

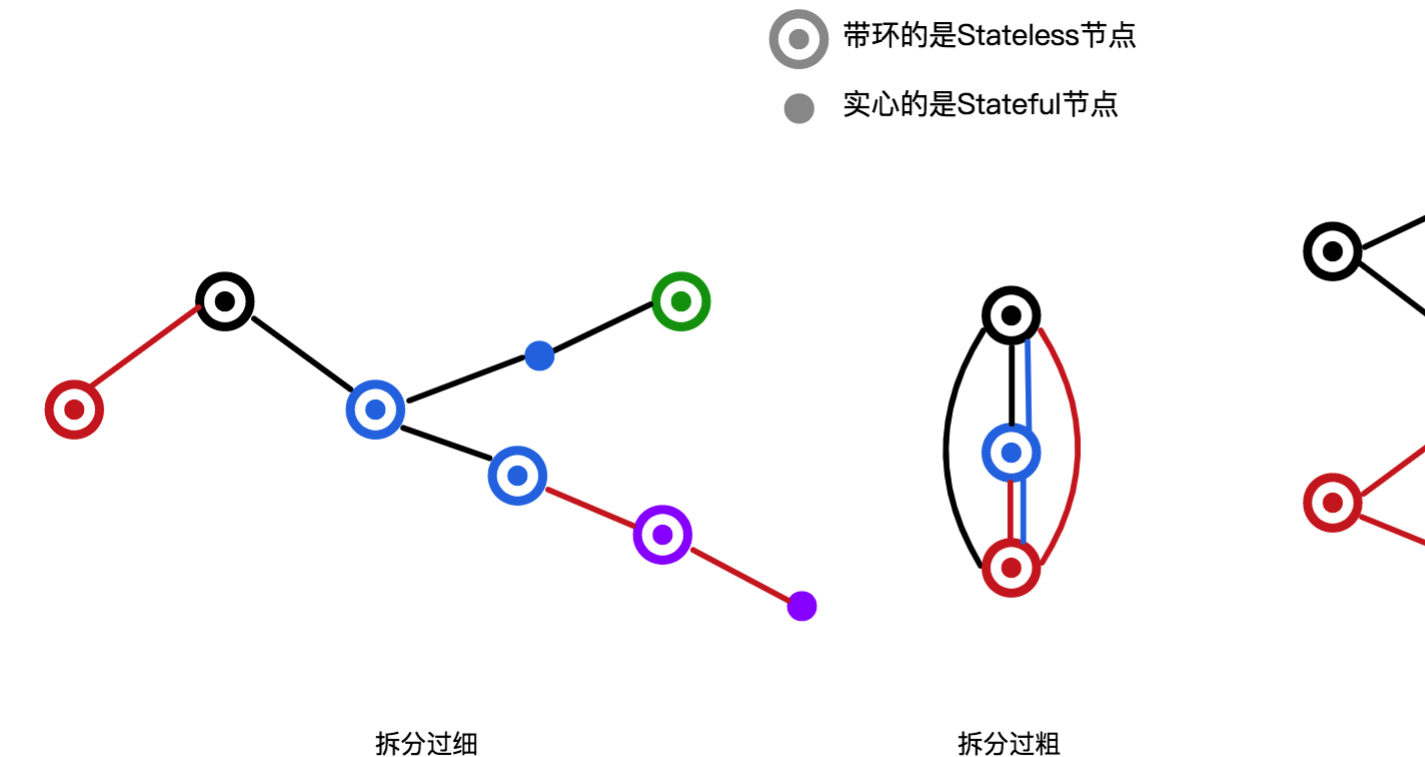
你好，我是秦粤。上一课中，我们学习了后端BaaS化的重要模块：微服务。现在我们知道微服务的核心理念就是先拆后合，拆解功能是为了提升我们功能的利用率。同步我们也了解了实现微服务的10要素，这10要素要真讲起来够单独开一门课的。如果你不熟悉，我向你推荐杨波老师的《微服务架构核心20讲》课程。

BaaS化的核心其实就是把我们的后端应用封装成RESTful API，然后对外提供服务，而为了后端应用更容易维护，我们需要将后端应用拆解成免运维的微服务。这个逻辑你要理解，这也是为什么我要花这么多篇幅给你谈微服务的关键原因。

上节课我们将“待办任务”Web服务的后端，拆解为用户微服务和待办任务微服务。但为什么要这样拆？是凭感觉，还是有具体的方法论？这里你可以停下来想想。

微服务的拆解和合并，都有一个度需要把握，因为我们在一拆一合之间，都是有成本产生的。如果我们拆解得太细，就必然会导致我们的调用链路增长。调用链路变长，首先影响的就是网络延迟，这个好理解，毕竟你路远了，可能“堵车”的地方也会变多；其次是运维成本的增加，调用链路越长，整个链条就越脆弱，因为其中一环出现问题，都会导致整个调用链条访问失败，而且我们排查问题也变得更加困难。

反过来看，如果我们拆解得太粗，调用链路倒是短了，但是这个微服务的复用性就差了，更别提因为高耦合带来的复杂且冗余的数据库表结构，让我们后续难以维护。我画了个图，你感受下。



拆之，领域驱动设计

那我们要合理地拆解微服务，应该怎么拆解呢？上节课其实我有提到，目前主流的解决方案就是领域驱动设计，也叫DDD[1]。DDD是Eric Evans在其2004年的同名书中提出来的一个思想，但一直仅仅局限在Java的圈子里，直到2014年，微服务兴起后大家才发现它可以指导微服务的拆分的，这才走进了大多数人的视野。用一句话简单总结，DDD就是一套方法论：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立拆分的领域模型。

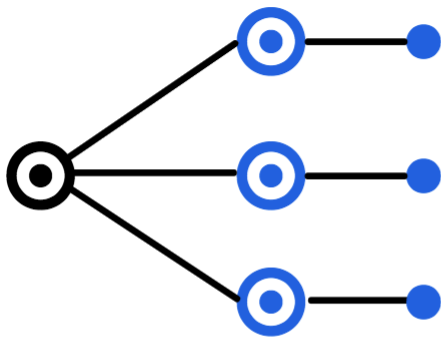
实际自己在使用微服务开发的过程发现,微服务整体应该是一个动态网络结构[2],随着业务的发展,这个网络结构也会发生变化。DDD能帮助我们前期分析出一个较好的网络结构,但实际上,我们更应该思考的是如何整体优化动态网络:减少核心节点,保护核心节点,降低网络深度等等。

怎么理解动态网络优化呢？我们可以做个思维实验：假设我们将所有的功能都拆解成微服务，任意的微服务节点之间都可以相互调用，调用越频繁它们之间的距离就越近。那么我们考虑一下，当我们网站的访问请求流量稳定后，我们整个微服务节点组成的网络状态是怎么样的？

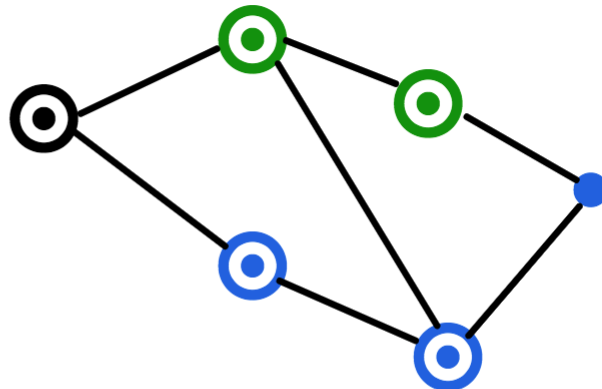
首先网络节点的相互制约总会让那些相互之间强依赖的、高耦合的节点，越走越近，最后聚集成一团节点。其次那些跟业务逻辑无关的节点，逐渐被边缘化，甚至消失。我们看这些聚集成团的节点，如果团里的点聚合太近了，其实是不适合拆分的，它们整体应该作成成一个微服务。等这些节点太近的团合并成一个微服务节点后，我们再看那些聚集在一起、又不太近的节点就是一个个微服务了。

所以，我们在启动项目时，不用太过纠结应该如何去拆解微服务。而应该持续关注，并思考每个微服务节点的合理性。就像看待动态网络一样，持续地调整优化，去除核心节点。最终它会伴随你业务的发展阶段，达到各个阶段的稳定动态网络结构。

就像我们上节课“待办任务”Web服务一样，我们可以先简单地将我们的项目后端分为：用户微服务和待办任务微服务。当然这里我们目前的业务太简单了，用DDD去分析，也是大材小用。随着我这个项目的业务发展，我们添加的功能会越来越多。让微服务根据业务一起成长演变就可以了。这并不是说我们就放任微服务不管了，而是从整体网络的角度思考，去看我们的微服务如何演进。



静态网络
基于机器IP的网络



动态网络
基于服务的网络

合之，Streaming

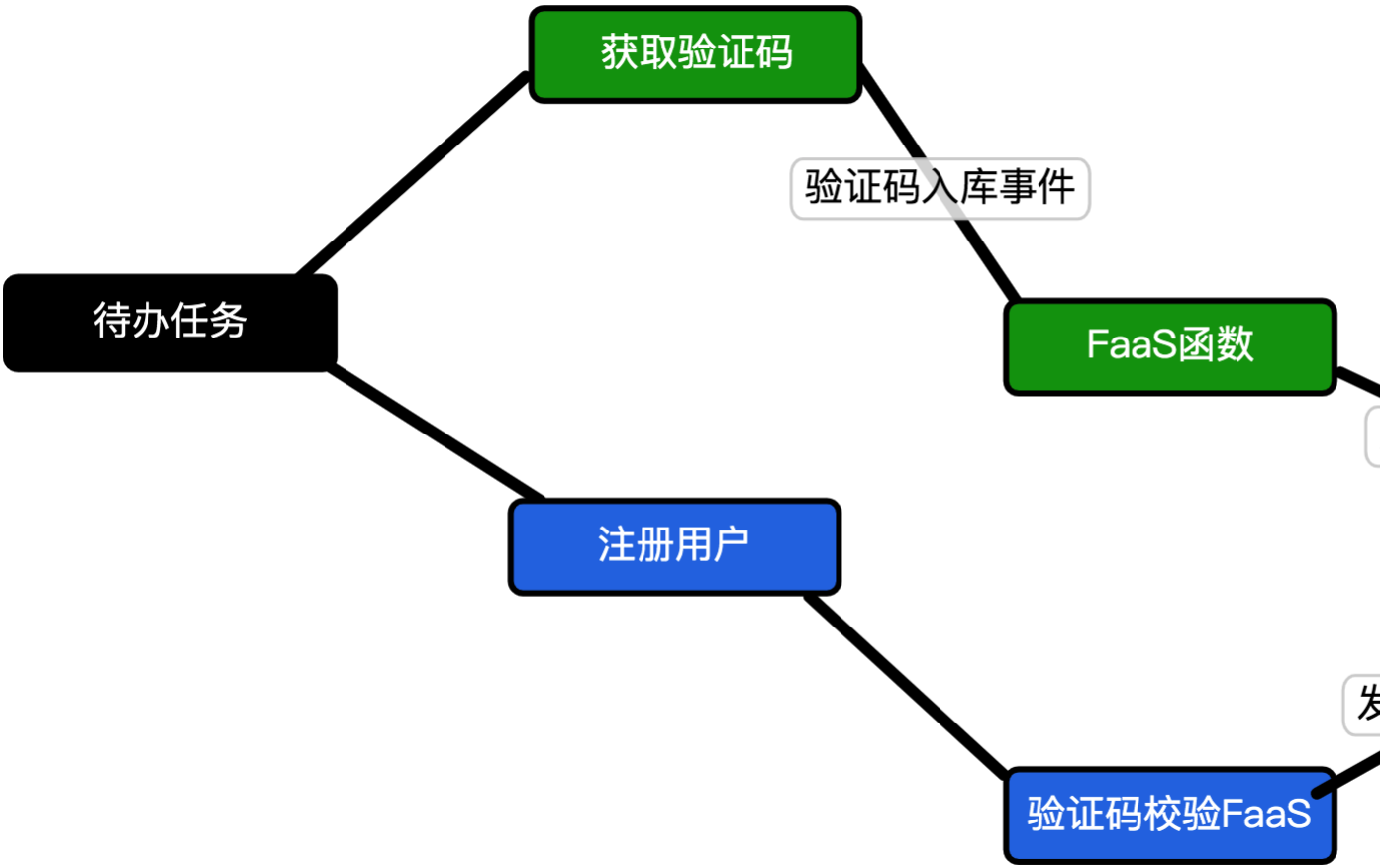
看完拆解，我们再看合并。合并呢，换个高大上的词其实就是前面课程中提到的编排。目前为止，我们整个“待办任务”Web应用架构的设计基本完成了，而且所有节点都是Stateless的了。变成Stateless节点后，其实对于前端的同学来说，一点都不陌生，比如React的单向数据流中的State也要求我们Immutable，Immutable其实就是Stateless。

我们上面已经看到了，拆解后的架构是个动态网络，那我们应该怎么合并或者编排呢？当然你像SFF那样通过传统的函数，将每个HTTP数据的请求结果通过数组或对象加工处理，再将这些结果返回也是可以的。但我在这里想向你介绍另外一种编排思路，工作流。



我们可以将用户的请求想象成我们的呼吸系统，我们的肺就是SFF，而微服务和FaaS节点就是需要氧气的各个器官。我们吸一口气，氧气进入肺部，血液循环将氧气按顺序流经我们每个器官，这就是请求链路。每个器官一接收到新鲜血液，就会吸取氧气返回二氧化碳，最终血液循环将二氧化碳带到肺部呼出，这个就是数据返回链路。我们的各个器官，就被请求链路通过新鲜血液到来的这个事件串联起来了，这个就是事件流，也就是用一个事件去串联FaaS或微服务。

现在我们用[\[第3课\]](#)讲的，PHP发邮件改造一下，举个例子。当用户注册时，我们完全可以将用户的信息和注册验证码存入数据库；PHP发邮件的FaaS触发器改为数据库插入新记录触发事件；用户从邮箱验证获取验证码，把验证码写到输入框后，点击验证，则是另一个HTTP触发器，触发FaaS函数校验验证码通过，修改数据库注册成功，并且返回302跳转到登录成功页面。具体流程可参考下图：



当然现在这个解决方案也有成熟对应的云BaaS服务：Serverless工作流[3]。

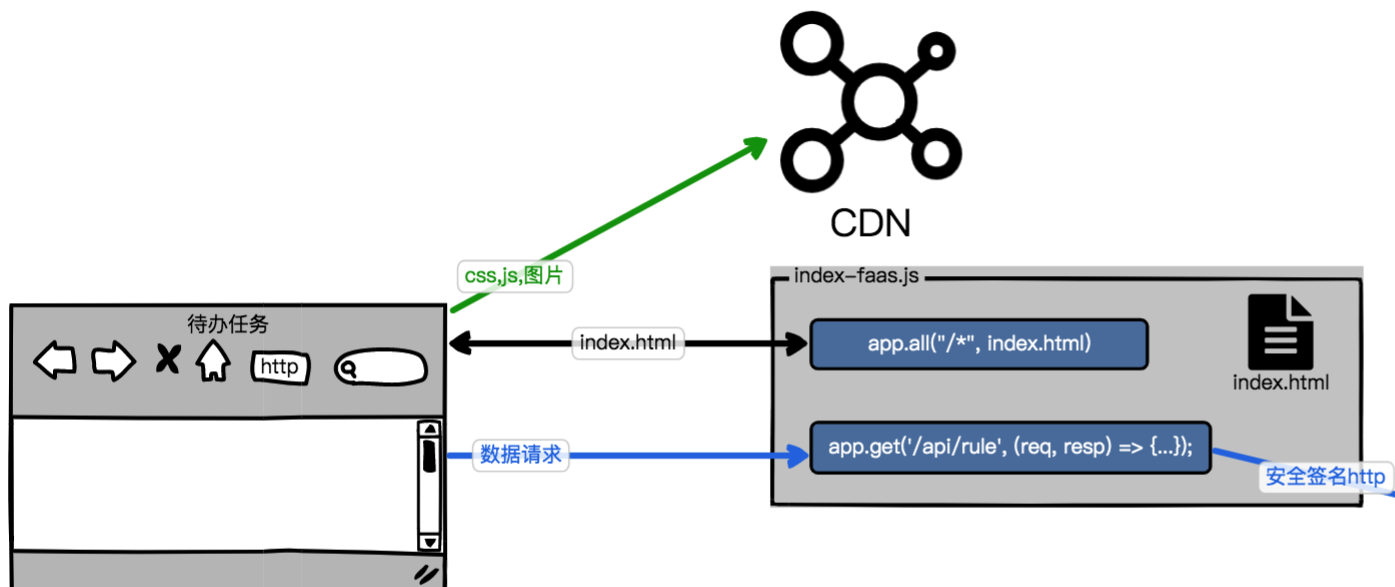
安全门神

理解了拆合的思想，我们就可以将目前“待办项目”的架构再演进一下：静态文件我们用CDN托管，前端项目只负责域名支撑和index.html，剩下的请求直接访问FaaS微服务。这时候，我估计你会问，咱们数据的安全性如何保障呢？是的，到目前为止，我们的FaaS都一直在用匿名模式访问，完全没有任何安全防护可言，也就是说目前我们FaaS服务的接口一直都在互联网上“裸奔”。

