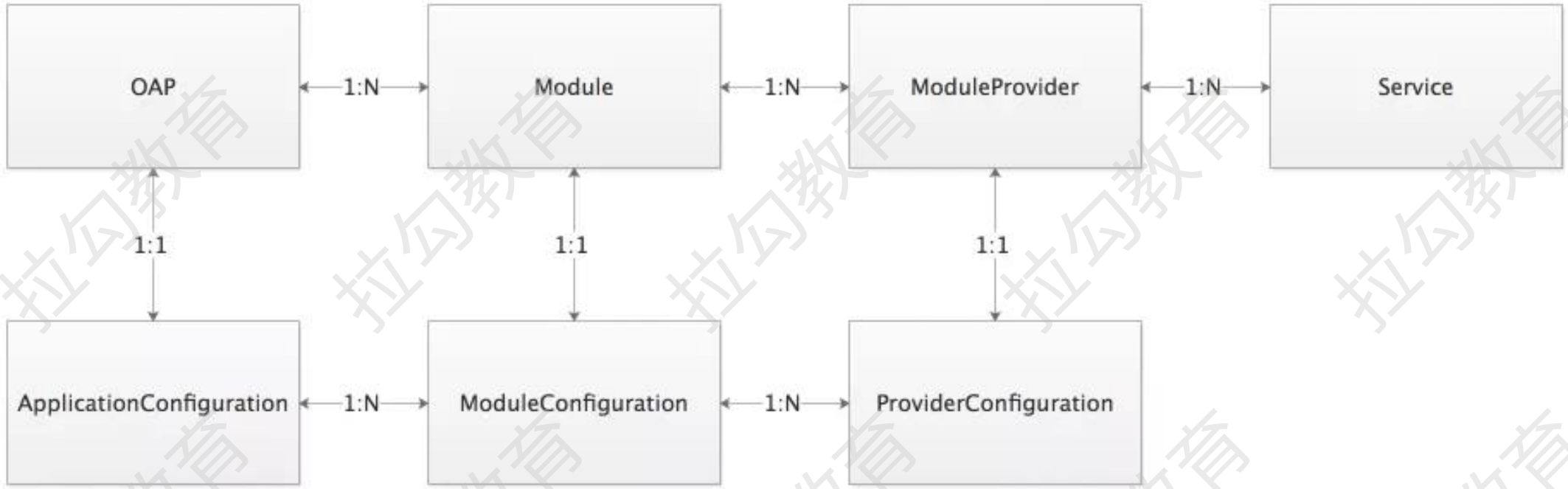
— 拉勾教育出品 —

第19讲：OAP 初始化流程精讲
一眼看透 SkyWalking OAP 骨架

OAP 架构



OAP 架构



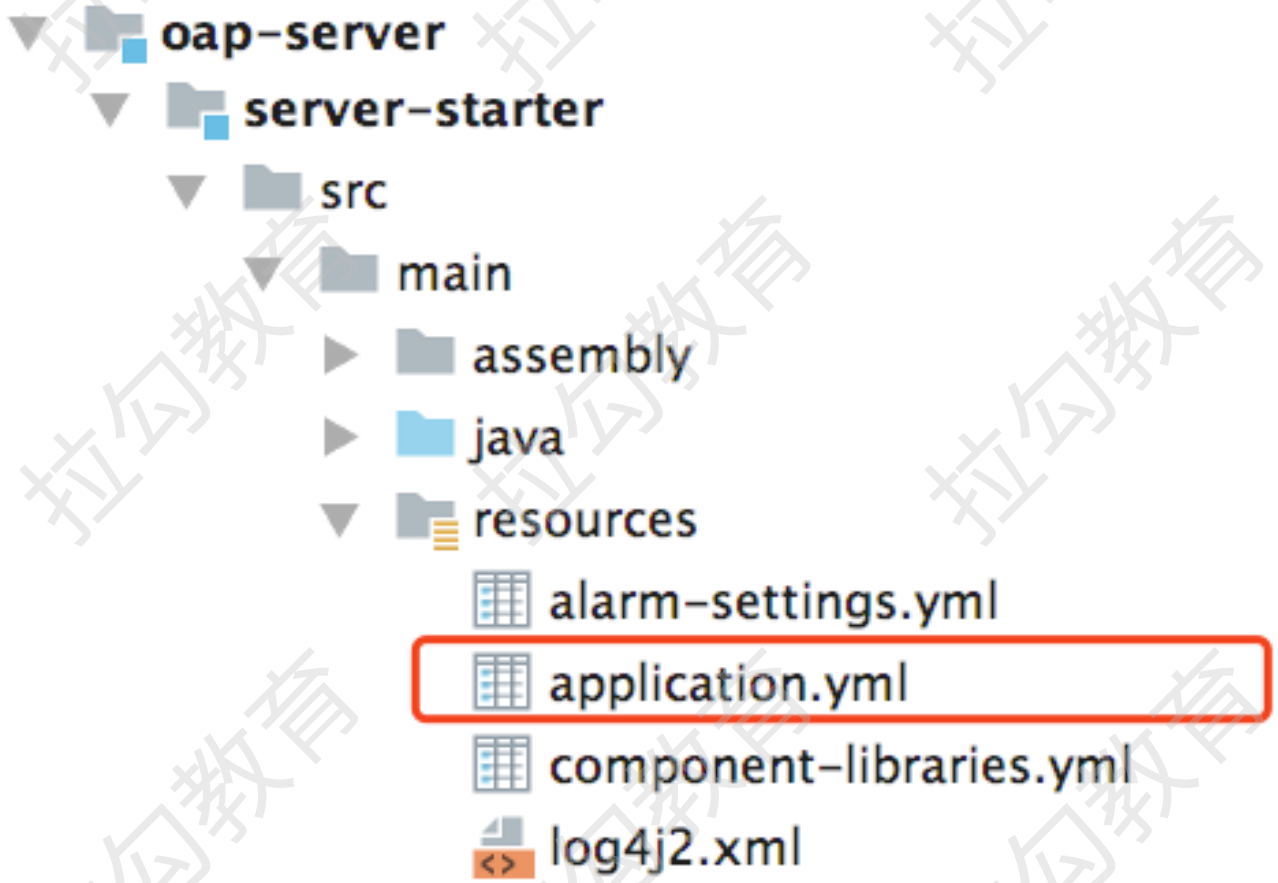
OAP 架构


```
public static void main(String[] args) {
    // mode这个环境变量有两个可选值：init、no-init，其中init值表示初始化底层
    //的存储结构，例如，使用ES时会初始化其中的索引，使用数据库时会初始化表结构；
    // no-init值表示不初始化上述存储结构
    String mode = System.getProperty("mode");
    RunningMode.setMode(mode);
    //创建ApplicationConfigLoader，加载配置文件
    ApplicationConfigLoader configLoader =
        new ApplicationConfigLoader();
    ApplicationConfiguration applicationConfiguration =
        configLoader.load();
    //创建ModuleManager，并根据配置初始化全部Module
