Web Service

组员：黄泠淇，饶才佳，刘雨晴，顾鸿樾

Overview

Fundamental

- RPC Remote Procedure Call
- REST RESTful API

Web Service

- SOA Service Oriented Architecture
- Microservice

Framework (for RPC & REST)

- 接口风格
- 接口描述语言
- 协议
- 数据传输总线
  基本：自动的数据转换，被隐藏的网络传输
  扩展：负载均衡，Cache等
- 异常和错误处理
- 典型的部署方式
- 其他的特点以及衍生的思考

RPC: Remote Procedure Call

- **接口风格**
- 接口描述语言
- 协议
- 数据传输总线
- 异常和错误处理
- 典型的部署方式
- 其他的特点以及衍生的思考

RPC: 接口风格

RPC 本身是一种PC，是传统PC的延申，他们的接口风格是一致的。

PC (Procedure Call)

- `LDict* l-retrieveDict(char* dicName);`
- `int l-get( LDict *dic, KeyType *key, ValType *val );`
- `int l-set( LDict *dic, KeyType *key, ValType *val );`
- `size_t l-size(LDict *dic);`
- `int l-clear(LDict *dic);`

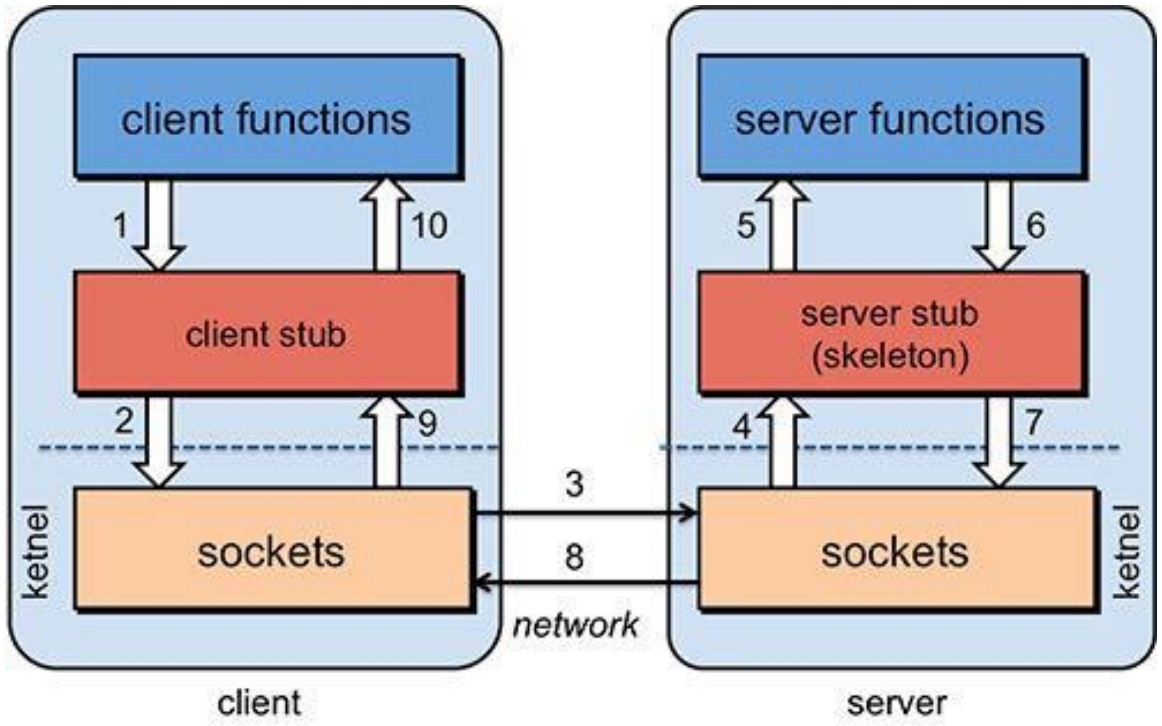
RPC (Remote Procedure Call)

- `RDict* r-retrieveDict(char* dicName);`
- `int r-get( RDict *dic, KeyType *key, ValType *val );`
- `int r-set( RDict *dic, KeyType *key, ValType *val );`
- `size_t r-size(RDict *dic);`
- `int r-clear(RDict *dic);`