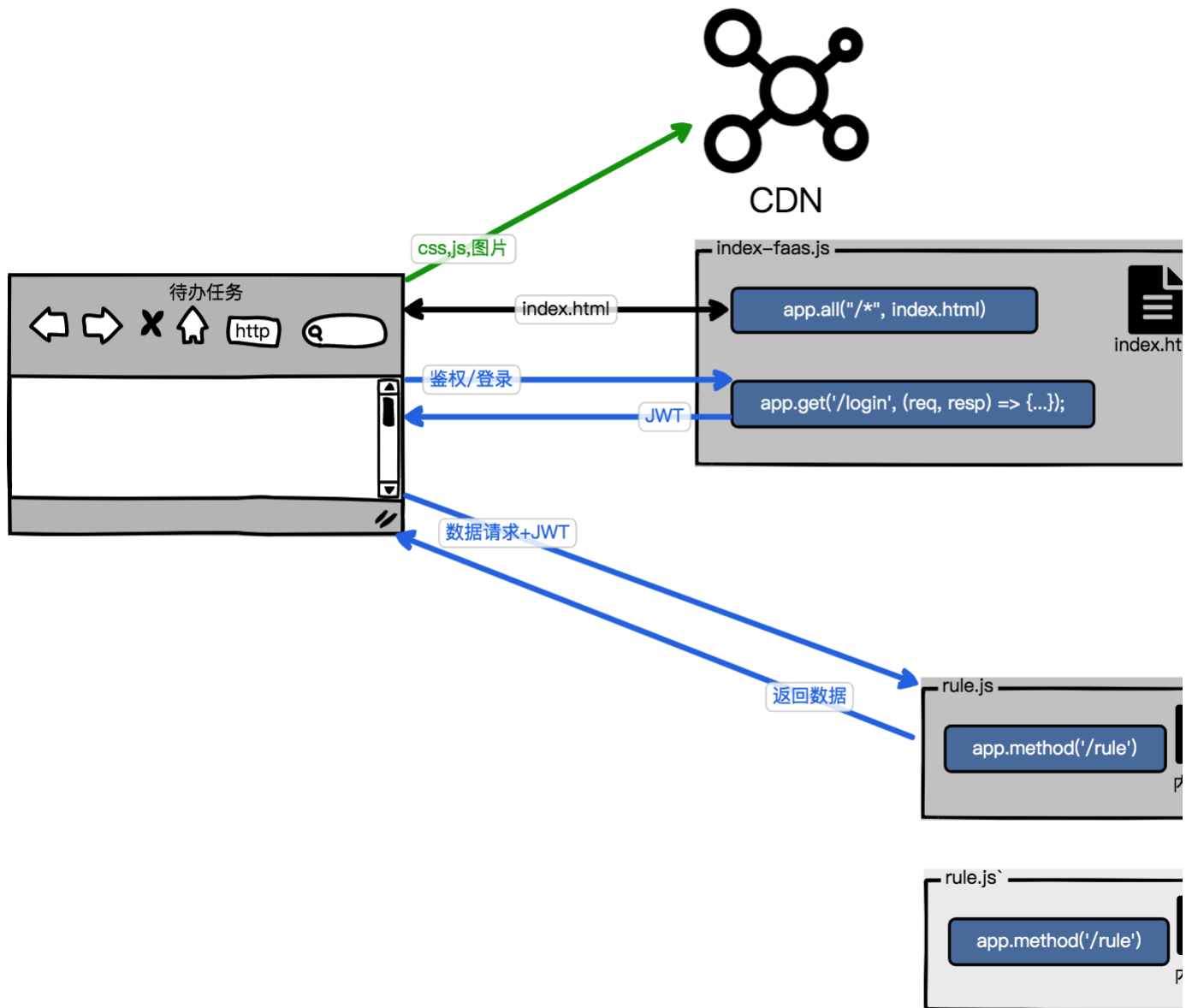
鉴权

其实，FaaS提供的安全防护通常是放在触发器上的。触发器的授权类型或认证方式我们可以设置为：匿名anonymous或函数function。匿名方式就是不需要签名认证，匿名的用户也能访问；而函数方式，则需要签名认证[4]，这个签名认证的算法，参数需要用到我们账户的访问密钥ak/sk[5]，ak/sk相当于我们云账户的银行卡密码，这么重要的账户信息，我们只能限定在服务端使用，前端代码里绝对不可以出现。

也就是说，我们只能在服务端使用函数安全认证方式。如果是这种方案，我们的“待办任务”架构就演进成下图这样了。



那有没有针对匿名认证方式的安全策略呢？当然有，这里我们同样需要借鉴一下微服务的鉴权设计：JSON Web Token，简称JWT[6]。JWT简单来说，就是将用户身份信息和签名信息，一起传到客户端去了。用户在请求数据时，带上这个JWT，服务端通过校验签名确定用户身份。JWT存在于客户端，JWT验证只需要通过服务端的sk和算法验证签名。同样，我画了张图，以帮助你理解。



要解决后端互调的安全性，我们用VPC或IP白名单，都很容易解决。比较难处理的是前后端的信任问题，JWT正好就提供了一种信任链的解决思路。当然，关于鉴权也有一些云服务商推出了一些更加安全易用的BaaS服务，例如AWS的IAM和Cognito[7]。

安全性是我们考虑架构设计时重要的一环，因为安全架构设计的失败，会直接导致我们资产的损失。鉴权是识别用户身份，防止用户信息泄漏和恶意攻击使用的。但根据我统计的数据，我们在日常99%的问题，都发生在新版本上线的环节。

那我们该怎么稳定持续地快速迭代，发布新版本上线呢？我们可以回想一下[第1课]，小程序和小程序的例子，小程序最后实现NoOps后，小程序则只要将代码合并到指定分支就可以发布上线了。那现实中，这点该怎么实现呢？

当我们的项目Serverless化以后，代码的质量变得尤为重要。你可以想想，Serverless化之前，你不小心上线了一个bug，影响的范围最大也就只有一个应用。但是Serverless化之后，如果是核心节点发布了严重的bug上线，那么影响的范围就是所有依赖它的线上应用了。

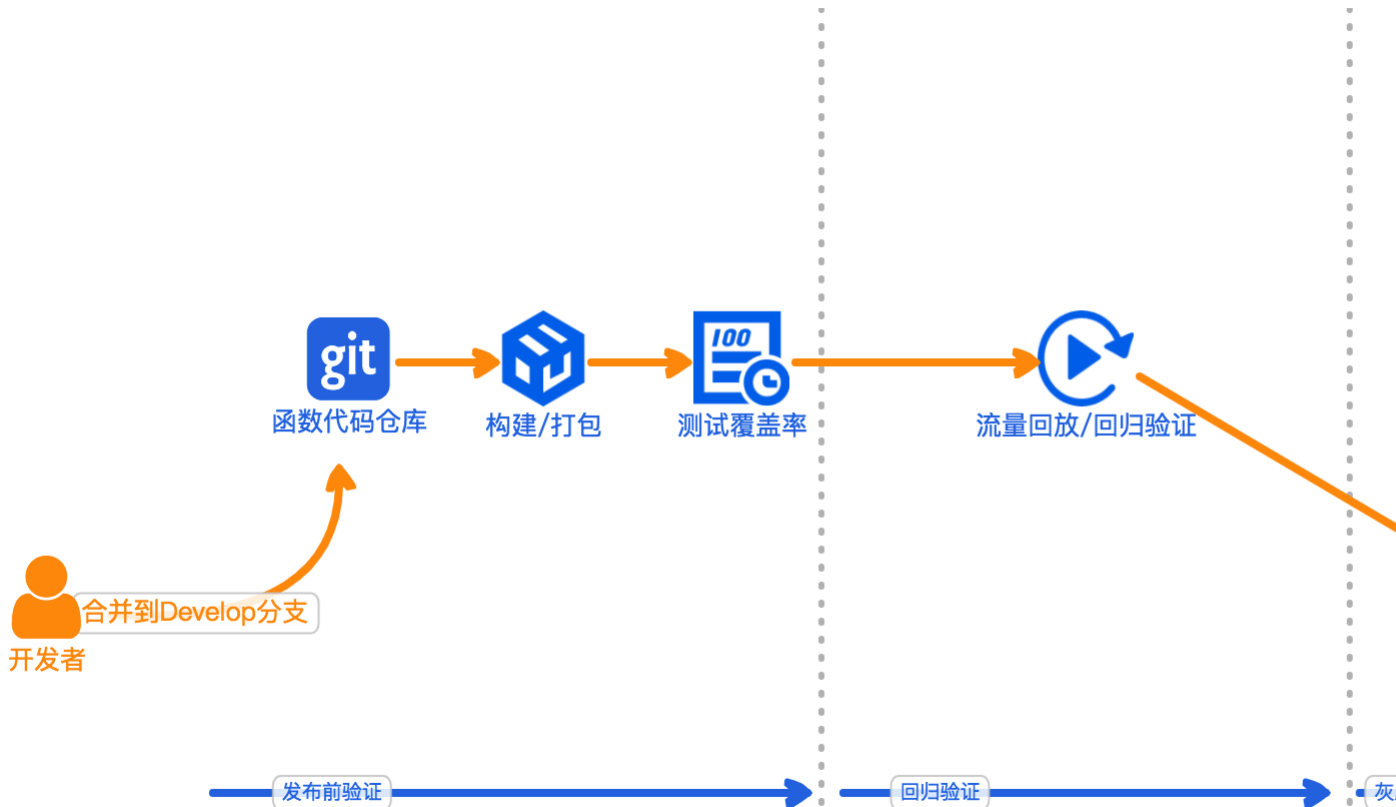
不过，你也不用太担心，微服务和FaaS都具备快速独立迭代的能力。以前我们一个应用的迭代周期通常要一周到两周。但对于Serverless化后的应用来说，每个节点借助独立运维的特性，可以随时随地地发布上线。

综上，我们知道了，微服务和FaaS都是快速迭代的，修复问题很快，但我们也不能每次都等问题出现，再去依赖这个能力呀。有没有什么办法可以提前发现问题，保证我们又快又稳？目前软件工程的最佳做法就是代码流水线的发布管道。

发布管道

发布管道的流水线主要有3个部分：

1. 代码发布前的验证，代码测试覆盖率CI/CD；
2. 模拟流量回归测试通过，发布到灰度环境；
3. 代码正式上线，灰度环境替换正式环境。流水线的每个节点产生的结果，都会作为下一个节点必要的启动参数。



我们先看看上图，我来解释下这个流程。

- 我们的代码合并到指定分支后，通常我会用Develop分支。
- Git的钩子就会触发后续的流水线，开始进入构建打包、测试流程。
- 测试节点做的事情就是跑所有测试Case，并且统计覆盖率。
- 覆盖率验证通过，代码实例用录制流量模拟验证。
- 模拟验证通过，发布代码实例到灰度环境。
- 线上根据灰度策略，将小部分流量导入灰度环境验证灰度版本。
- 在灰度窗口期，比如两个小时，灰度验证没有异常则用灰度版本替换正式版本；反之则立即丢弃这个灰度版本，止损。

这套流程，目前规模大一些的互联网公司发布流程基本都在这样跑，如果你不是很了解，可以自己尝试用我们介绍的Serverless工作流或者云服务商提供的工作流工具[8]动手搭建下。

在这套流程的基础上，很多企业为了追求更高的稳定性，还会设定环境隔离的流水线和安全卡口。比如隔离测试环境和线上环境，测试环境用来复现故障。每次代码进入发布管道，都必须先在测试环境跑通，跑通后安全卡口放行，才能进入线上环境的流水线。

总结

这节课，我们继续讲后端的BaaS化。我们再梳理一下这节课的重要知识点吧。

1. 如何拆解BaaS应用，我们学习了微服务的重要拆解思想DDD：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。另外我也介绍了另一种更适合初创企业的拆分思路：动态网络演进。
2. 拆解完之后，我们就要考虑合并。这里我们介绍了代码编排以外的另一种编排方式：事件流编排，它就是通过一个个事件顺序将我们的微服务或FaaS串联起来。
3. 为了解决拆解后，微服务之间的信任问题。我们先了解了FaaS触发器的安全方案：数字签名。还借鉴了微服务的鉴权做法JWT，将用户鉴权加密信息放在客户端，让鉴权服务变成Stateless。最后，为了让微服务又快又稳地发布版本，我们借鉴了微服务的发布管道：打造自动灰度流水线。

作业

这节课的作业就是我们JWT鉴权的“待办任务”Web应用，你来部署上线。

后端代码GitHub地址：<https://github.com/pusongyang/todolist-backend/tree/lesson06>

前端代码GitHub地址：<https://github.com/pusongyang/todolist-frontend>

演示预览地址：<http://lesson6.jike-serverless.online/list>

期待你的作业，如果今天的内容让你有所收获，也欢迎你把文章分享给身边的朋友，邀请他加入学习。

参考资料

- [1] https://en.wikipedia.org/wiki/Domain-driven_design
- [2] https://en.wikipedia.org/wiki/Dynamic_network_analysis
- [3] <https://www.aliyun.com/product/fnf>
- [4] <https://github.com/alibaba/aliyun-fc-nodejs-sdk/blob/master/lib/client.js?spm=a2c4g.11186623.2.15.16e016d7lo8NBQ#L840>
- [5] https://help.aliyun.com/document_detail/154851.html?spm=5176.2020520153.0.0.371a415dLXylZ
- [6] <https://jwt.io/>
- [7] https://docs.aws.amazon.com/zh_cn/cognito/latest/developerguide/what-is-amazon-cognito.html
- [8] <https://www.aliyun.com/product/yunxiao/devops?spm=5176.10695662.1173276.1.6c724a38akCjgo>

你好，我是秦粤。上一课中，我们学习了后端BaaS化的重要模块：微服务。现在我们知道微服务的核心理念就是先拆后合，拆解功能是为了提升我们功能的利用率。同步我们也了解了实现微服务的10要素，这10要素要真讲起来够单独开一门课的。如果你不熟悉，我向你推荐杨波老师的《微服务架构核心20讲》课程。