    ModuleManager manager = new ModuleManager();
```

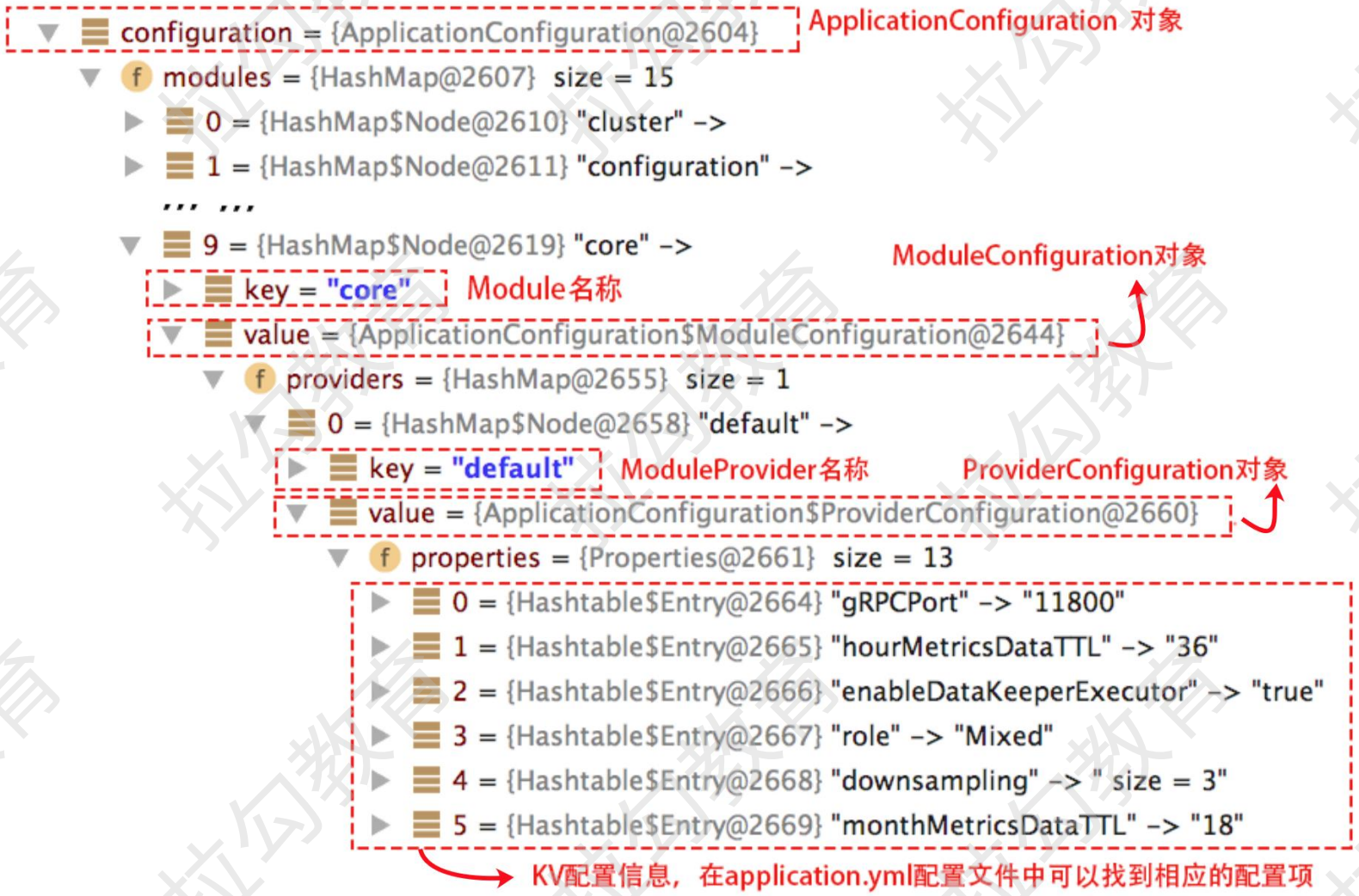
OAP 架构


```
        new ApplicationConfigLoader();
    ApplicationConfiguration applicationConfiguration =
        configLoader.load();
    //创建ModuleManager，并根据配置初始化全部Module
    ModuleManager manager = new ModuleManager();
    manager.init(applicationConfiguration);
    //查找指定Module中的指定Service进行使用，后面会展开核心Module进行分析
    manager.find(TelemetryModule.NAME).provider()
    .getService(MetricsCreator.class).createGauge("uptime",
      "oap server start up time", MetricsTag.EMPTY_KEY,
      MetricsTag.EMPTY_VALUE)
        .setValue(System.currentTimeMillis() / 1000d);
  }
```