RPC: 接口风格

- RPC允许访问其他进程提供的服务;
- 传统的PC仅能访问地址空间中的服务。

- LDict* 引用字典本身;
- RDict* 引用代理（Proxy），远程字典存在于字典管理进程的地址空间中。



RPC: Remote Procedure Call

- 接口风格
- **接口描述语言**
- 协议
- 数据传输总线
- 异常和错误处理
- 典型的部署方式
- 其他的特点以及衍生的思考

RPC: 接口描述语言

- `int l-get(LDict *dic, KeyType *key, ValType *val);`
- `int r-get(RDict *dic, KeyType *key, ValType *val);`

- 面向接口编程:
- `int get(Dict *dic, KeyType *key, ValType *val);`
- get 和 Dict* 是他们的 universal form。

RPC: 接口描述语言

- `int get(Dict *dic, KeyType *key, ValType *val);`
- 接口使用C语言定义（即选择C语言作为接口描述语言）。
- 依赖于C语言。

RPC: 接口描述语言

- IDL：Interface Description Language（源于CORBA）
- 只关注描述接口
- 不依赖任何一种特定的编程语言

- IDL能翻译成某个特定语言的接口描述

RPC: 接口描述语言

```
// gRPC 的 IDL
service Greeter {
  rpc SayHello (HelloRequest) returns (HelloReply) {}
  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}
}
message HelloRequest {
  string name = 1;
}
message HelloReply {
  string message = 1;
}
```

RPC: 接口描述语言

```
// gRPC IDL 翻译成 java interface 的一种可能形式
interface GreeterGrpc {
  static GreeterGrpc getStub(InetAddress addr); // client从addr处，获取一个GreeterGrpc服务的代理对象
  HelloReply sayHello(HelloRequest req);
  HelloReply sayHelloAgain(HelloRequest req);
}
interface HelloRequest {
  static HelloRequestBuilder newBuilder();      // 获取一个builder用于创建HelloRequest数据对象。
  String getName();
}
interface HelloReply {
  static HelloReplyBuilder newBuilder();
  String getMessage();
}
```

RPC: Remote Procedure Call

- 接口风格
- 接口描述语言
- **协议**
- 数据传输总线
- 异常和错误处理
- 典型的部署方式
- 其他的特点以及衍生的思考

RPC: 协议（略）

- RPC是一种进程间通信;
- RPC抽象程度高，没有操作系统直接支持（Windows COM除外）；
- TCP/IP协议跨平台，耦合低;
- 不过TCP/IP是传输层协议，需要一个格式化的应用层协议来承载RPC请求和相应。