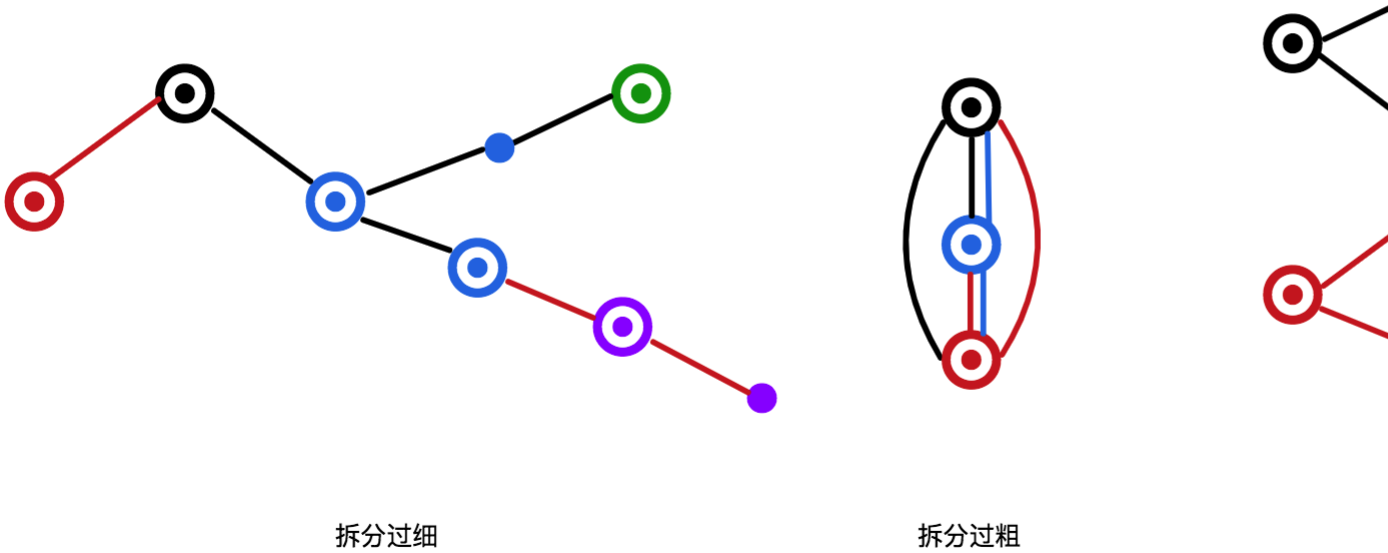
BaaS化的核心其实就是把我们的后端应用封装成RESTful API，然后对外提供服务，而为了后端应用更容易维护，我们需要将后端应用拆解成免运维的微服务。这个逻辑你要理解，这也是为什么我要花这么多篇幅给你谈微服务的关键原因。

上节课我们将“待办任务”Web服务的后端，拆解为用户微服务和待办任务微服务。但为什么要这样拆？是凭感觉，还是有具体的方法论？这里你可以停下来想想。

微服务的拆解和合并，都有一个度需要把握，因为我们在一拆一合之间，都是有成本产生的。如果我们拆解得太细，就必然会导致我们的调用链路增长。调用链路变长，首先影响的就是网络延迟，这个好理解，毕竟你路远了，可能“堵车”的地方也会变多；其次是运维成本的增加，调用链路越长，整个链条就越脆弱，因为其中一环出现问题，都会导致整个调用链条访问失败，而且我们排查问题也变得更加困难。

反过来看，如果我们拆解得太粗，调用链路倒是短了，但是这个微服务的复用性就差了，更别提因为高耦合带来的复杂且冗余的数据库表结构，让我们后续难以维护。我画了个图，你感受下。

带环的是Stateless节点
实心的是Stateful节点



拆之，领域驱动设计

那我们要合理地拆解微服务，应该怎么拆解呢？上节课其实我有提到，目前主流的解决方案就是领域驱动设计，也叫DDD[1]。DDD是Eric Evans在其2004年的同名书中提出来的一个思想，但一直仅仅局限在Java的圈子里，直到2014年，微服务兴起后大家才发现它可以指导微服务的拆分，这才走进了大多数人的视野。用一句话简单总结，DDD就是一套方法论：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。

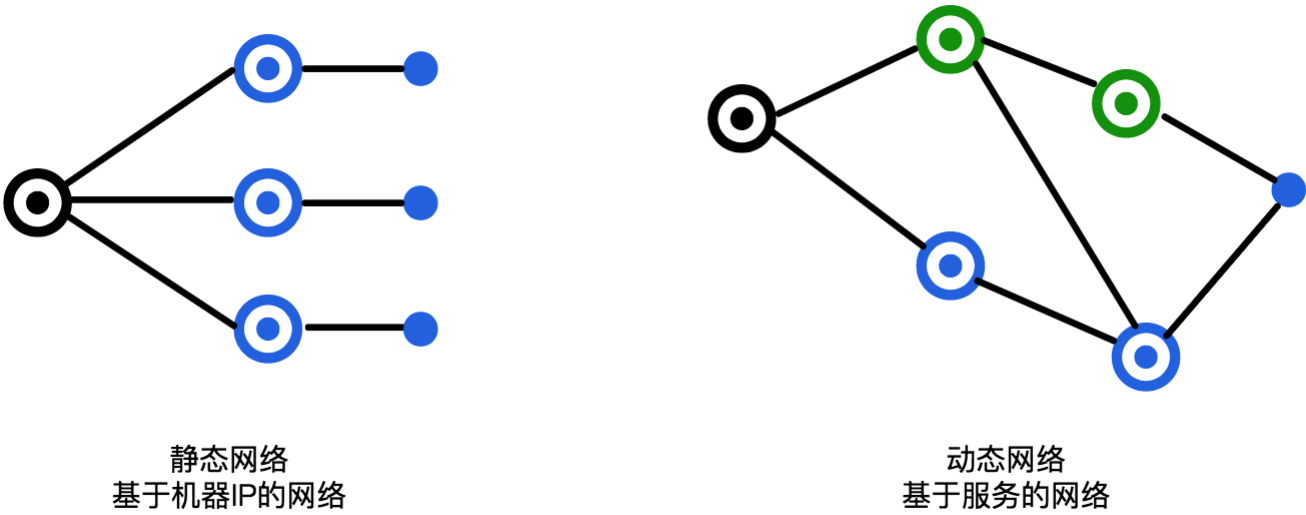
实际我自己在用微服务开发的过程发现，微服务整体应该是一个动态网络结构[2]，随着业务的发展，这个网络结构也会发生变化。DDD能帮助我们前期分析出一个较好的网络结构，但实际上，我们更应该思考的是如何整体优化动态网络：减少核心节点，保护核心节点，降低网络深度等等。

怎么理解动态网络优化呢？我们可以做个思维实验：假设我们将所有的功能都拆解成微服务，任意的微服务节点之间都可以相互调用，调用越频繁它们之间的距离就越近。那么我们考虑一下，当我们网站的访问请求流量稳定后，我们整个微服务节点组成的网络状态是怎么样的？

首先网络节点的相互制约总会让那些相互之间强依赖的、高耦合的节点，越走越近，最后聚集成一团节点。其次那些跟业务逻辑无关的节点，逐渐被边缘化，甚至消失。我们看这些聚集成团的节点，如果团里的点聚合太近了，其实是不适合拆分的，它们整体应该作成一个微服务。等这些节点太近的团合并成一个微服务节点后，我们再看那些聚集在一起、又不太近的节点就是一个微服务了。

所以，我们在启动项目时，不用太过纠结应该如何去拆解微服务。而应该持续关注，并思考每个微服务节点的合理性。就像看待动态网络一样，持续地调整优化，去除核心节点。最终它会伴随你业务的发展阶段，达到各个阶段的稳定动态网络结构。

就像我们上节课“待办任务”Web服务一样，我们可以先简单地将我们的项目后端分为：用户微服务和待办任务微服务。当然这里我们目前的业务太简单了，用DDD去分析，也是大材小用。随着我这个项目的业务发展，我们添加的功能会越来越多。让微服务根据业务一起成长演变就可以了。这并不是说我们就放任微服务不管了，而是从整体网络的角度思考，去看我们的微服务如何演进。



合之，Streaming

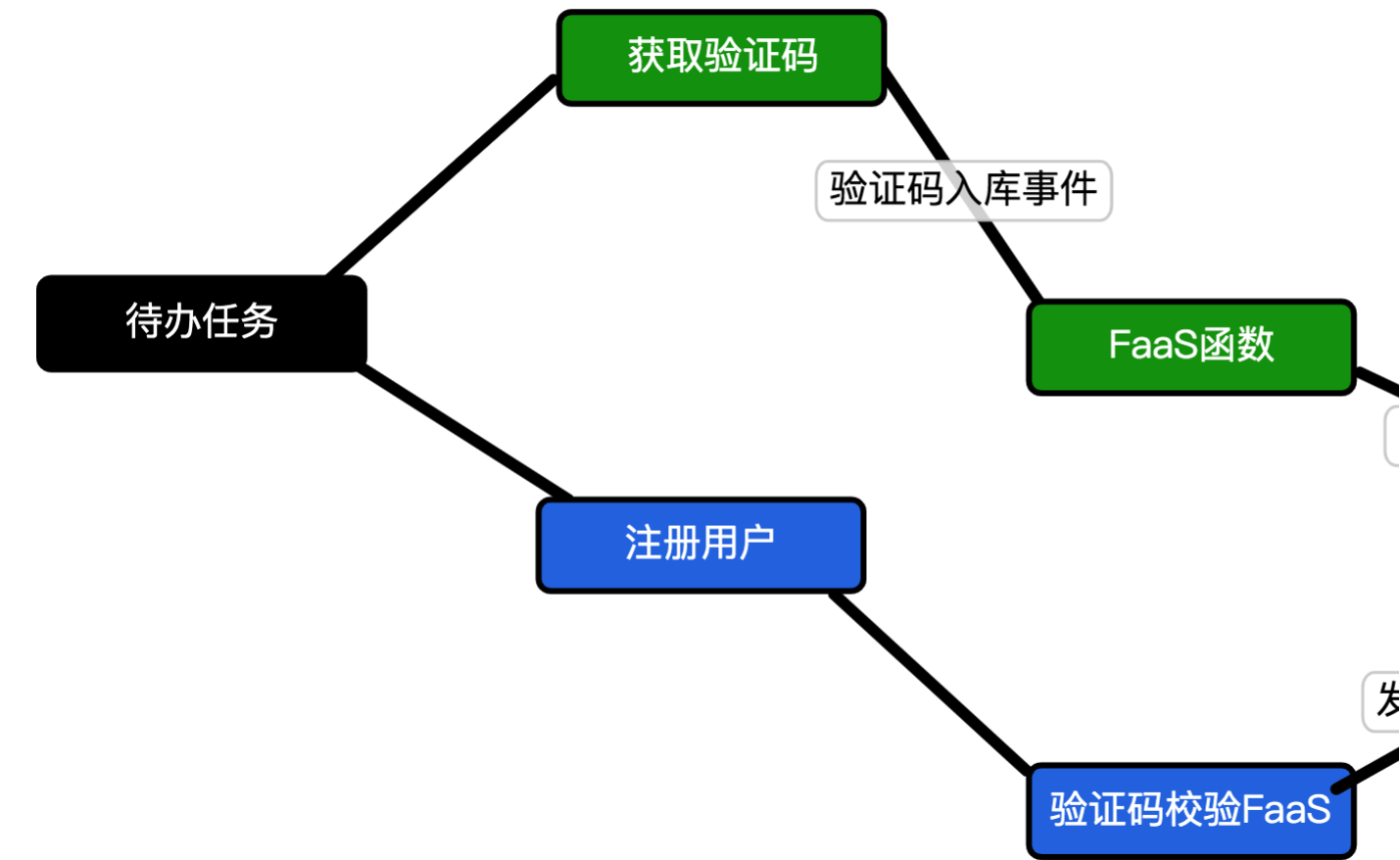
看完拆解，我们再看合并。合并呢，换个高大上的词其实就是前面课程中提到的编排。目前为止，我们整个“待办任务”Web应用架构的设计基本完成了，而且所有节点都是Stateless的了。变成Stateless节点后，其实对于前端的同学来说，一点都不陌生，比如React的单向数据流中的State也要求我们Immutable，Immutable其实就是Stateless。

我们上面已经看到了，拆解后的架构是个动态网络，那我们应该怎么合并或者编排呢？当然你像SFF那样通过传统的函数，将每个HTTP数据的请求结果通过数组或对象加工处理，再将这些结果返回也是可以的。但我在这里想向你介绍另外一种编排思路， workflow。



我们可以将用户的请求想象成我们的呼吸系统，我们的肺就是SFF，而微服务和FaaS节点就是需要氧气的各个器官。我们吸一口气，氧气进入肺部，血液循环将氧气按顺序流经我们每个器官，这就是请求链路。每个器官一接收到新鲜血液，就会吸取氧气返回二氧化碳，最终血液循环将二氧化碳带到肺部呼出，这个就是数据返回链路。我们的各个器官，就被请求链路通过新鲜血液到来的这个事件串联起来了，这个就是事件流，也就是用一个事件去串联FaaS或微服务。

现在我们用[第3课]讲的，PHP发邮件改造一下，举个例子。当用户注册时，我们完全可以将用户的信息和注册验证码存入数据库；PHP发邮件的FaaS触发器改为数据库插入新记录触发事件；用户从邮箱验证获取验证码，把验证码写到输入框后，点击验证，则是另一个HTTP触发器，触发FaaS函数校验验证码通过，修改数据库注册成功，并且返回302跳转到登录成功页面。具体流程可参考下图：



当然现在这个解决方案也有成熟对应的云BaaS服务：Serverless工作流[3]。

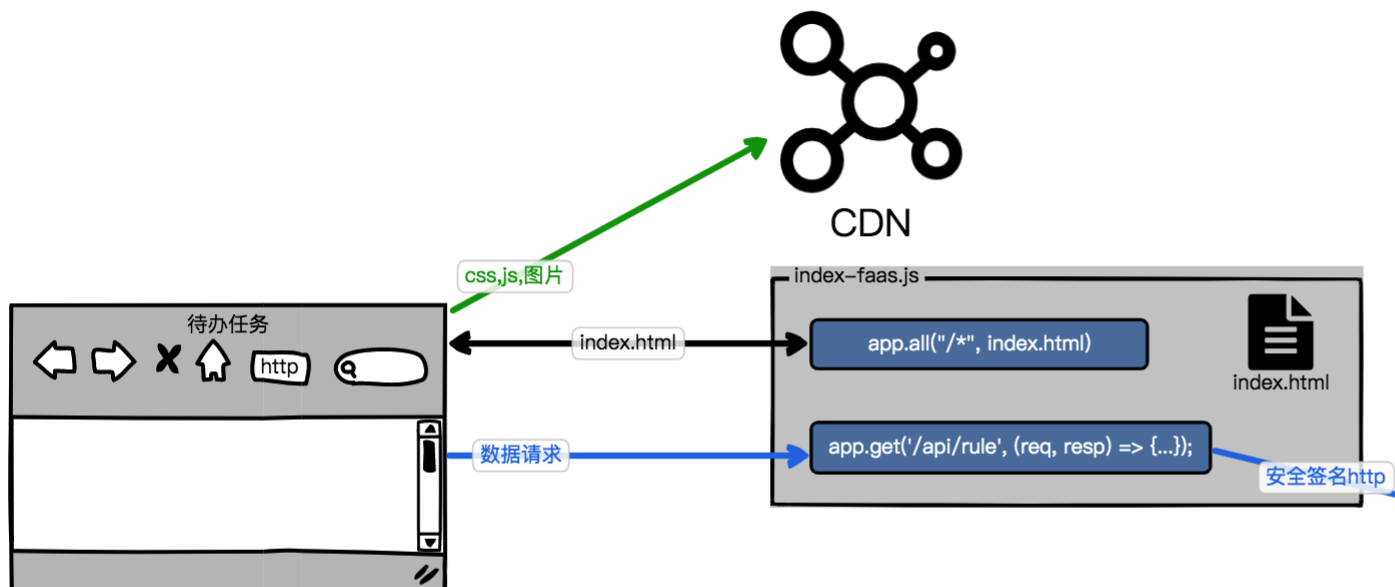
安全门神

理解了拆合的思想，我们就可以将目前“待办项目”的架构再演进一下：静态文件我们用CDN托管，前端项目只负责域名支撑和index.html，剩下的请求直接访问FaaS微服务。这时候，我估计你会问，咱们数据的安全性如何保障呢？是的，到目前为止，我们的FaaS都一直在用匿名模式访问，完全没有任何安全防护可言，也就是说目前我们FaaS服务的接口一直都在互联网上“裸奔”。