初始化配置


oap-server
server-starter
src
main
assembly
java
resources
alarm-settings.yml
application.yml
component-libraries.yml
log4j2.xml

初始化配置



ModuleManager


```
// Key是Module的名称，Value是相应的ModuleDefine，ModuleDefine就是对Module的抽象
Map<String, ModuleDefine> loadedModules = new HashMap<>();
```

ModuleManager


```
public void init(ApplicationConfiguration applicationConfiguration){
    //根据配置拿到所有Module的名称
    String[] moduleNames = applicationConfiguration.moduleList();
    //通过SPI方式加载ModuleDefine接口和 ModuleProvider接口的实现
    ServiceLoader<ModuleDefine> moduleServiceLoader =
        ServiceLoader.load(ModuleDefine.class);
    ServiceLoader<ModuleProvider> moduleProviderLoader =
        ServiceLoader.load(ModuleProvider.class);

    LinkedList<String> moduleList =
        new LinkedList<>(Arrays.asList(moduleNames));
    for (ModuleDefine module : moduleServiceLoader) {
      for (String moduleName : moduleNames) {
```

ModuleManager


```
for (ModuleDefine module : moduleServiceLoader) {
  for (String moduleName : moduleNames) {
    if (moduleName.equals(module.name())) {
      //通过SPI可能加载很多ModuleDefine实现以及ModuleProvider实
      //现类，但是这里只初始化在配置文件中出现过的Module,并调用
      //其prepare()方法
      ModuleDefine newInstance =
        module.getClass().newInstance();
      newInstance.prepare(this, applicationConfiguration
        .getModuleConfiguration(moduleName),
          moduleProviderLoader);
      //记录初始化的ModuleDefine对象
```