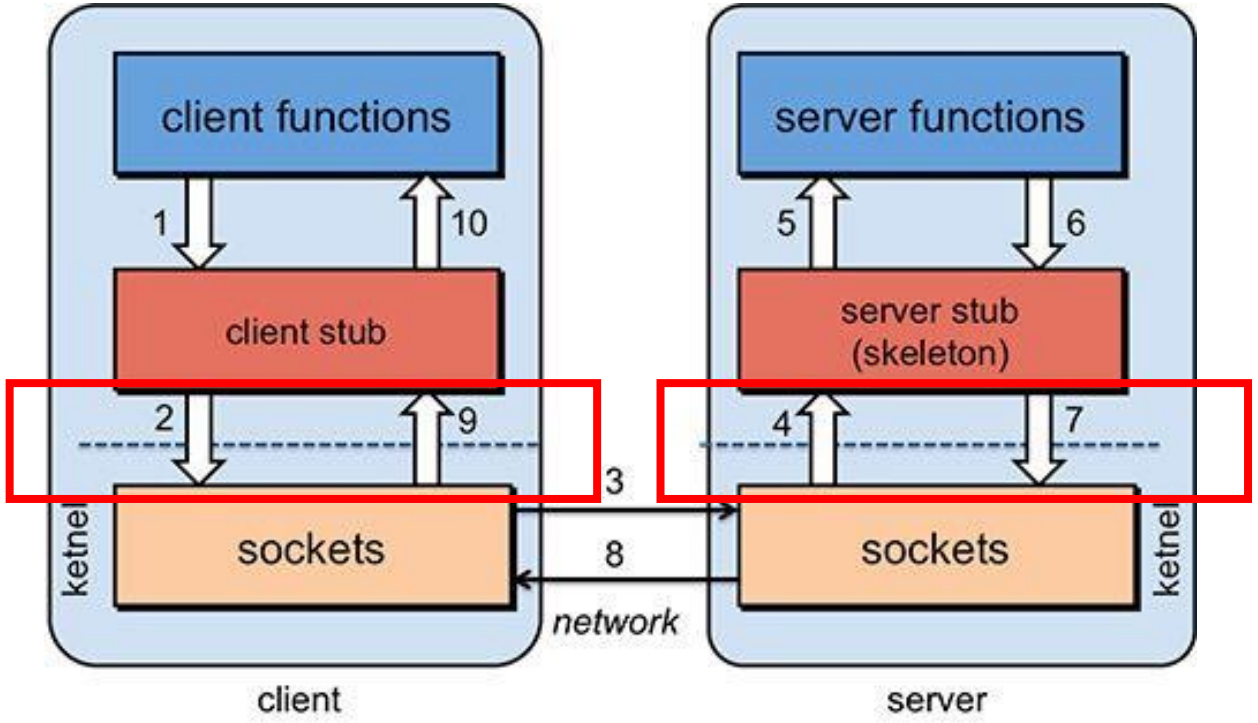
- gRPC的传输协议比较复杂，耦合高（具体协议参考文档）。

RPC: Remote Procedure Call

- 接口风格
- 接口描述语言
- 协议
- **数据传输总线**
- 异常和错误处理
- 典型的部署方式
- 其他的特点以及衍生的思考

RPC: 数据传输总线（略）

- 一般RPC协议复杂，很多团队没有能力去实现和维护“过程调用”与“RPC传输协议”的转换。
- 需要一个自动生成转换代码的功能。



RPC: 数据传输总线（略）

- gRPC根据IDL生成java版本的interface和序列化代码

- ```
  protoc --proto_path=src        \
         --java_out=build/gen    \
         src/foo.proto
  ```

- 这条指令会生成一些“.java”文件，可以在自己的java代码中使用。
  ```

RPC: 数据传输总线（略）

```
// gRPC 的 server 端
private class GreeterImpl extends GreeterGrpc.GreeterImplBase {

  @Override
  public void sayHello(HelloRequest req,
                       StreamObserver<HelloReply> responseObserver) {
    …
  }
  @Override
  public void sayHelloAgain(HelloRequest req,
                            StreamObserver<HelloReply> responseObserver) {
    …
  }
}
```

RPC: Remote Procedure Call

- 接口风格
- 接口描述语言
- 协议
- 数据传输总线
- **异常和错误处理**
- 典型的部署方式
- 其他的特点以及衍生的思考

RPC: 异常和错误处理（略）

- RPC 的接口风格和传统的PC相同，错误/异常处理的风格应该与传统的PC一致。

RPC: Remote Procedure Call

- 接口风格
- 接口描述语言
- 协议
- 数据传输总线
- 异常和错误处理
- **典型的部署方式**
- 其他的特点以及衍生的思考

RPC: 典型的部署方式