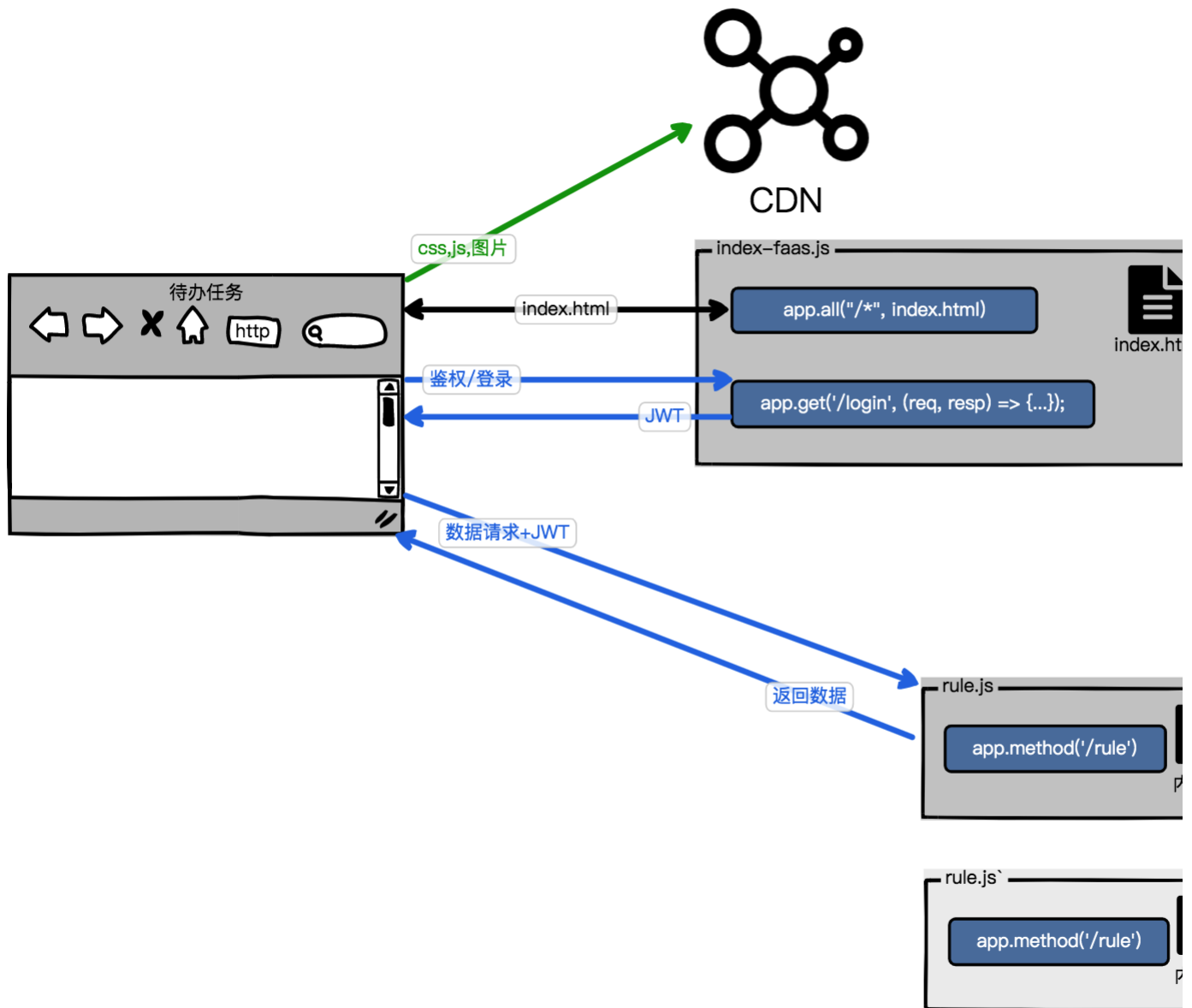
鉴权

其实，FaaS提供的安全防护通常是放在触发器上的。触发器的授权类型或认证方式我们可以设置为：匿名anonymous或函数function。匿名方式就是不需要签名认证，匿名的用户也能访问；而函数方式，则是需要签名认证[4]，这个签名认证的算法，参数需要用到我们账户的访问密钥ak/sk[5]，ak/sk相当于我们云账户的银行卡密码，这么重要的账户信息，我们只能限定在服务端使用，前端代码里绝对不可以出现。

也就是说，我们只能在服务端使用函数安全认证方式。如果是这种方案，我们的“待办任务”架构就演进成下图这样了。



那有没有针对匿名认证方式的安全策略呢？当然有，这里我们同样需要借鉴一下微服务的鉴权设计：JSON Web Token，简称JWT[6]。JWT简单来说，就是将用户身份信息和签名信息，一起传到客户端去了。用户在请求数据时，带上这个JWT，服务端通过校验签名确定用户身份。JWT存在于客户端，JWT验证只需要通过服务端的sk和算法验证签名。同样，我画了张图，以帮助你理解。



要解决后端互调的安全性，我们用VPC或IP白名单，都很容易解决。比较难处理的是前后端的信任问题，JWT正好就提供了一种信任链的解决思路。当然，关于鉴权也有一些云服务商推出了一些更加安全易用的BaaS服务，例如AWS的IAM和Cognito[7]。

安全性是我们考虑架构设计时重要的一环，因为安全架构设计的失败，会直接导致我们资产的损失。鉴权是识别用户身份，防止用户信息泄漏和恶意攻击使用的。但根据我统计的数据，我们在日常99%的问题，都发生在新版本上线的环节。

那我们该怎么稳定持续地快速迭代，发布新版本上线呢？我们可以回想一下[第1课]，小程和小服的例子，小程最后实现NoOps后，小服则只要将代码合并到指定分支就可以发布上线了。那现实中，这点该怎么实现呢？

当我们的项目Serverless化以后，代码的质量变得尤为重要。你可以想想，Serverless化之前，你不小心上线了一个bug，影响的范围最大也就只有一个应用。但是Serverless化之后，如果是核心节点发布了严重的bug上线，那么影响的范围就是所有依赖它的线上应用了。

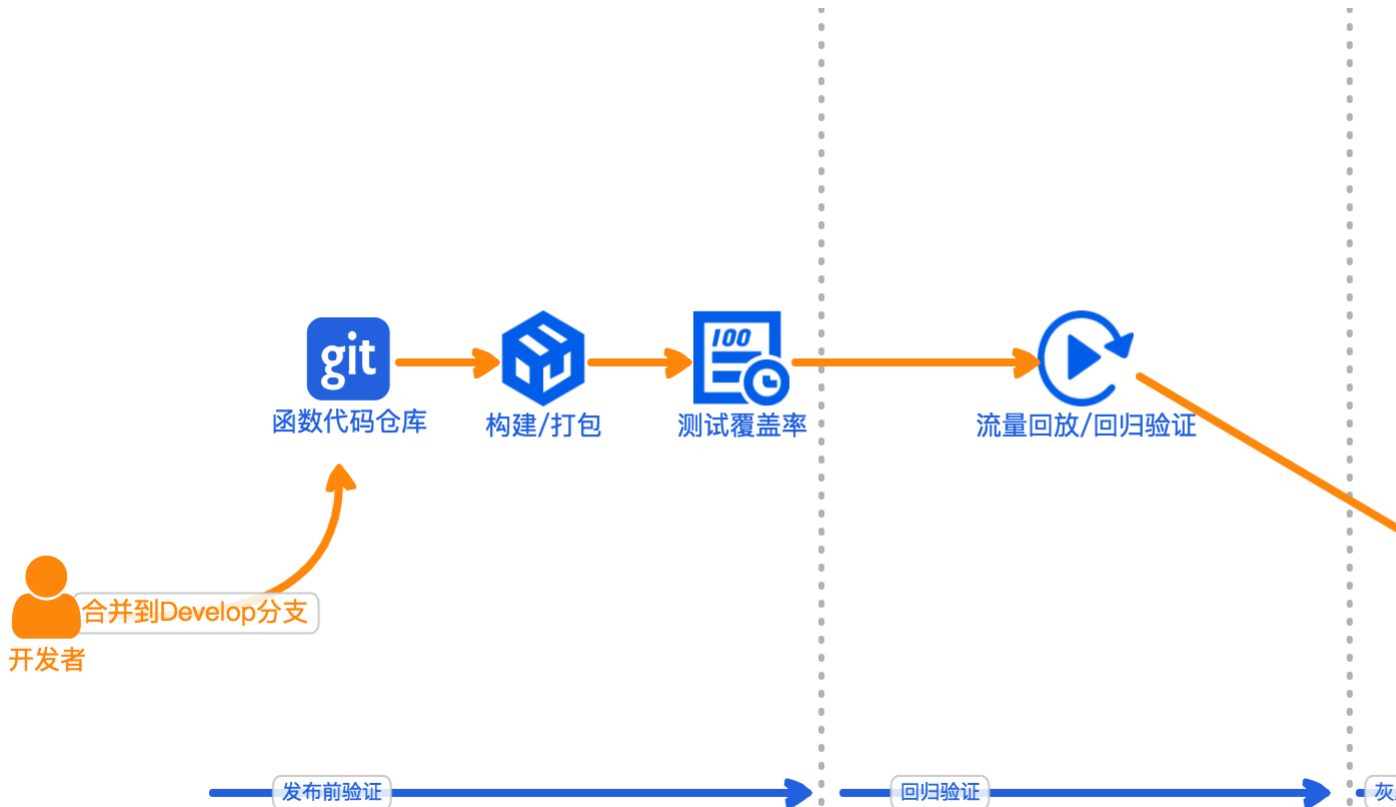
不过，你也不用太担心，微服务和FaaS都具备快速独立迭代的能力。以前我们一个应用的迭代周期通常要一周到两周。但对于Serverless化后的应用来说，每个节点借助独立运维的特性，可以随时随地的发布上线。

综上，我们知道了，微服务和FaaS都是快速迭代的，修复问题很快，但我们也不能每次都等问题出现，再去依赖这个能力呀。有没有什么办法可以提前发现问题，保证我们又快又稳？目前软件工程的最佳做法就是代码流水线的发布管道。

发布管道

发布管道的流水线主要有3个部分：

1. 代码发布前的验证，代码测试覆盖率CI/CD；
2. 模拟流量回归测试通过，发布到灰度环境；
3. 代码正式上线，灰度环境替换正式环境。流水线的每个节点产生的结果，都会作为下一个节点必要的启动参数。



我们先看看上图，我来解释下这个流程。

- 我们的代码合并到指定分支后，通常我会用Develop分支。
- Git的钩子就会触发后续的流水线，开始进入构建打包、测试流程。
- 测试节点做的事情就是跑所有测试Case，并且统计覆盖率。
- 覆盖率验证通过，代码实例用录制流量模拟验证。
- 模拟验证通过，发布代码实例到灰度环境。
- 线上根据灰度策略，将小部分流量导入灰度环境验证灰度版本。
- 在灰度窗口期，比如两个小时，灰度验证没有异常则用灰度版本替换正式版本；反之则立即丢弃这个灰度版本，止损。

这套流程，目前规模大一些的互联网公司发布流程基本都在这样跑，如果你不是很了解，可以自己尝试用我们介绍的Serverless工作流或者云服务商提供的工作流工具[8]动手搭建下。

在这套流程的基础上，很多企业为了追求更高的稳定性，还会设定环境隔离的流水线和安全卡口。比如隔离测试环境和线上环境，测试环境用来复现故障。每次代码进入发布管道，都必须先在测试环境跑通，跑通后安全卡口放行，才能进入线上环境的流水线。

总结

这节课，我们继续讲后端的BaaS化。我们再梳理一下这节课的重要知识点吧。

1. 如何拆解BaaS应用，我们学习了微服务的重要拆解思想DDD：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。另外我也介绍了另一种更适合初创企业的拆分思路：动态网络演进。
2. 拆解完之后，我们就要考虑合并。这里我们介绍了代码编排以外的另一种编排方式：事件流编排，它就是通过一个个事件顺序将我们的微服务或FaaS串联起来。
3. 为了解决拆解后，微服务之间的信任问题。我们先了解了FaaS触发器的安全方案：数字签名。还借鉴了微服务的鉴权做法JWT，将用户鉴权加密信息放在客户端，让鉴权服务变成Stateless。最后，为了让微服务又快又稳地发布版本，我们借鉴了微服务的发布管道：打造自动灰度流水线。

作业

这节课的作业就是我们JWT鉴权的“待办任务”Web应用，你来部署上线。

后端代码GitHub地址：<https://github.com/pusongyang/todolist-backend/tree/lesson06>

前端代码GitHub地址：<https://github.com/pusongyang/todolist-frontend>

演示预览地址：<http://lesson6.jike-serverless.online/list>

期待你的作业，如果今天的内容让你有所收获，也欢迎你把文章分享给身边的朋友，邀请他加入学习。

参考资料

[1] https://en.wikipedia.org/wiki/Domain-driven_design

[2] https://en.wikipedia.org/wiki/Dynamic_network_analysis

[3] <https://www.aliyun.com/product/fnf>

[4] <https://github.com/alibaba/aliyun-ic-nodejs-sdk/blob/master/lib/client.js?spm=a2c4g.11186623.2.15.16e016d7lo8NBQ#L840>

[5] https://help.aliyun.com/document_detail/154851.html?spm=5176.2020520153.0.0.371a415dLXylZ

[6] <https://jwt.io/>

[7] https://docs.aws.amazon.com/zh_cn/cognito/latest/developerguide/what-is-amazon-cognito.html

[8] <https://www.aliyun.com/product/yunxiao/devops?spm=5176.10695662.1173276.1.6c724a38akCjgo>

你好，我是秦粤。上一课中，我们学习了后端BaaS化的重要模块：微服务。现在我们知道微服务的核心理念就是先拆后合，拆解功能是为了提升我们功能的利用率。同步我们也了解了实现微服务的10要素，这10要素要真讲起来够单独开一门课的。如果你不熟悉，我向你推荐杨波老师的《微服务架构核心20讲》课程。