ModuleManager

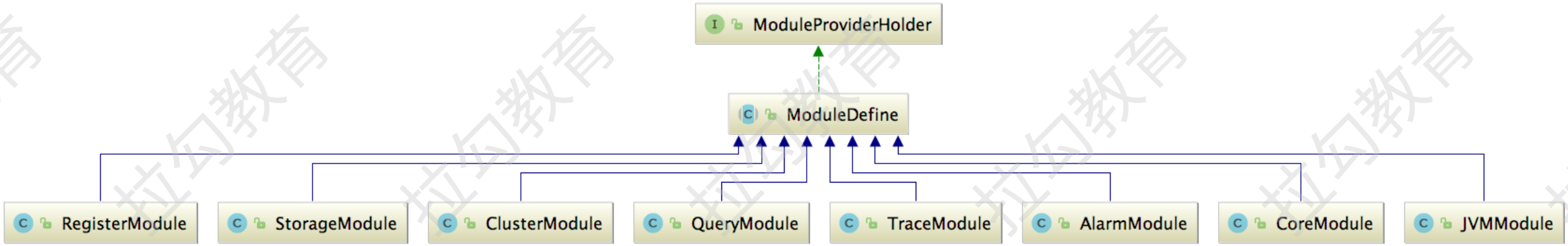

```
                newInstance.prepare(this, applicationConfiguration
                    .getModuleConfiguration(moduleName),
                        moduleProviderLoader);
                //记录初始化的ModuleDefine对象
                loadedModules.put(moduleName, newInstance);
                moduleList.remove(moduleName);
            }
        }
    }

    isInPrepareStage = false;
    //初始化BootstrapFlow，具体的初始化逻辑后面展开分析
    BootstrapFlow bootstrapFlow = new BootstrapFlow(loadedModules);
```

ModuleManager


```
                moduleList.remove(moduleName);
            }
        }
    }
    isInPrepareStage = false;
    //初始化BootstrapFlow，具体的初始化逻辑后面展开分析
    BootstrapFlow bootstrapFlow = new BootstrapFlow(loadedModules);
    // 启动Module
    bootstrapFlow.start(this);
    //启动流程结束之后会通知相关组件
    bootstrapFlow.notifyAfterCompleted();
}
```

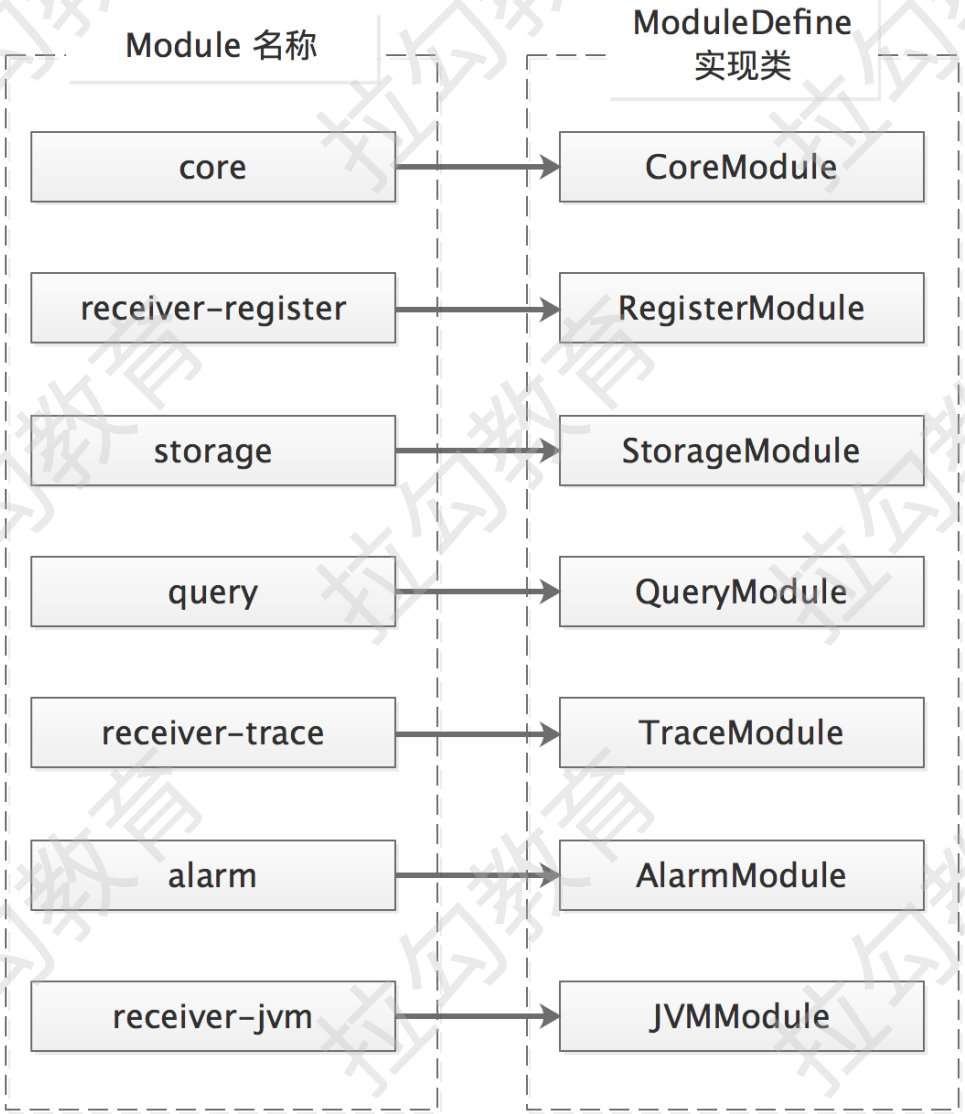
ModuleDefine



ModuleDefine


```
// 当前Module的名称
private final String name;
//属于该Module的所有ModuleProvider对象
private final LinkedList<ModuleProvider> loadedProviders = new
LinkedList<>();
```