```
// RPC服务端以进程为单位部署和发布，在TCP/IP中表现为进程与某个port绑定。
function main() {
  var server = new grpc.Server();
  server.addProtoService(hello_proto.Greeter.service, serviceImplObj);
  server.bind('0.0.0.0:50051', grpc.ServerCredentials.createInsecure());
  server.start();
}
//下面是client的获取远程服务的代码
function main() {
  var proxy = new hello_proto.Greeter('localhost:50051',
                                      grpc.credentials.createInsecure());
  ..
}
```

RPC: 典型的部署方式

- RPC服务端是将需要发布的service注册到server中;
- 然后server再用IP:PROT发布到网络上。

RPC: Remote Procedure Call

- 接口风格
- 接口描述语言
- 协议
- 数据传输总线
- 异常和错误处理
- 典型的部署方式
- **其他的特点以及衍生的思考**

RPC:其他的特点以及衍生的思考

- 现代RPC在解耦方面已经做出了很多努力（IDL，基于TCP/IP协议，服务发布和发现）。
- 但仍然遗留有很多耦合高的地方:
- RPC的IDL设计往往会考虑主流语言的特性（双刃剑），而IDL的设计也需要小心，任何不兼容的更新都会让以往的代码失效;
- RPC传输协议过于复杂，难以维护，支持的语言往往也比较有限，新语言的支持需要实现整个RPC传输协议（gRPC只支持一些主流语言）;
- 基于TCP/IP：服务发布和获取使用 IP:port 。

REST: RESTful API

- **接口风格**
- 接口描述语言
- 协议
- 数据传输总线
- 异常和错误处理
- 典型的部署方式
- 其他的特点以及衍生的思考

REST:接口风格

- RESTful API ~= URL + HTTP Method + 数据表示

- http://www.nmc.cn/f/rest/real/58367
- Http Method = GET
- 数据表示 json

- 从nmc.cn中请求real服务，查询上海（编号为58367）的实时天气数据，数据使用json表示。

REST:接口风格

- RESTful API ~= **URL** + HTTP Method + 数据表示

- URL的设计可以很灵活:
- http://www.nmc.cn/f/rest/real/58367
- http://www.nmc.cn/f/rest/aqi/58367
- http://www.nmc.cn/f/rest/real?citycode=58367
- https://apis.map.qq.com/jsapi?qt=geoc&addr=上海&output=jsonp
- `/animals?zoo_id=3` 或 `/zoo/3/animals`
- https://developer.github.com/v3/media

REST:接口风格

- RESTful API ~= URL + **HTTP Method** + 数据表示
- 数据操作不仅有查询，通常可以分类为CRUD（增删改查），RESTful API使用不同的HTTP Method来表达CRUD：
- GET　　：查询数据
- POST　：增加数据
- PUT　　：数据更改
- DELETE ：数据删除
- （REST 假设人们只有这4种操作，但是够用了）

REST:接口风格

- 在常见的编程语言中CRUD一般用不同的名字表达：
- `Dict: get(key, val), add(key, val), set(key, val), remove(key), clearAll()`

- RESTful API不建议大家在URL中指示动作：

```
/dict/dictName/get-val?key=xxx
/dict/dictName/add-val?key=xxx   在 HTTP BODY 中加上 val 数据
/dict/dictName/set-val?key=xxx   在 HTTP BODY 中加上 val 数据
/dict/dictName/remove?key=xxx
/dict/dictName/clear-all
```

- 而用HTTP Method表达动作：

```
/dict/dictName?key=xxx    Method=GET
/dict/dictName?key=xxx    Method=POST    在 HTTP BODY 中加上 val 数据
/dict/dictName?key=xxx    Method=PUT     在 HTTP BODY 中加上 val 数据
/dict/dictName?key=xxx    Method=DELETE
/dict/dictName            Method=DELETE  不指定key，表示删除所有数据
```

REST:接口风格

- RESTful API ~= URL + HTTP Method + **数据表示**

- RESTful API不仅假设数据操作只有 CRUD 这4种,
- 还假设数据传输类型只有 “JSON”, “XML”, “JPEG” 等少数几种。

- 在HTTP HEADER中指定数据表示
- `Content-Type: application/json; charset=UTF-8`

- RPC可以允许用户自定义数据表示:

```
// 节选自之前的gRPC IDL的代码
message HelloRequest {
  string name = 1;
}
```

REST: RESTful API

- 接口风格
- **接口描述语言**
- 协议
- 数据传输总线
- 异常和错误处理
- 典型的部署方式
- 其他的特点以及衍生的思考

REST: 接口描述语言（略）

- 使用URL+HTTP就可以描述一个RESTful API，不需要专门的接口描述语言。

- RPC将接口描述保存在一个文本文件中：
- gRPC将一个IDL实例保存在".proto"文件中。

- REST可以将接口描述记录在开发者文档中；
- 也可以用Hypermedia发布到Web上（被称为HATEOAS）。

REST: 接口描述语言（略）

- Github的hypermedia：https://api.github.com/

```
{
  ...,
 "gists_url": "https://api.github.com/gists{/gist_id}",
  ...
}
```

REST: RESTful API

- 接口风格
- 接口描述语言
- **协议**
- 数据传输总线
- 异常和错误处理
- 典型的部署方式
- 其他的特点以及衍生的思考

REST: 协议

- 使用HTTP作为应用层协议，一般的程序语言都有完善的HTTP库。
- HTTP协议除了可以表达 Method，数据表示外，还有其他功能：
- Versioning：`X-GitHub-Media-Type: github.v3`
- Client可以在HTTP Request中指定期望得到的数据表示：
- `Accept: application/vnd.github.VERSION.html+json`

REST: RESTful API

- 接口风格
- 接口描述语言
- 协议
- **数据传输总线**
- 异常和错误处理
- 典型的部署方式
- 其他的特点以及衍生的思考

REST: 数据传输总线（略）

- 一般的程序语言都有HTTP的库：