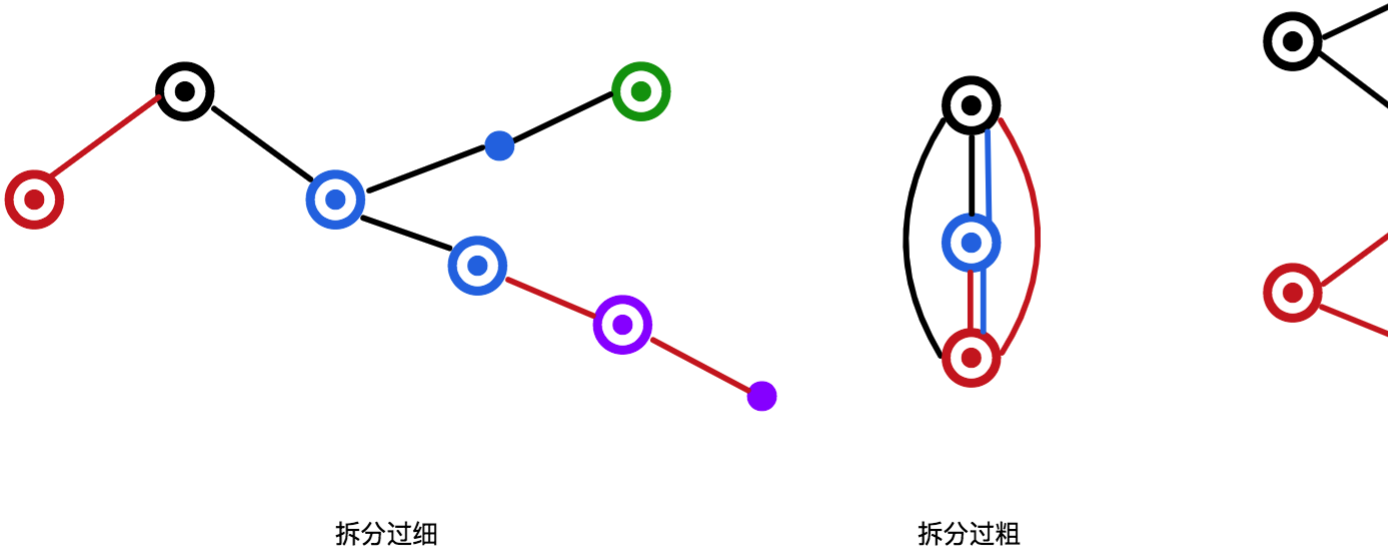
BaaS化的核心其实就是把我们的后端应用封装成RESTful API，然后对外提供服务，而为了后端应用更容易维护，我们需要将后端应用拆解成免运维的微服务。这个逻辑你要理解，这也是为什么我要花这么多篇幅给你谈微服务的关键原因。

上节课我们将“待办任务”Web服务的后端，拆解为用户微服务和待办任务微服务。但为什么要这样拆？是凭感觉，还是有具体的方法论？这里你可以停下来想想。

微服务的拆解和合并，都有一个度需要把握，因为我们在一拆一合之间，都是有成本产生的。如果我们拆解得太细，就必然会导致我们的调用链路增长。调用链路变长，首先影响的就是网络延迟，这个好理解，毕竟你路远了，可能“堵车”的地方也会变多；其次是运维成本的增加，调用链路越长，整个链条就越脆弱，因为其中一环出现问题，都会导致整个调用链条访问失败，而且我们排查问题也变得更加困难。

反过来看，如果我们拆解得太粗，调用链路倒是短了，但是这个微服务的复用性就差了，更别提因为高耦合带来的复杂且冗余的数据库表结构，让我们后续难以维护。我画了个图，你感受下。

带环的是Stateless节点
实心的是Stateful节点



拆之，领域驱动设计

那我们要合理地拆解微服务，应该怎么拆解呢？上节课其实我有提到，目前主流的解决方案就是领域驱动设计，也叫DDD[1]。DDD是Eric Evans在其2004年的同名书中提出来的一个思想，但一直仅仅局限在Java的圈子里，直到2014年，微服务兴起后大家才发现它可以指导微服务的拆分，这才走进了大多数人的视野。用一句话简单总结，DDD就是一套方法论：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。

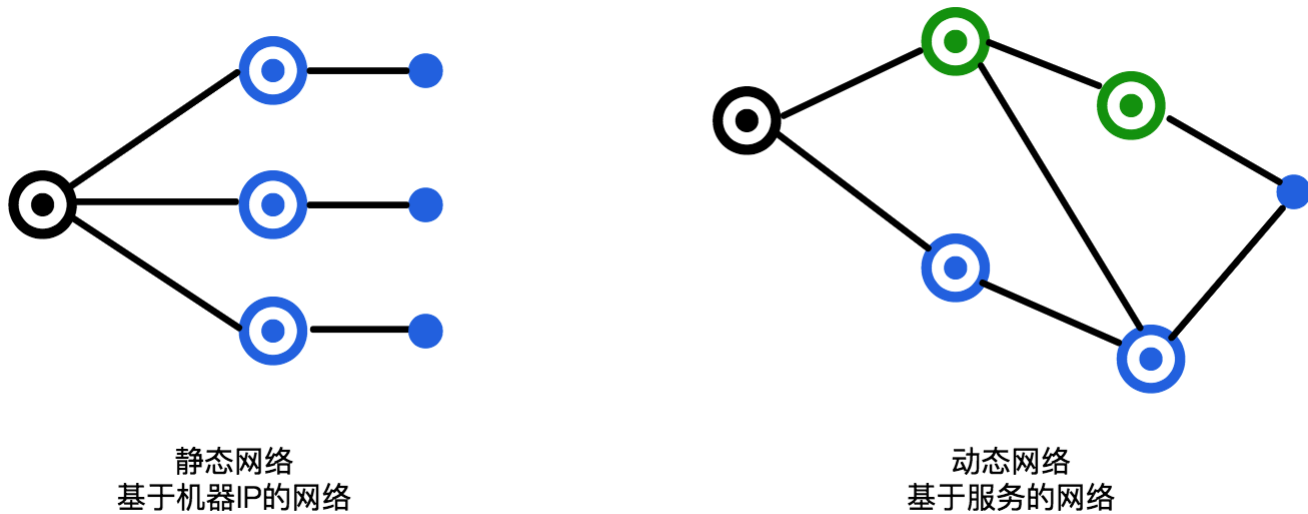
实际我自己在用微服务开发的过程发现，微服务整体应该是一个动态网络结构[2]，随着业务的发展，这个网络结构也会发生变化。DDD能帮助我们前期分析出一个较好的网络结构，但实际上，我们更应该思考的是如何整体优化动态网络：减少核心节点，保护核心节点，降低网络深度等等。

怎么理解动态网络优化呢？我们可以做个思维实验：假设我们将所有的功能都拆解成微服务，任意的微服务节点之间都可以相互调用，调用越频繁它们之间的距离就越近。那么我们考虑一下，当我们网站的访问请求流量稳定后，我们整个微服务节点组成的网络状态是怎么样的？

首先网络节点的相互制约总会让那些相互之间强依赖的、高耦合的节点，越走越近，最后聚集成一团节点。其次那些跟业务逻辑无关的节点，逐渐被边缘化，甚至消失。我们看这些聚集成团的节点，如果团里的点聚合太近了，其实是不适合拆分的，它们整体应该作成一个微服务。等这些节点太近的团合并成一个微服务节点后，我们再看那些聚集在一起、又不太近的节点就是一个微服务了。

所以，我们在启动项目时，不用太过纠结应该如何去拆解微服务。而应该持续关注，并思考每个微服务节点的合理性。就像看待动态网络一样，持续地调整优化，去除核心节点。最终它会伴随你业务的发展阶段，达到各个阶段的稳定动态网络结构。

就像我们上节课“待办任务”Web服务一样，我们可以先简单地将我们的项目后端分为：用户微服务和待办任务微服务。当然这里我们目前的业务太简单了，用DDD去分析，也是大材小用。随着我这个项目的业务发展，我们添加的功能会越来越多。让微服务根据业务一起成长演变就可以了。这并不是说我们就放任微服务不管了，而是从整体网络的角度思考，去看我们的微服务如何演进。



合之，Streaming

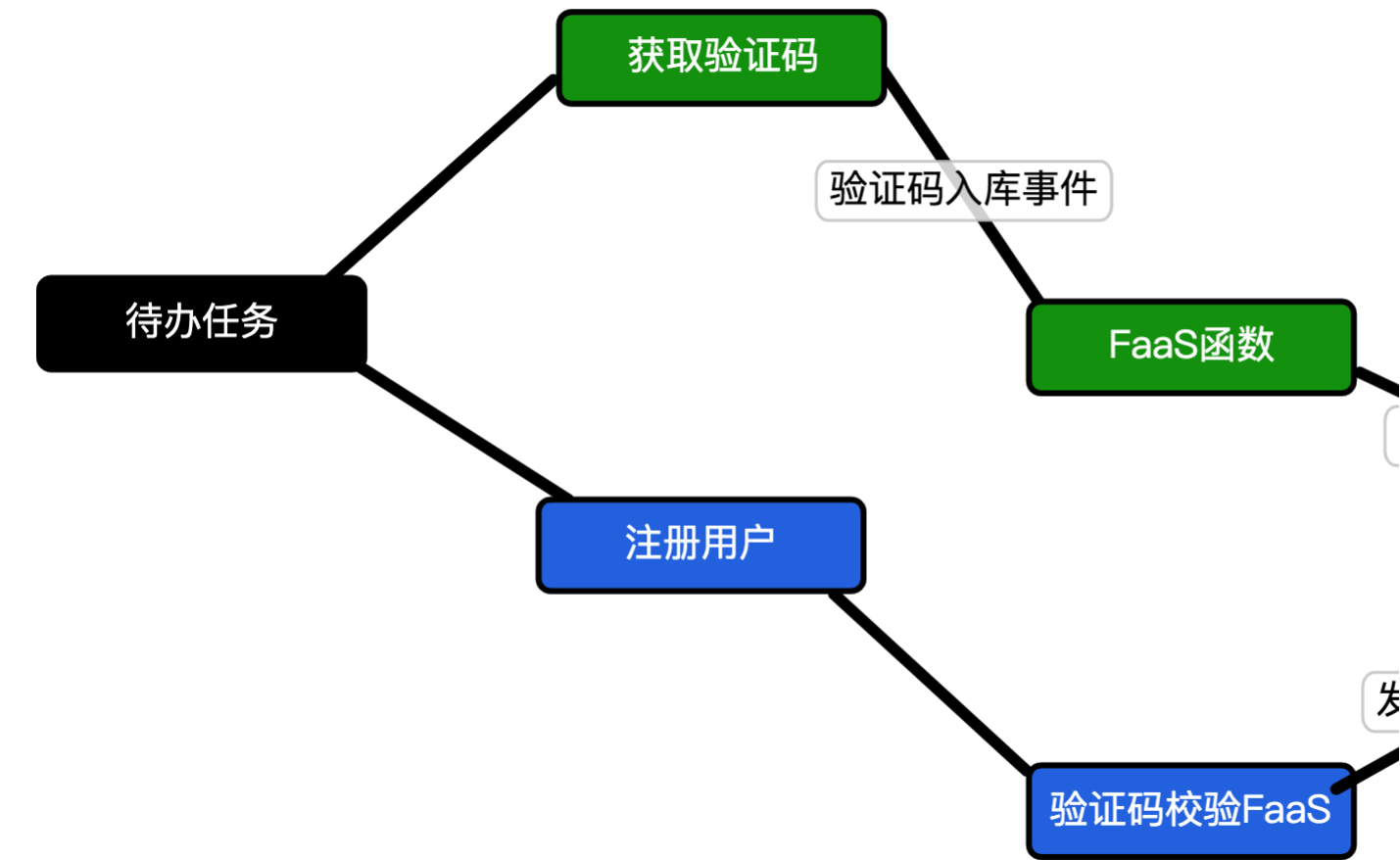
看完拆解，我们再看合并。合并呢，换个高大上的词其实就是前面课程中提到的编排。目前为止，我们整个“待办任务”Web应用架构的设计基本完成了，而且所有节点都是Stateless的了。变成Stateless节点后，其实对于前端的同学来说，一点都不陌生，比如React的单向数据流中的State也要求我们Immutable，Immutable其实就是Stateless。

我们上面已经看到了，拆解后的架构是个动态网络，那我们应该怎么合并或者编排呢？当然你像SFF那样通过传统的函数，将每个HTTP数据的请求结果通过数组或对象加工处理，再将这些结果返回也是可以的。但我在这里想向你介绍另外一种编排思路， workflow。



我们可以将用户的请求想象成我们的呼吸系统，我们的肺就是SFF，而微服务和FaaS节点就是需要氧气的各个器官。我们吸一口气，氧气进入肺部，血液循环将氧气按顺序流经我们每个器官，这就是请求链路。每个器官一接收到新鲜血液，就会吸取氧气返回二氧化碳，最终血液循环将二氧化碳带到肺部呼出，这个就是数据返回链路。我们的各个器官，就被请求链路通过新鲜血液到来的这个事件串联起来了，这个就是事件流，也就是用一个事件去串联FaaS或微服务。

现在我们用[第3课]讲的，PHP发邮件改造一下，举个例子。当用户注册时，我们完全可以将用户的信息和注册验证码存入数据库；PHP发邮件的FaaS触发器改为数据库插入新记录触发事件；用户从邮箱验证获取验证码，把验证码写到输入框后，点击验证，则是另一个HTTP触发器，触发FaaS函数校验验证码通过，修改数据库注册成功，并且返回302跳转到登录成功页面。具体流程可参考下图：



当然现在这个解决方案也有成熟对应的云BaaS服务：Serverless工作流[3]。

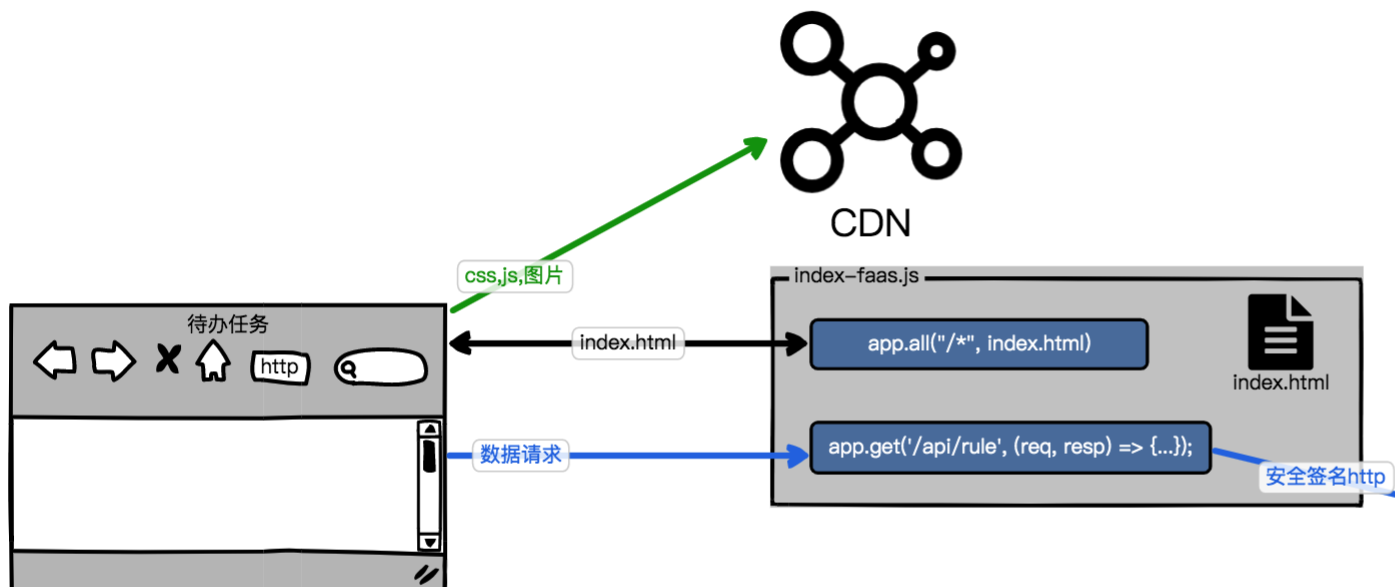
安全门神

理解了拆合的思想，我们就可以将目前“待办项目”的架构再演进一下：静态文件我们用CDN托管，前端项目只负责域名支撑和index.html，剩下的请求直接访问FaaS微服务。这时候，我估计你会问，咱们数据的安全性如何保障呢？是的，到目前为止，我们的FaaS都一直在用匿名模式访问，完全没有任何安全防护可言，也就是说目前我们FaaS服务的接口一直都在互联网上“裸奔”。