ModuleDefine



ModuleDefine


```
void prepare(ModuleManager moduleManager,
    ApplicationConfiguration.ModuleConfiguration configuration,
      ServiceLoader<ModuleProvider> moduleProviderLoader) {
   for (ModuleProvider provider : moduleProviderLoader) {
    //这里只关心配置文件中指定的、与当前Module相关的ModuleProvider实现类
     if (!configuration.has(provider.name())) {
       continue;
     }
     if (provider.module().equals(getClass())) {
       //初始化ModuleProvider对象
       ModuleProvider newProvider =
```

ModuleDefine


```
        //初始化ModuleProvider对象
        ModuleProvider newProvider =
            provider.getClass().newInstance();
        newProvider.setManager(moduleManager);
        //设置ModuleProvider与 Module之间的关联关系
        newProvider.setModuleDefine(this);
        //将ModuleProvider对象记录到loadedProviders集合
        loadedProviders.add(newProvider);
    }
}

//检查:该Module没有任何关联的ModuleProvider，会在这里报错(省略该检查代码)
```

ModuleDefine


```
//检查:该Module没有任何关联的ModuleProvider，会在这里报错(省略该检查代码)
for (ModuleProvider moduleProvider : loadedProviders) {
  //前面读取配置信息时，ModuleProvider的配置信息是存储到ProviderConfig
  //之中的Properties集合之中，此处，每个ModuleProvider都会关联一个
  //ModuleConfig对象，并ProviderConfig中的配置信息拷贝到ModuleConfig
  //对象中的相应字段，实现Properties到Java Bean的转换
  copyProperties(moduleProvider.createConfigBeanIfAbsent(),
      configuration.getProviderConfiguration(
          moduleProvider.name()), this.name(),
              moduleProvider.name());
  //调用ModuleProvider的prepare()方法，继续prepare流程
```

ModuleDefine


```
    //之中的Properties集合之中，此处，每个ModuleProvider都会关联一个
    // ModuleConfig对象，并ProviderConfig中的配置信息拷贝到ModuleConfig
    //对象中的相应字段，实现Properties到Java Bean的转换
    copyProperties(moduleProvider.createConfigBeanIfAbsent(),
        configuration.getProviderConfiguration(
            moduleProvider.name()), this.name(),
                moduleProvider.name());
    //调用ModuleProvider的prepare()方法，继续prepare流程
    moduleProvider.prepare();
}
```

ModuleDefine

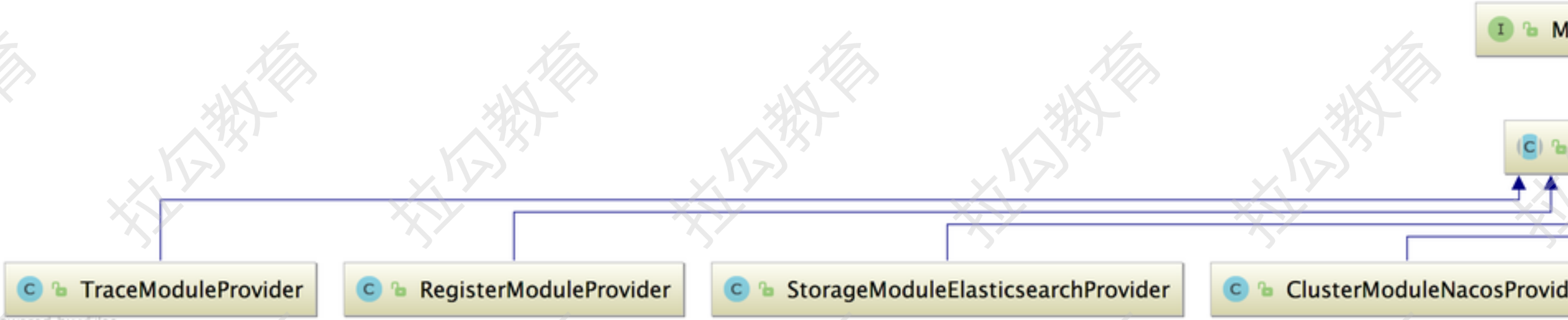

```
public final ModuleProvider provider() throws {
    if (loadedProviders.size() > 1 || loadedProviders.size() == 0) {
        throw new xxxException( “...” ); // 抛异常
    }
    return loadedProviders.getFirst(); //返回唯一的ModuleProvider实例
}
```