- ```
  jQuery.post(url,  // 指定url
              data, // jQuery会自动将其翻译为某种数据表示
              success(data, textStatus, jqXHR), //异步处理函数
              dataType) // 指定期望返回的数据表示
  ```
  ```

REST: RESTful API

- 接口风格
- 接口描述语言
- 协议
- 数据传输总线
- **异常和错误处理**
- 典型的部署方式
- 其他的特点以及衍生的思考

REST: 异常和错误处理

- 根据HTTP Status Code处理异常。
- 200表示成功，403表示权限验证失败

- ```
  jQuery.post(url, data,
                success(data, textStatus, jqXHR),
                dataType)
  ```
  ```

REST: RESTful API

- 接口风格
- 接口描述语言
- 协议
- 数据传输总线
- 异常和错误处理
- **典型的部署方式**
- 其他的特点以及衍生的思考

REST: 典型的部署方式（略）

- 可以像Apache那样自己处理http请求；
- 也可以像nginx那样使用反向代理，将http请求转发给其他的服务。
- Spring Boot构建完毕后可以直接作为一个HTTP服务器启动。

- 任何http服务器都可以部署RESTful API，选择也很多。

REST: RESTful API

- 接口风格
- 接口描述语言
- 协议
- 数据传输总线
- 异常和错误处理
- 典型的部署方式
- **其他的特点以及衍生的思考**

REST: 衍生思考

RPC

- 发布在Internet上，使用IP:port获取服务；