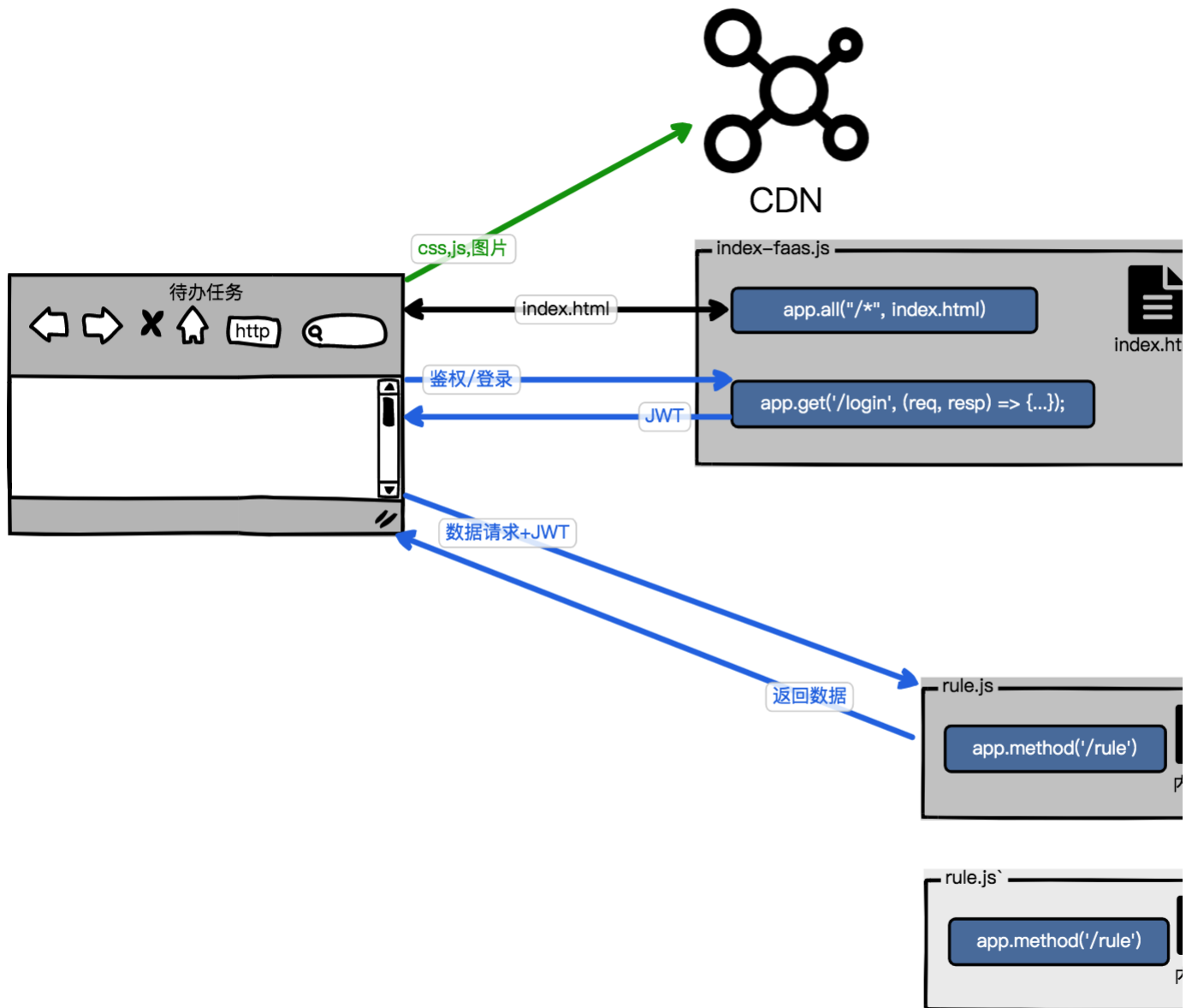
鉴权

其实，FaaS提供的安全防护通常是放在触发器上的。触发器的授权类型或认证方式我们可以设置为：匿名anonymous或函数function。匿名方式就是不需要签名认证，匿名的用户也能访问；而函数方式，则是需要签名认证[4]，这个签名认证的算法，参数需要用到我们账户的访问密钥ak/sk[5]，ak/sk相当于我们云账户的银行卡密码，这么重要的账户信息，我们只能限定在服务端使用，前端代码里绝对不可以出现。

也就是说，我们只能在服务端使用函数安全认证方式。如果是这种方案，我们的“待办任务”架构就演进成下图这样了。



那有没有针对匿名认证方式的安全策略呢？当然有，这里我们同样需要借鉴一下微服务的鉴权设计：JSON Web Token，简称JWT[6]。JWT简单来说，就是将用户身份信息和签名信息，一起传到客户端去了。用户在请求数据时，带上这个JWT，服务端通过校验签名确定用户身份。JWT存在于客户端，JWT验证只需要通过服务端的sk和算法验证签名。同样，我画了张图，以帮助你理解。



要解决后端互调的安全性，我们用VPC或IP白名单，都很容易解决。比较难处理的是前后端的信任问题，JWT正好就提供了一种信任链的解决思路。当然，关于鉴权也有一些云服务商推出了一些更加安全易用的BaaS服务，例如AWS的IAM和Cognito[7]。

安全性是我们考虑架构设计时重要的一环，因为安全架构设计的失败，会直接导致我们资产的损失。鉴权是识别用户身份，防止用户信息泄漏和恶意攻击使用的。但根据我统计的数据，我们在日常99%的问题，都发生在新版本上线的环节。

那我们该怎么稳定持续地快速迭代，发布新版本上线呢？我们可以回想一下[第1课]，小程和小服的例子，小程最后实现NoOps后，小服则只要将代码合并到指定分支就可以发布上线了。那现实中，这点该怎么实现呢？

当我们的项目Serverless化以后，代码的质量变得尤为重要。你可以想想，Serverless化之前，你不小心上线了一个bug，影响的范围最大也就只有一个应用。但是Serverless化之后，如果是核心节点发布了严重的bug上线，那么影响的范围就是所有依赖它的线上应用了。

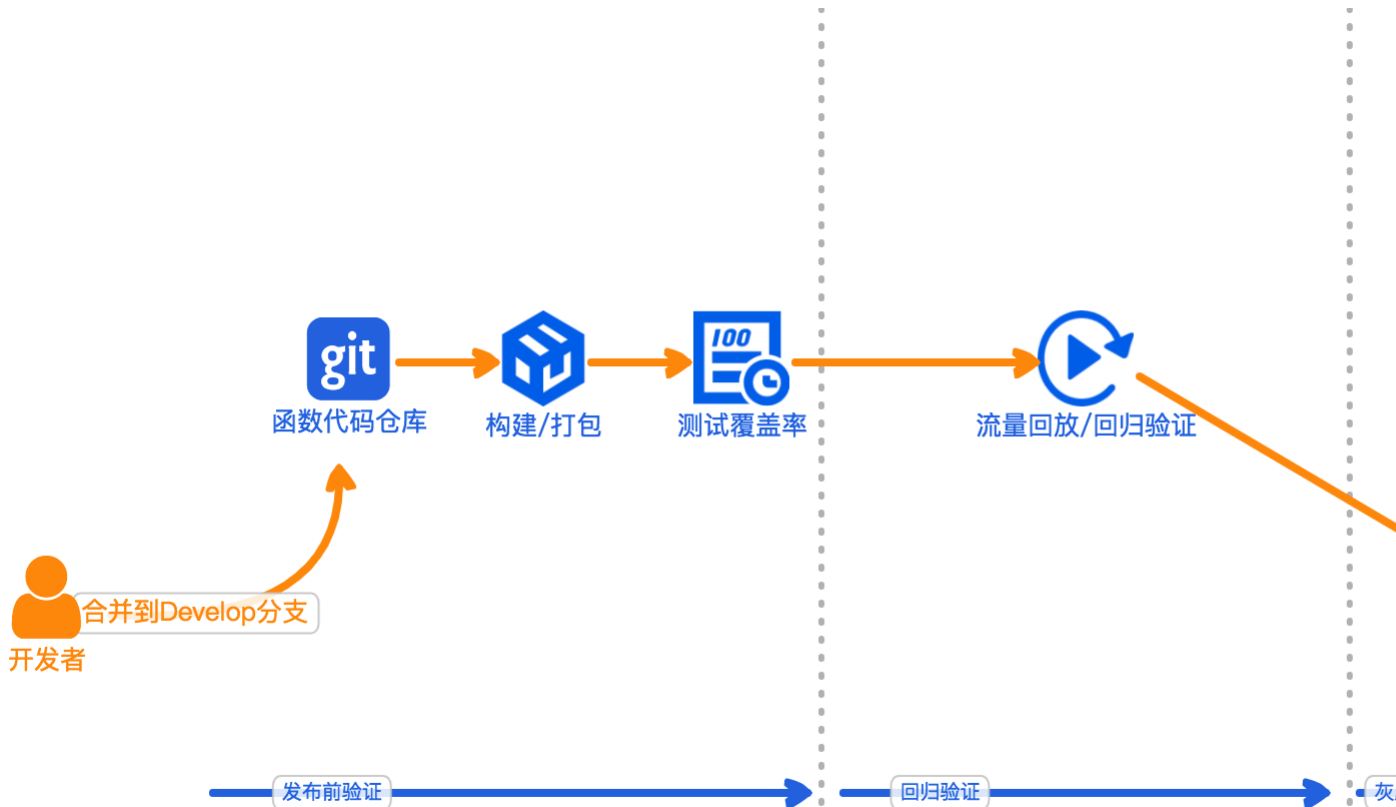
不过，你也不用太担心，微服务和FaaS都具备快速独立迭代的能力。以前我们一个应用的迭代周期通常要一周到两周。但对于Serverless化后的应用来说，每个节点借助独立运维的特性，可以随时随地的发布上线。

综上，我们知道了，微服务和FaaS都是快速迭代的，修复问题很快，但我们也不能每次都等问题出现，再去依赖这个能力呀。有没有什么办法可以提前发现问题，保证我们又快又稳？目前软件工程的最佳做法就是代码流水线的发布管道。

发布管道

发布管道的流水线主要有3个部分：

1. 代码发布前的验证，代码测试覆盖率CI/CD；
2. 模拟流量回归测试通过，发布到灰度环境；
3. 代码正式上线，灰度环境替换正式环境。流水线的每个节点产生的结果，都会作为下一个节点必要的启动参数。



我们先看看上图，我来解释下这个流程。

- 我们的代码合并到指定分支后，通常我会用Develop分支。
- Git的钩子就会触发后续的流水线，开始进入构建打包、测试流程。
- 测试节点做的事情就是跑所有测试Case，并且统计覆盖率。
- 覆盖率验证通过，代码实例用录制流量模拟验证。
- 模拟验证通过，发布代码实例到灰度环境。
- 线上根据灰度策略，将小部分流量导入灰度环境验证灰度版本。
- 在灰度窗口期，比如两个小时，灰度验证没有异常则用灰度版本替换正式版本；反之则立即丢弃这个灰度版本，止损。

这套流程，目前规模大一些的互联网公司发布流程基本都在这样跑，如果你不是很了解，可以自己尝试用我们介绍的Serverless工作流或者云服务商提供的工作流工具[8]动手搭建下。

在这套流程的基础上，很多企业为了追求更高的稳定性，还会设定环境隔离的流水线和安全卡口。比如隔离测试环境和线上环境，测试环境用来复现故障。每次代码进入发布管道，都必须先在测试环境跑通，跑通后安全卡口放行，才能进入线上环境的流水线。

总结

这节课，我们继续讲后端的BaaS化。我们再梳理一下这节课的重要知识点吧。

1. 如何拆解BaaS应用，我们学习了微服务的重要拆解思想DDD：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。另外我也介绍了另一种更适合初创企业的拆分思路：动态网络演进。
2. 拆解完之后，我们就要考虑合并。这里我们介绍了代码编排以外的另一种编排方式：事件流编排，它就是通过一个个事件顺序将我们的微服务或FaaS串联起来。
3. 为了解决拆解后，微服务之间的信任问题。我们先了解了FaaS触发器的安全方案：数字签名。还借鉴了微服务的鉴权做法JWT，将用户鉴权加密信息放在客户端，让鉴权服务变成Stateless。最后，为了让微服务又快又稳地发布版本，我们借鉴了微服务的发布管道：打造自动灰度流水线。

作业

这节课的作业就是我们JWT鉴权的“待办任务”Web应用，你来部署上线。

后端代码GitHub地址：<https://github.com/pusongyang/todolist-backend/tree/lesson06>

前端代码GitHub地址：<https://github.com/pusongyang/todolist-frontend>

演示预览地址：<http://lesson6.jike-serverless.online/list>

期待你的作业，如果今天的内容让你有所收获，也欢迎你把文章分享给身边的朋友，邀请他加入学习。

参考资料

[1] https://en.wikipedia.org/wiki/Domain-driven_design

[2] https://en.wikipedia.org/wiki/Dynamic_network_analysis

[3] <https://www.aliyun.com/product/fnf>

[4] <https://github.com/alibaba/aliyun-fc-nodejs-sdk/blob/master/lib/client.js?spm=a2c4g.11186623.2.15.16e016d7lo8NBQ#L840>

[5] https://help.aliyun.com/document_detail/154851.html?spm=5176.2020520153.0.0.371a415dLXylZ

[6] <https://jwt.io/>

[7] https://docs.aws.amazon.com/zh_cn/cognito/latest/developerguide/what-is-amazon-cognito.html

[8] <https://www.aliyun.com/product/yunxiao/devops?spm=5176.10695662.1173276.1.6c724a38akCjgo>

你好，我是秦粤。上一课中，我们学习了后端BaaS化的重要模块：微服务。现在我们知道微服务的核心理念就是先拆后合，拆解功能是为了提升我们功能的利用率。同步我们也了解了实现微服务的10要素，这10要素要真讲起来够单独开一门课的。如果你不熟悉，我向你推荐杨波老师的《微服务架构核心20讲》课程。

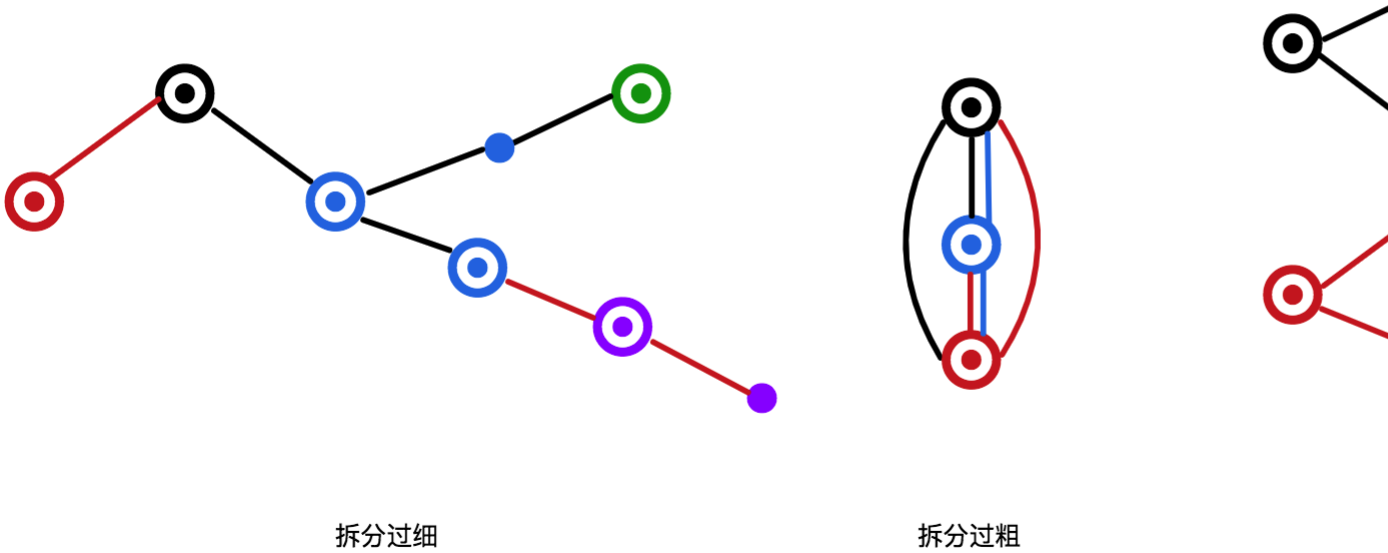
BaaS化的核心其实就是把我们的后端应用封装成RESTful API，然后对外提供服务，而为了后端应用更容易维护，我们需要将后端应用拆解成免运维的微服务。这个逻辑你要理解，这也是为什么我要花这么多篇幅给你谈微服务的关键原因。