ModuleProvider

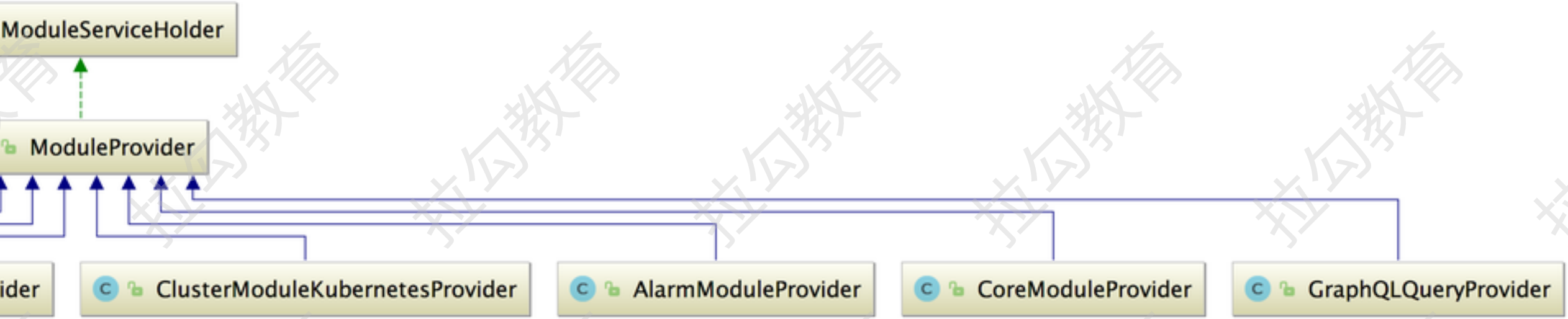

```
public interface ModuleServiceHolder {
    //注册Service实例，当Service类型与service对象不匹配时，会报错
    void registerServiceImplementation(
        Class<? extends Service> serviceType, Service service);

    //获取指定类型的Service对象
    <T extends Service> T getService(Class<T> serviceType);
}
```