- IDL在事实上会考虑特定语言的特性；
- 协议复杂，难以维护和扩展；
- RPC技术发展可能不稳定，有可能出现不兼容的改变。

- RPC协议设计通常会选择性能很高的方案；
- 与实际代码亲和力高，开发效率高；
- IDL能自定义数据类型。

RESTful API

- 用web发布，使用URL获取服务；
- IDL不考虑任何特定语言的特性；
- 协议简单，支持广泛；
- HTTP协议很稳定，基本不会有不兼容的改变。

- HTTP协议性能较差。
- 各种http库接口比较复杂，rest数据表示单调，开发效率低。

REST: 衍生思考

- 虽然HTTP Method，数据表示相对于RPC来说比较简陋；
- 实际上是在引导大家设计出简单的API。

- 使用RESTful API，人们必须将一个复杂的服务拆分为很多简单的服务。简单的服务有利于构建出高并发的服务架构。

- REST还引导人们将各种数据服务实体表示成一个资源，然后用RESTful API发布到Web中（ROA：面向资源的架构）。

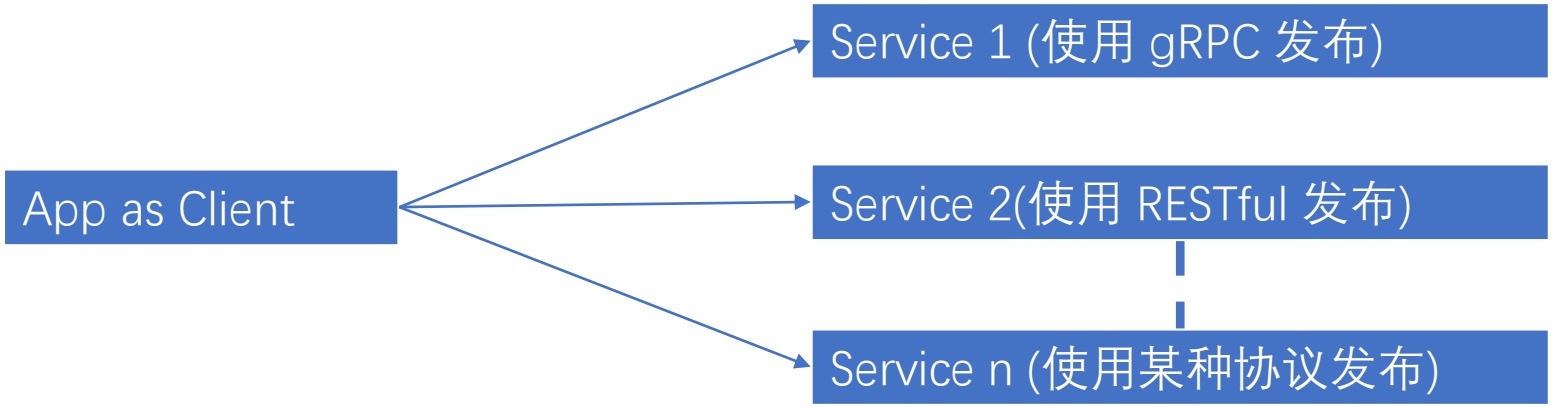
REST: 衍生思考

- GraphQL：尝试定制化服务器返回的数据表示。

- Github: https://developer.github.com/v4/
- (不过我们在使用过程中遇到了一些麻烦)

SOA: Service Oriented Architecture

- 一般的App会用到很多服务



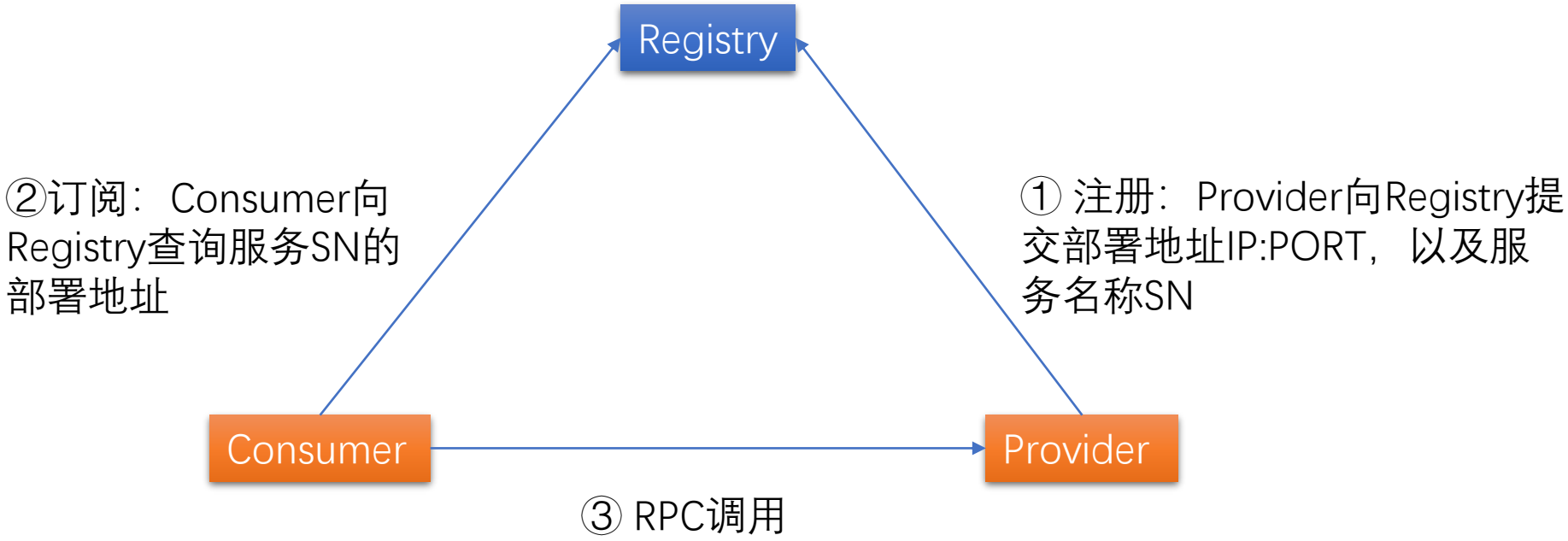
- 各个服务可能使用不同的协议;
- 即使协议相同，接口描述确实未知的;
- 服务获取的方式不同;

SOA: Service Oriented Architecture

- 同时服务发布者也有很多难以实现的需求:
- 性能（分布式，负载均衡）；
- 扩展性（随时可以升级服务）；