上节课我们将“待办任务”Web服务的后端，拆解为用户微服务和待办任务微服务。但为什么要这样拆？是凭感觉，还是有具体的方法论？这里你可以停下来想想。

微服务的拆解和合并，都有一个度需要把握，因为我们在一拆一合之间，都是有成本产生的。如果我们拆解得太细，就必然会导致我们的调用链路增长。调用链路变长，首先影响的就是网络延迟，这个好理解，毕竟你路远了，可能“堵车”的地方也会变多；其次是运维成本的增加，调用链路越长，整个链条就越脆弱，因为其中一环出现问题，都会导致整个调用链条访问失败，而且我们排查问题也变得更加困难。

反过来看，如果我们拆解得太粗，调用链路倒是短了，但是这个微服务的复用性就差了，更别提因为高耦合带来的复杂且冗余的数据库表结构，让我们后续难以维护。我画了个图，你感受下。

带环的是Stateless节点
实心的是Stateful节点



拆之，领域驱动设计

那我们要合理地拆解微服务，应该怎么拆解呢？上节课其实我有提到，目前主流的解决方案就是领域驱动设计，也叫DDD[1]。DDD是Eric Evans在其2004年的同名书中提出来的一个思想，但一直仅仅局限在Java的圈子里，直到2014年，微服务兴起后大家才发现它可以指导微服务的拆分，这才走进了大多数人的视野。用一句话简单总结，DDD就是一套方法论：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。

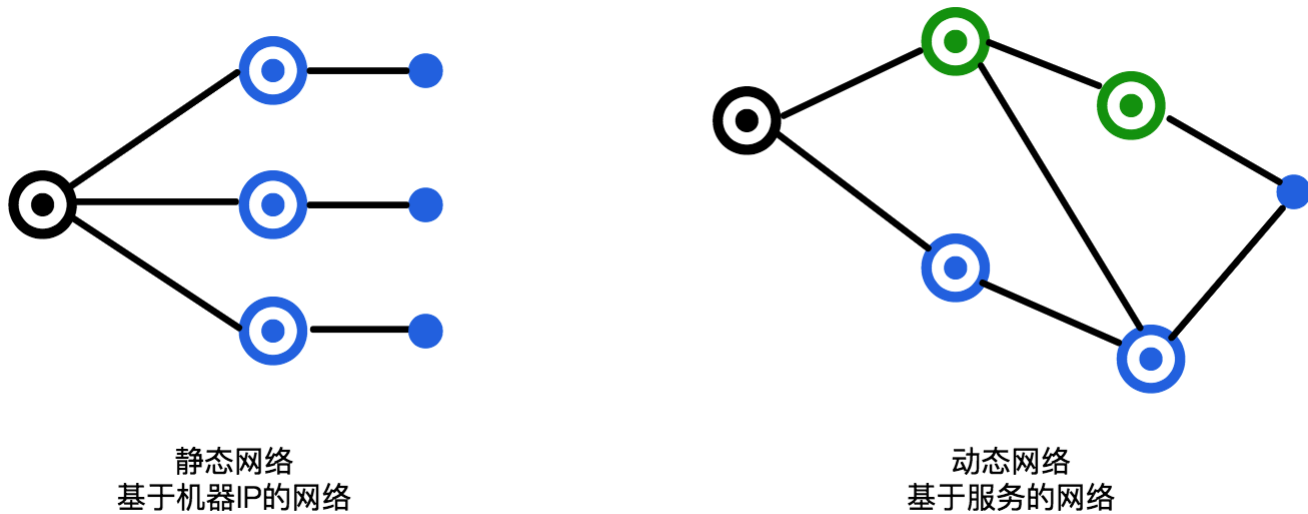
实际我自己在用微服务开发的过程发现，微服务整体应该是一个动态网络结构[2]，随着业务的发展，这个网络结构也会发生变化。DDD能帮助我们前期分析出一个较好的网络结构，但实际上，我们更应该思考的是如何整体优化动态网络：减少核心节点，保护核心节点，降低网络深度等等。

怎么理解动态网络优化呢？我们可以做个思维实验：假设我们将所有的功能都拆解成微服务，任意的微服务节点之间都可以相互调用，调用越频繁它们之间的距离就越近。那么我们考虑一下，当我们网站的访问请求流量稳定后，我们整个微服务节点组成的网络状态是怎么样的？

首先网络节点的相互制约总会让那些相互之间强依赖的、高耦合的节点，越走越近，最后聚集成一团节点。其次那些跟业务逻辑无关的节点，逐渐被边缘化，甚至消失。我们看这些聚集成团的节点，如果团里的点聚合太近了，其实是不适合拆分的，它们整体应该作成一个微服务。等这些节点太近的团合并成一个微服务节点后，我们再看那些聚集在一起、又不太近的节点就是一个微服务了。

所以，我们在启动项目时，不用太过纠结应该如何去拆解微服务。而应该持续关注，并思考每个微服务节点的合理性。就像看待动态网络一样，持续地调整优化，去除核心节点。最终它会伴随你业务的发展阶段，达到各个阶段的稳定动态网络结构。

就像我们上节课“待办任务”Web服务一样，我们可以先简单地将我们的项目后端分为：用户微服务和待办任务微服务。当然这里我们目前的业务太简单了，用DDD去分析，也是大材小用。随着我这个项目的业务发展，我们添加的功能会越来越多。让微服务根据业务一起成长演变就可以了。这并不是说我们就放任微服务不管了，而是从整体网络的角度思考，去看我们的微服务如何演进。



合之，Streaming

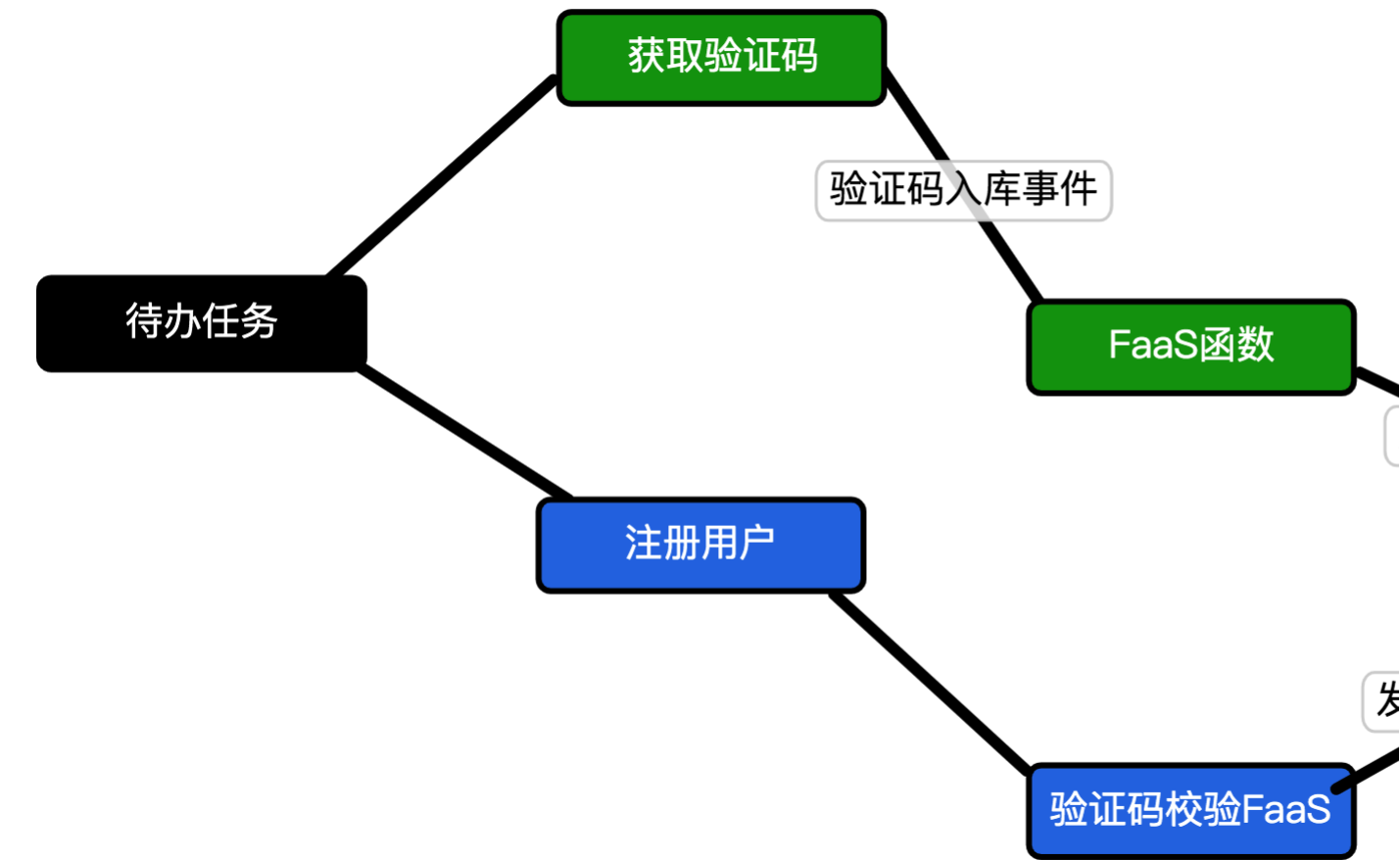
看完拆解，我们再看合并。合并呢，换个高大上的词其实就是前面课程中提到的编排。目前为止，我们整个“待办任务”Web应用架构的设计基本完成了，而且所有节点都是Stateless的了。变成Stateless节点后，其实对于前端的同学来说，一点都不陌生，比如React的单向数据流中的State也要求我们Immutable，Immutable其实就是Stateless。

我们上面已经看到了，拆解后的架构是个动态网络，那我们应该怎么合并或者编排呢？当然你像SFF那样通过传统的函数，将每个HTTP数据的请求结果通过数组或对象加工处理，再将这些结果返回也是可以的。但我在这里想向你介绍另外一种编排思路， workflow。



我们可以将用户的请求想象成我们的呼吸系统，我们的肺就是SFF，而微服务和FaaS节点就是需要氧气的各个器官。我们吸一口气，氧气进入肺部，血液循环将氧气按顺序流经我们每个器官，这就是请求链路。每个器官一接收到新鲜血液，就会吸取氧气返回二氧化碳，最终血液循环将二氧化碳带到肺部呼出，这个就是数据返回链路。我们的各个器官，就被请求链路通过新鲜血液到来的这个事件串联起来了，这个就是事件流，也就是用一个事件去串联FaaS或微服务。

现在我们用[第3课]讲的，PHP发邮件改造一下，举个例子。当用户注册时，我们完全可以将用户的信息和注册验证码存入数据库；PHP发邮件的FaaS触发器改为数据库插入新记录触发事件；用户从邮箱验证获取验证码，把验证码写到输入框后，点击验证，则是另一个HTTP触发器，触发FaaS函数校验验证码通过，修改数据库注册成功，并且返回302跳转到登录成功页面。具体流程可参考下图：



当然现在这个解决方案也有成熟对应的云BaaS服务：Serverless工作流[3]。

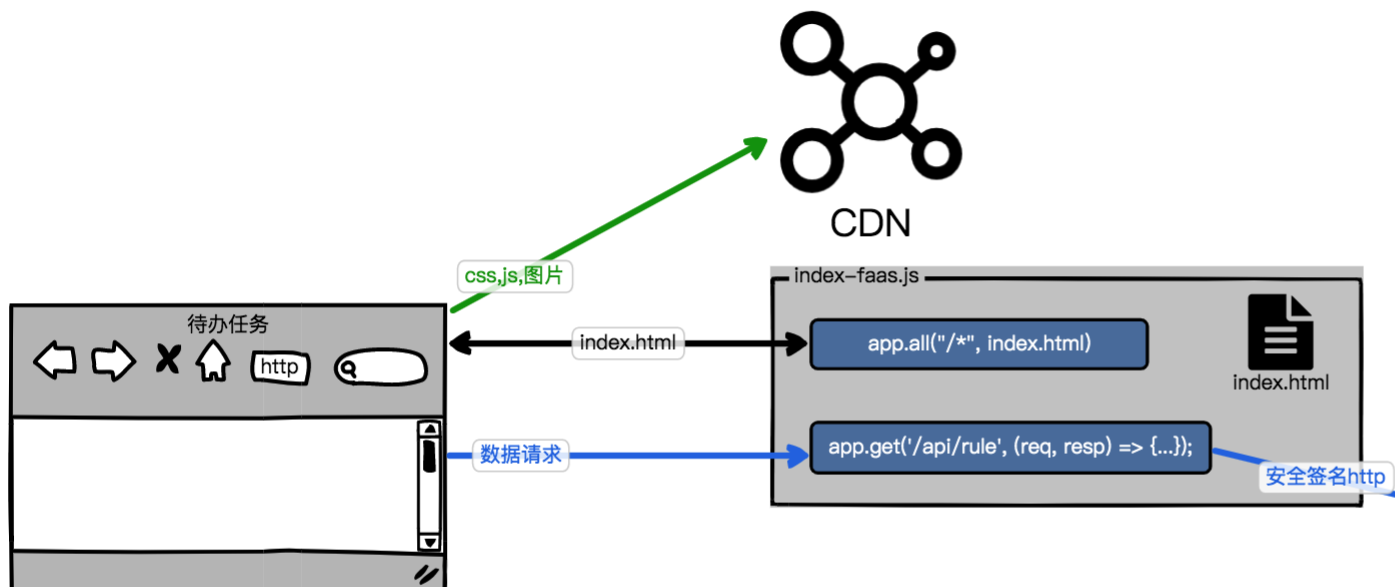
安全门神

理解了拆合的思想，我们就可以将目前“待办项目”的架构再演进一下：静态文件我们用CDN托管，前端项目只负责域名支撑和index.html，剩下的请求直接访问FaaS微服务。这时候，我估计你会问，咱们数据的安全性如何保障呢？是的，到目前为止，我们的FaaS都一直在用匿名模式访问，完全没有任何安全防护可言，也就是说目前我们FaaS服务的接口一直都在互联网上“裸奔”。