ModuleProvider



ModuleProvider



ModuleProvider

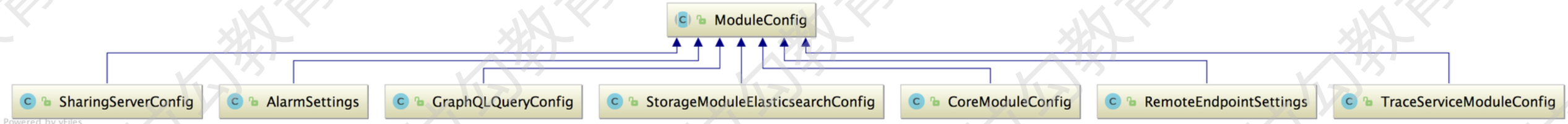
ModuleProvider 中还定义了一些通用的抽象方法：

- name() 方法：返回当前 ModuleProvider 的名称，该名称在同一个 Module 下是唯一的

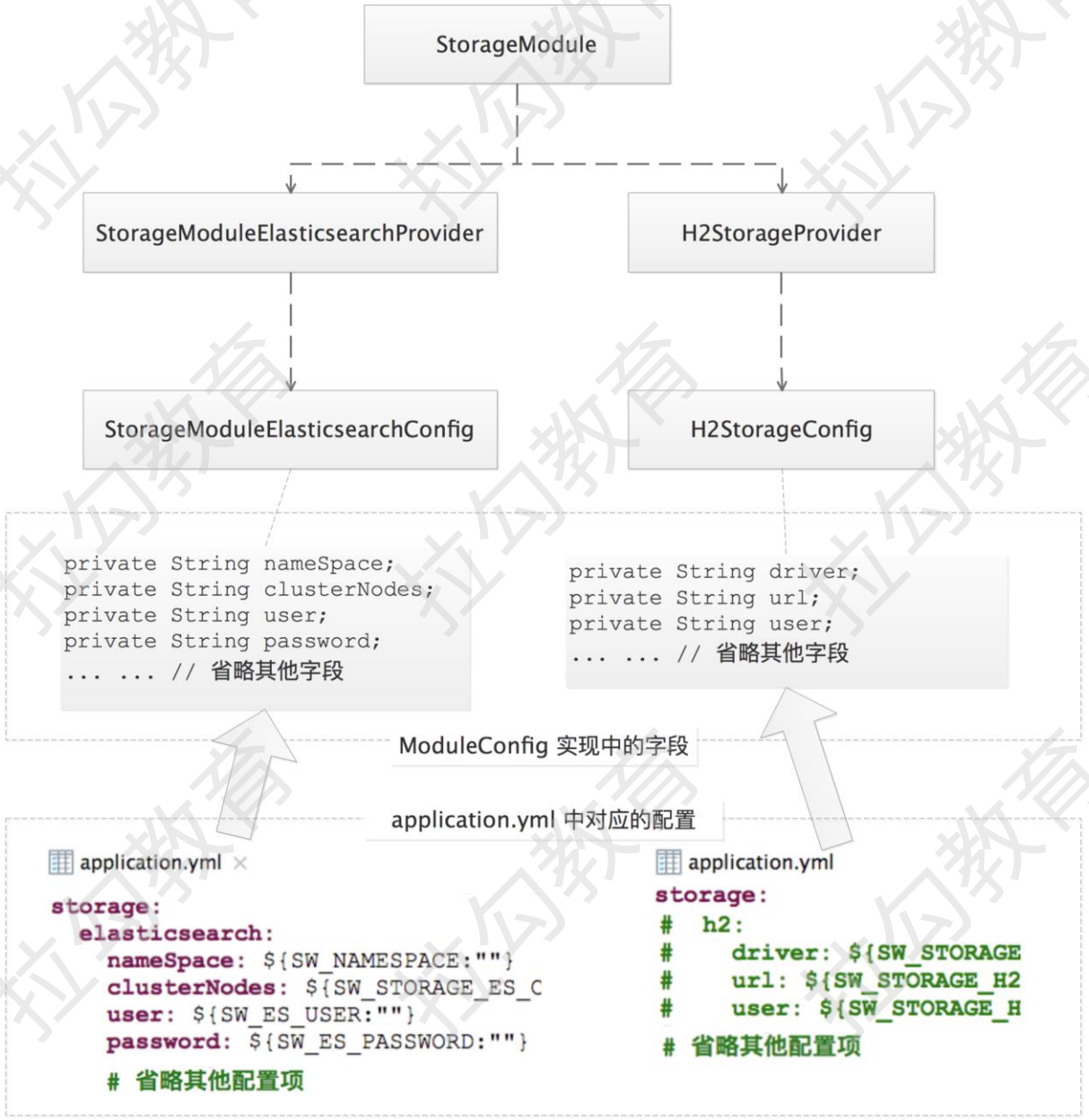
  例如，StorageModule 负责实现 OAP 的持久化存储功能，Module 名称为 "storage"，具体依赖的底层存储可以是 ElasticSearch、H2 等，分别对应StorageModuleElasticsearchProvider、H2StorageProvider 两个 ModuleProvider 实现类，ModuleProvider 名称分别是 "elasticsearch" 和 "h2"

- createConfigBeanIfAbsent() 方法: 返回当前 ModuleProvider 对应的 ModuleConfig 对象ModuleConfig 是一个空的抽象类，其实现类都是用于存储配置信息的 Java Bean