- 伸缩性（可以随时向服务集群中加入新的机器）；
- 可用性（容错，快速重启）；
- 安全性（访问控制）。

SOA: 发布与发现

- Registry将一个RPC服务编码为一个URL，Consumer使用RESTful API获取这个服务的IP:PORT地址，然后才进行RPC。



SOA: 发布与发现

DNS

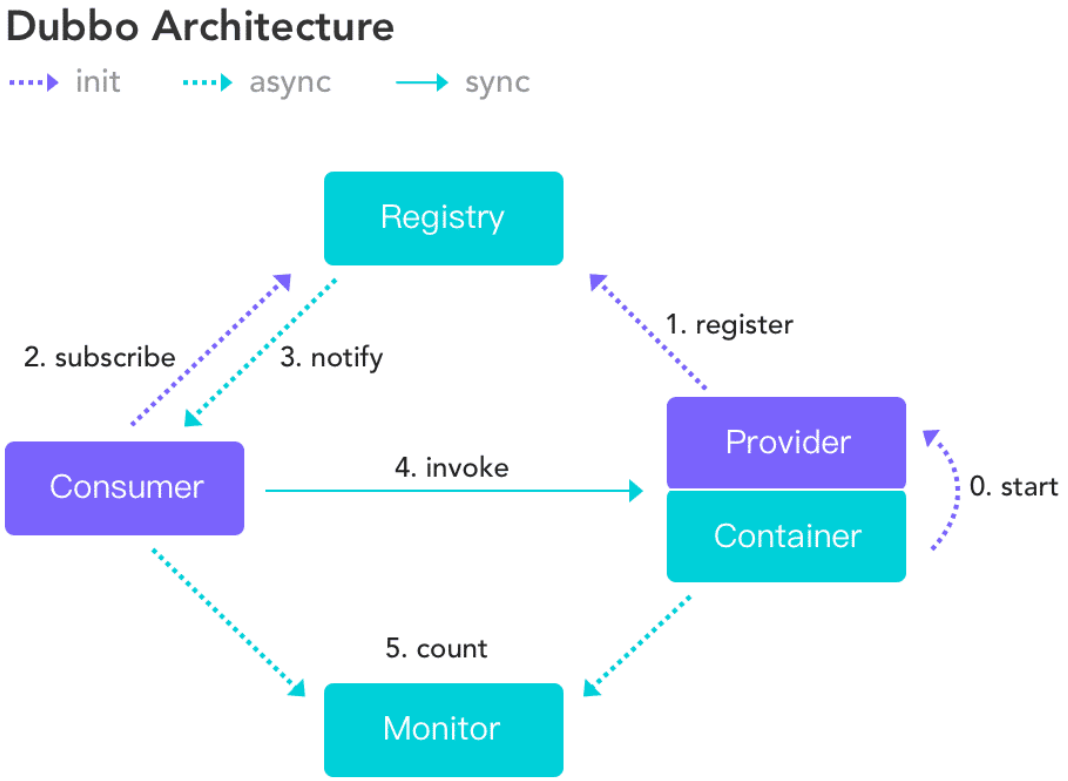
- 将domain name翻译成一系列可能的IP:PORT;
- DNS能返回某个domain name所支持的所有应用层协议;
- DNS可以做到分区服务（twitter在欧洲的服务器地址与在美国的服务器地址不同）。

Registry

- 将service name翻译成服务部署的地址;
- Registry可以提供某个服务的传输层协议和接口描述（SOAP与WSDL）;
- Registry也可以返回多个等同服务的部署地址（可以做负载均衡）。

SOA: 发布与发现

- Dubbo允许service注册多个Provider;
- Monitor监控Provider的访问情况;
- Consumer可以根据Dubbo的Monitor服务选择合适的Provider;
- 对于负载高的Provider, Monitor可以拦截新的请求（熔断）;

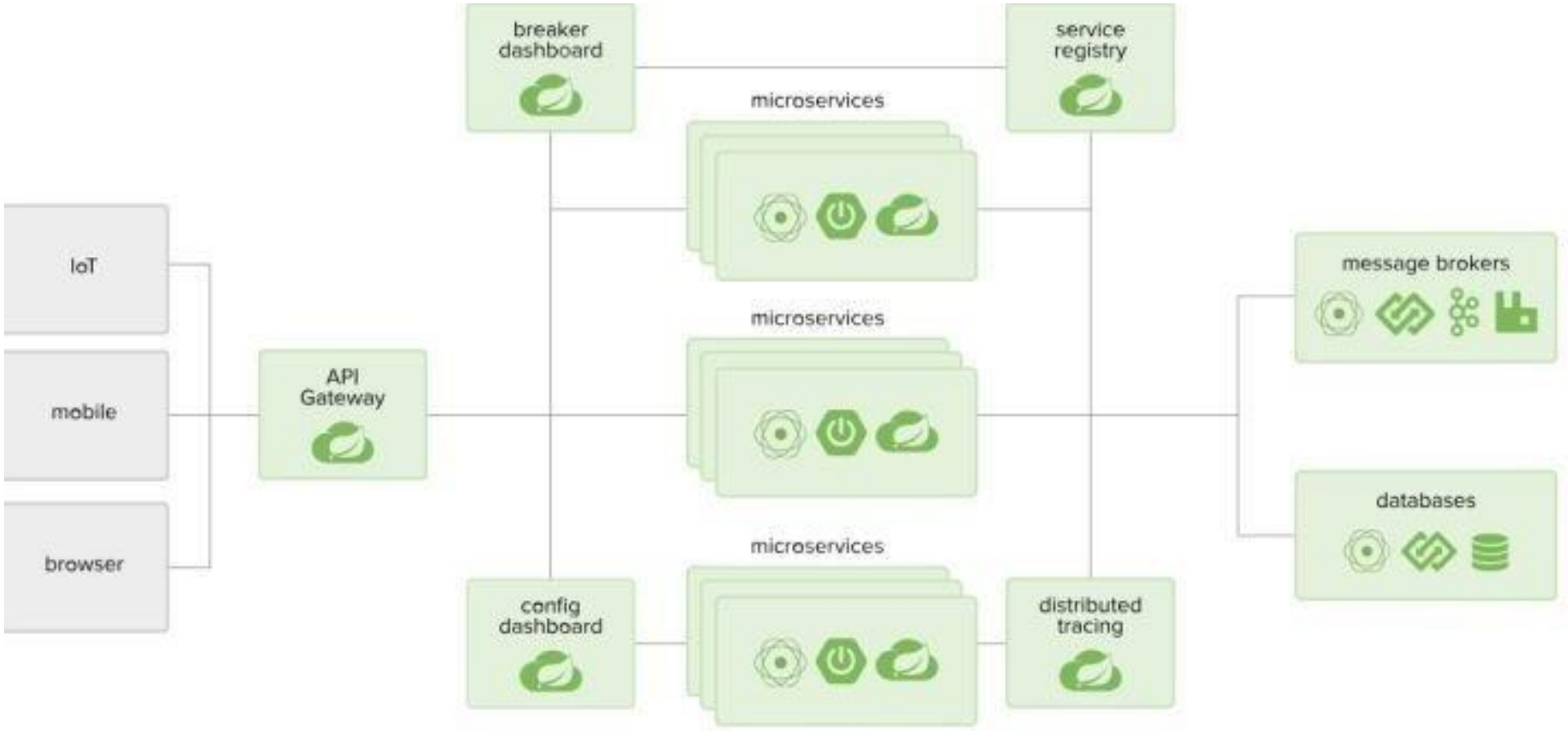


Microservice

- SOA使用Registry服务将众多的service联系起来;
- Dubbo则又从service中细分了Monitor服务;
- 我们可以更进一步拆分服务，把某些需要执行的特定步骤做成一个单独的服务;

Microservice

- Spring Cloud



Spring Cloud Gateway（反向代理）

```
// Gateway反向代理，将用户请求分发给其他服务，外部表现为RESTful
@RestController
@SpringBootApplication
public class GatewaySampleApplication {
        @Value("${remote.home}")
        private URI home;

        @GetMapping("/test")
        public ResponseEntity<?> proxy(ProxyExchange<byte[]> proxy) throws Exception {
                return proxy.uri(home.toString() + "/image/png").get();
        }
}
```

Eureka (Registry)

```
@SpringBootApplication
@EnableEurekaServer
public class Application {
    public static void main(String[] args) {
        new SpringApplicationBuilder(Application.class)
                .web(true)
                .run(args);
    }
}
```

Spring Cloud

- Feign（服务整合器）：将多个service组合成一个新的RESTful API
- CircuitBreaker（熔断器）：对于负载过高的Provider自动拦截新的请求
- ConfigServer（分布式配置）：自动将配置文件的改动推送给各个服务
- Bus（服务总线）：分布式消息队列
- distributed sessions：分布式session存储

- 我们可以使用Spring Cloud全家桶快速构建微服务架构的web service。