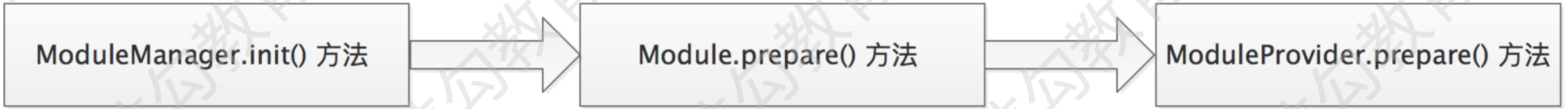
ModuleProvider



ModuleProvider



BootstrapFlow

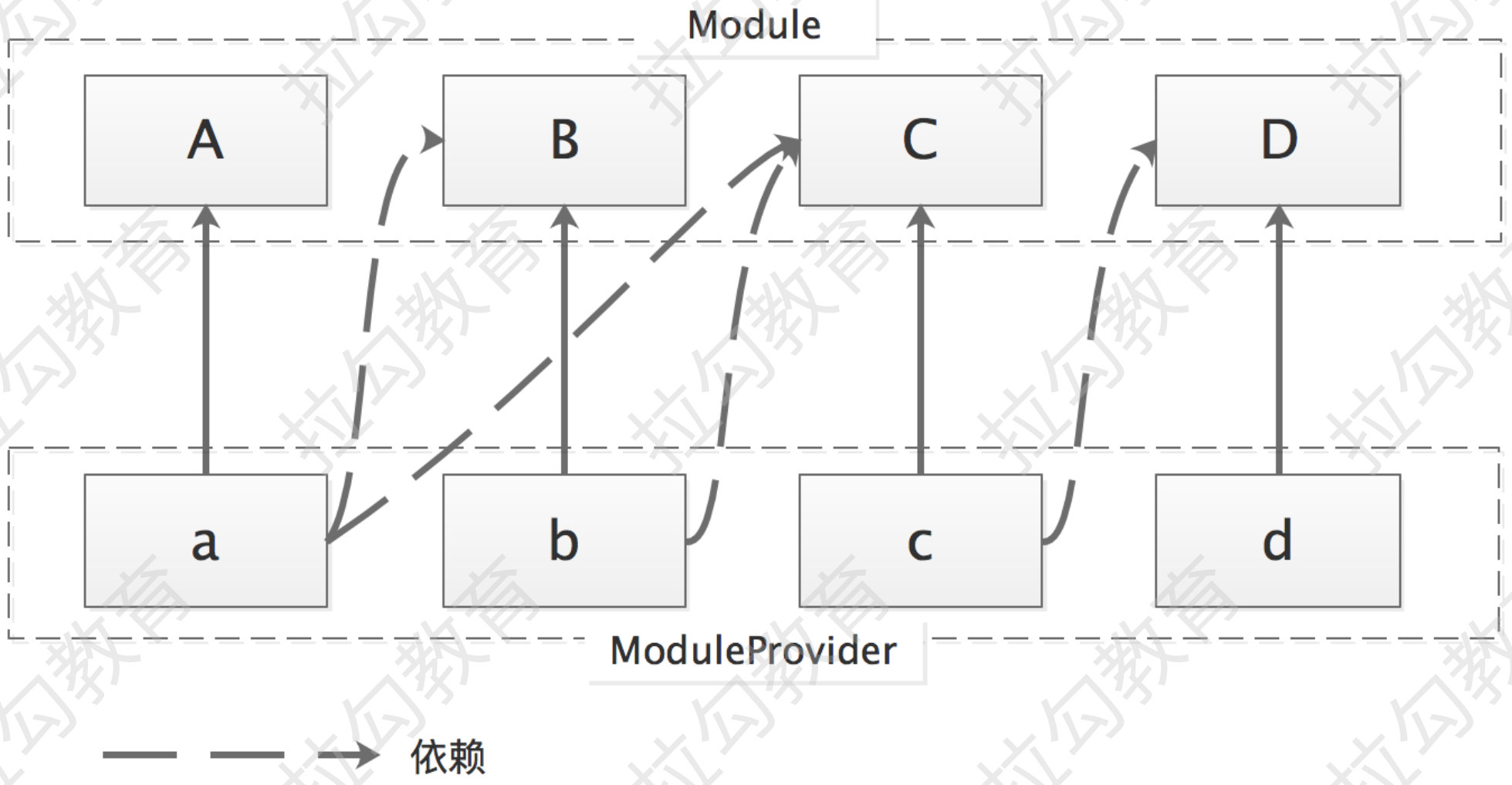


BootstrapFlow

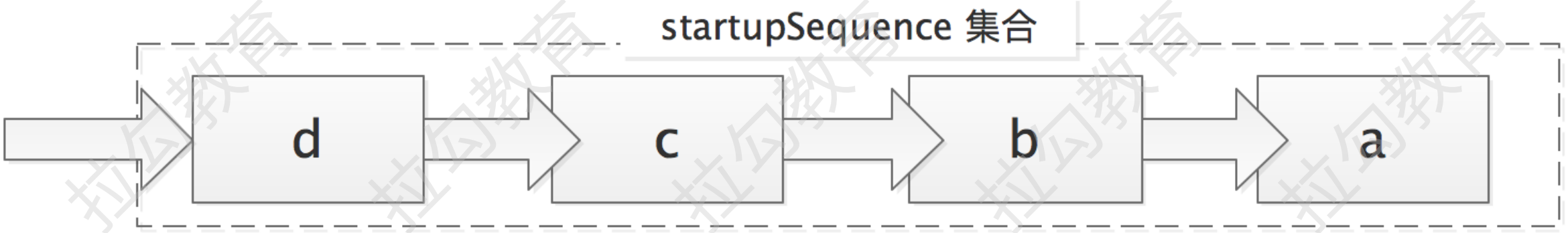
拉勾教育
互联网人实战大学

```
void start(ModuleManager moduleManager) {
  //按照startupSequence集合的顺序启动所有ModuleProvider实例
  for (ModuleProvider provider : startupSequence) {
    //检测当前ModuleProvider依赖的Module对象是否存在
    String[] requiredModules = provider.requiredModules();
    if (requiredModules != null) {
      for (String module : requiredModules) {
        if (!moduleManager.has(module)) { ... ... // ??? }
      }
    }
    //检查当前ModuleProvider对象是否能提供其所属Module需要的Service
    provider.requiredCheck(provider.getModule().services());
    provider.start(); //上述两项检查都通过之后，才能启动ModuleProvider
  }
}
```

BootstrapFlow



BootstrapFlow



Next：第18讲《深入剖析 Configuration 插件，实现可插拔接入多种配置中心》

拉勾教育

—互联网人实战大学—

关注拉勾「教育公众号」
获取更多课程信息

L / A / G / O / U