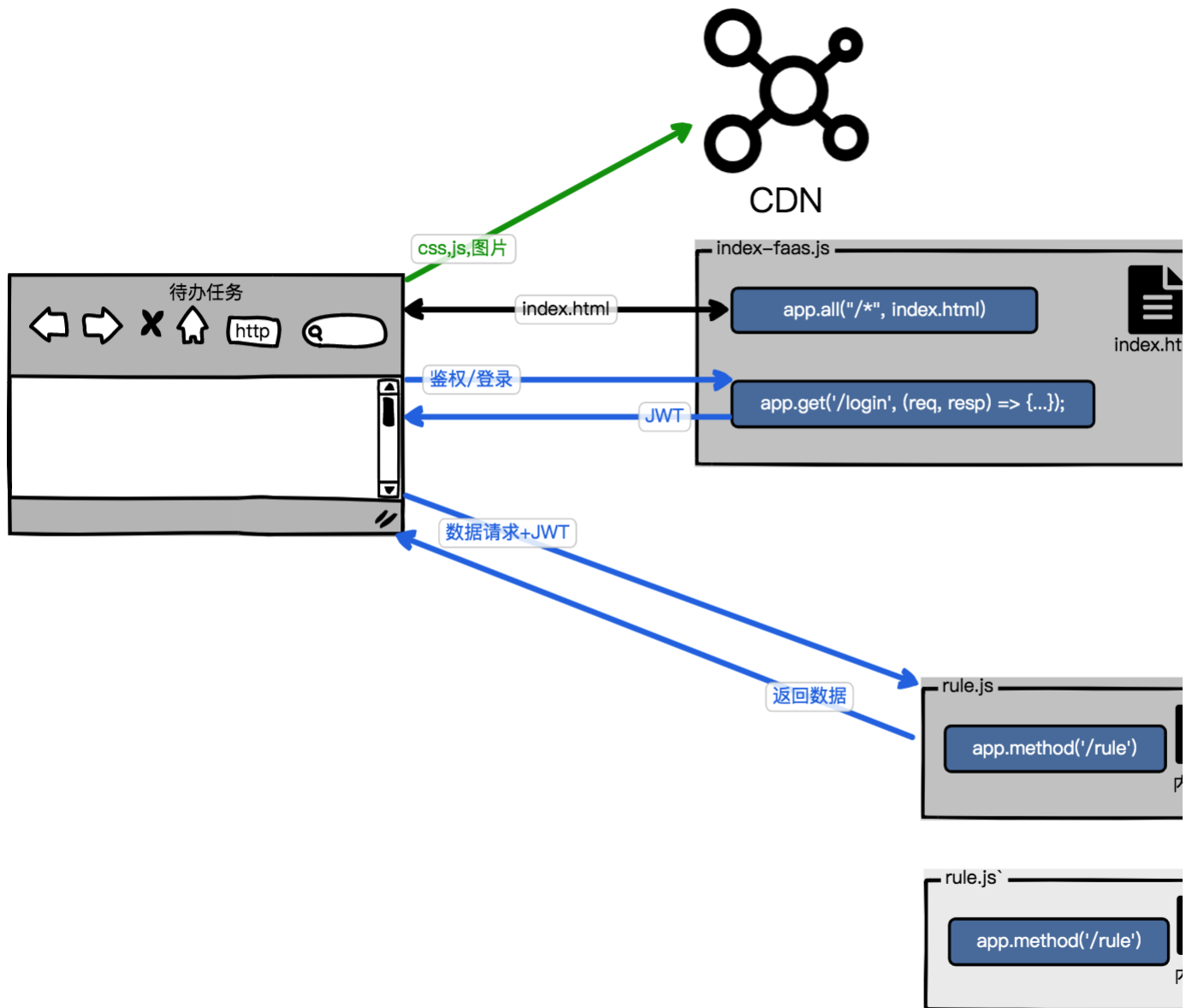
鉴权

其实，FaaS提供的安全防护通常是放在触发器上的。触发器的授权类型或认证方式我们可以设置为：匿名anonymous或函数function。匿名方式就是不需要签名认证，匿名的用户也能访问；而函数方式，则是需要签名认证[4]，这个签名认证的算法，参数需要用到我们账户的访问密钥ak/sk[5]，ak/sk相当于我们云账户的银行卡密码，这么重要的账户信息，我们只能限定在服务端使用，前端代码里绝对不可以出现。

也就是说，我们只能在服务端使用函数安全认证方式。如果是这种方案，我们的“待办任务”架构就演进成下图这样了。



那有没有针对匿名认证方式的安全策略呢？当然有，这里我们同样需要借鉴一下微服务的鉴权设计：JSON Web Token，简称JWT[6]。JWT简单来说，就是将用户身份信息和签名信息，一起传到客户端去了。用户在请求数据时，带上这个JWT，服务端通过校验签名确定用户身份。JWT存在于客户端，JWT验证只需要通过服务端的sk和算法验证签名。同样，我画了张图，以帮助你理解。



要解决后端互调的安全性，我们用VPC或IP白名单，都很容易解决。比较难处理的是前后端的信任问题，JWT正好就提供了一种信任链的解决思路。当然，关于鉴权也有一些云服务商推出了一些更加安全易用的BaaS服务，例如AWS的IAM和Cognito[7]。

安全性是我们考虑架构设计时重要的一环，因为安全架构设计的失败，会直接导致我们资产的损失。鉴权是识别用户身份，防止用户信息泄漏和恶意攻击使用的。但根据我统计的数据，我们在日常99%的问题，都发生在新版本上线的环节。

那我们该怎么稳定持续地快速迭代，发布新版本上线呢？我们可以回想一下[第1课]，小程和小服的例子，小程最后实现NoOps后，小服则只要将代码合并到指定分支就可以发布上线了。那现实中，这点该怎么实现呢？

当我们的项目Serverless化以后，代码的质量变得尤为重要。你可以想想，Serverless化之前，你不小心上线了一个bug，影响的范围最大也就只有一个应用。但是Serverless化之后，如果是核心节点发布了严重的bug上线，那么影响的范围就是所有依赖它的线上应用了。

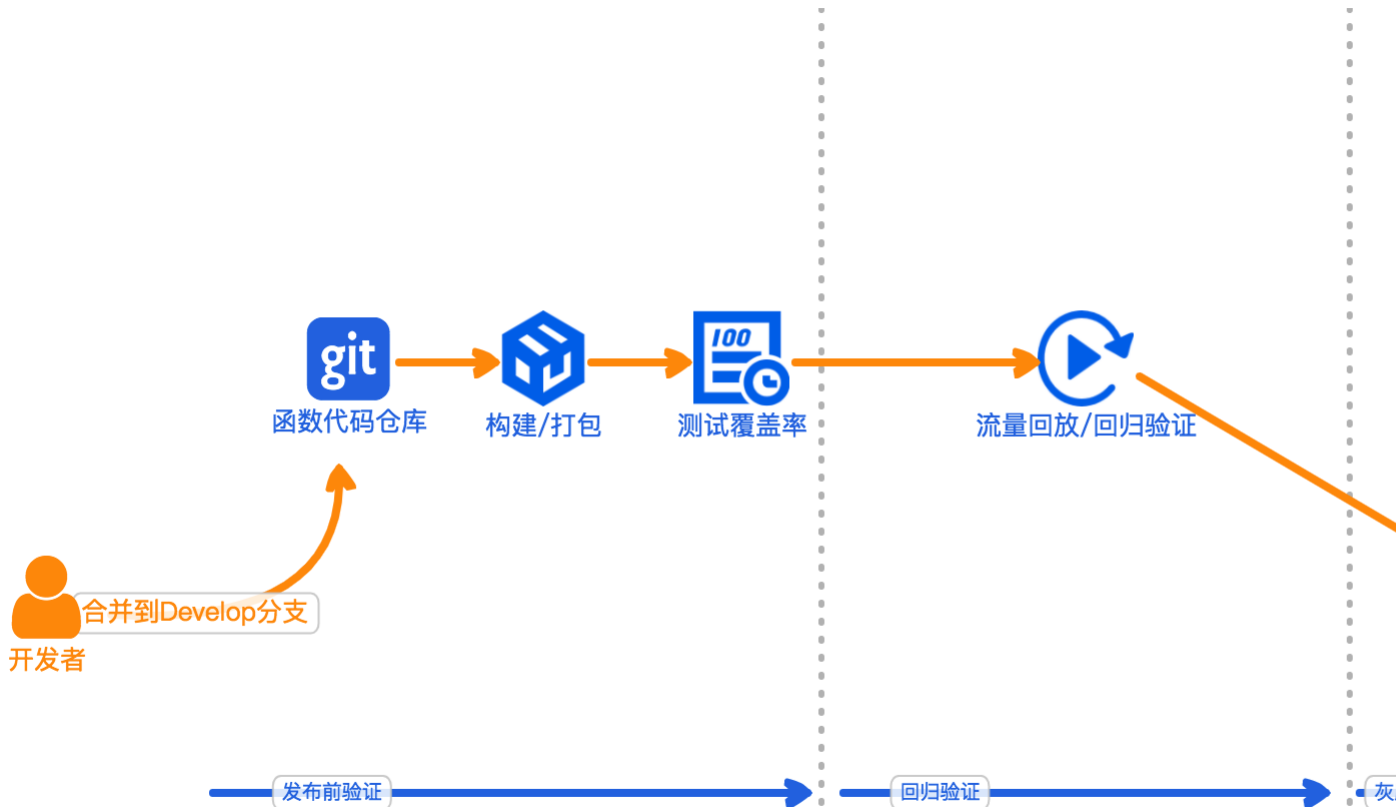
不过，你也不用太担心，微服务和FaaS都具备快速独立迭代的能力。以前我们一个应用的迭代周期通常要一周到两周。但对于Serverless化后的应用来说，每个节点借助独立运维的特性，可以随时随地地发布上线。

综上，我们知道了，微服务和FaaS都是快速迭代的，修复问题很快，但我们也不能每次都等问题出现，再去依赖这个能力呀。有没有什么办法可以提前发现问题，保证我们又快又稳？目前软件工程的最佳做法就是代码流水线的发布管道。

发布管道

发布管道的流水线主要有3个部分：

1. 代码发布前的验证，代码测试覆盖率CI/CD；
2. 模拟流量回归测试通过，发布到灰度环境；
3. 代码正式上线，灰度环境替换正式环境。流水线的每个节点产生的结果，都会作为下一个节点必要的启动参数。



我们先看看上图，我来解释下这个流程。

- 我们的代码合并到指定分支后，通常我会用Develop分支。
- Git的钩子就会触发后续的流水线，开始进入构建打包、测试流程。
- 测试节点做的事情就是跑所有测试Case，并且统计覆盖率。
- 覆盖率验证通过，代码实例用录制流量模拟验证。
- 模拟验证通过，发布代码实例到灰度环境。
- 线上根据灰度策略，将小部分流量导入灰度环境验证灰度版本。
- 在灰度窗口期，比如两个小时，灰度验证没有异常则用灰度版本替换正式版本；反之则立即丢弃这个灰度版本，止损。

这套流程，目前规模大一些的互联网公司发布流程基本都在这样跑，如果你不是很了解，可以自己尝试用我们介绍的Serverless工作流或者云服务商提供的工作流工具[8]动手搭建下。

在这套流程的基础上，很多企业为了追求更高的稳定性，还会设定环境隔离的流水线和安全卡口。比如隔离测试环境和线上环境，测试环境用来复现故障。每次代码进入发布管道，都必须先在测试环境跑通，跑通后安全卡口放行，才能进入线上环境的流水线。

总结

这节课，我们继续讲后端的BaaS化。我们再梳理一下这节课的重要知识点吧。

1. 如何拆解BaaS应用，我们学习了微服务的重要拆解思想DDD：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。另外我也介绍了另一种更适合初创企业的拆分思路：动态网络演进。
2. 拆解完之后，我们就要考虑合并。这里我们介绍了代码编排以外的另一种编排方式：事件流编排，它就是通过一个个事件顺序将我们的微服务或FaaS串联起来。
3. 为了解决拆解后，微服务之间的信任问题。我们先了解了FaaS触发器的安全方案：数字签名。还借鉴了微服务的鉴权做法JWT，将用户鉴权加密信息放在客户端，让鉴权服务变成Stateless。最后，为了让微服务又快又稳地发布版本，我们借鉴了微服务的发布管道：打造自动灰度流水线。

作业

这节课的作业就是我们JWT鉴权的“待办任务”Web应用，你来部署上线。

后端代码GitHub地址：<https://github.com/pusongyang/todolist-backend/tree/lesson06>

前端代码GitHub地址：<https://github.com/pusongyang/todolist-frontend>

演示预览地址：<http://lesson6.jike-serverless.online/list>

期待你的作业，如果今天的内容让你有所收获，也欢迎你把文章分享给身边的朋友，邀请他加入学习。

参考资料

- [1] https://en.wikipedia.org/wiki/Domain-driven_design
- [2] https://en.wikipedia.org/wiki/Dynamic_network_analysis
- [3] <https://www.aliyun.com/product/fnf>
- [4] <https://github.com/alibaba/aliyun-ic-nodejs-sdk/blob/master/lib/client.js?spm=a2c4g.11186623.2.15.16e016d7lo8NBQ#L840>
- [5] https://help.aliyun.com/document_detail/154851.html?spm=5176.2020520153.0.0.371a415dLXylZ
- [6] <https://jwt.io/>
- [7] https://docs.aws.amazon.com/zh_cn/cognito/latest/developerguide/what-is-amazon-cognito.html
- [8] <https://www.aliyun.com/product/yunxiao/devops?spm=5176.10695662.1173276.1.6c724a38akCjgo>

你好，我是秦粤。上一课中，我们学习了后端BaaS化的重要模块：微服务。现在我们知道微服务的核心理念就是先拆后合，拆解功能是为了提升我们功能的利用率。同步我们也了解了实现微服务的10要素，这10要素要真讲起来够单独开一门课的。如果你不熟悉，我向你推荐杨波老师的《微服务架构核心20讲》课程。

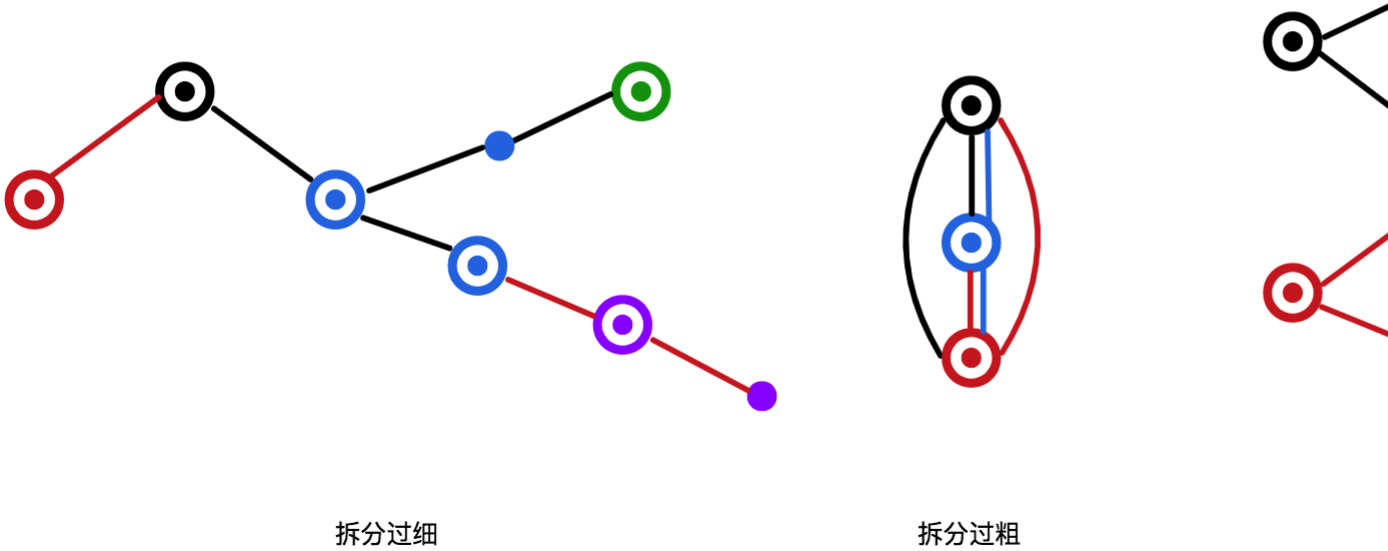
BaaS化的核心其实就是把我们的后端应用封装成RESTful API，然后对外提供服务，而为了后端应用更容易维护，我们需要将后端应用拆解成免运维的微服务。这个逻辑你要理解，这也是为什么我要花这么多篇幅给你谈微服务的关键原因。

上节课我们将“待办任务”Web服务的后端，拆解为用户微服务和待办任务微服务。但为什么要这样拆？是凭感觉，还是有具体的方法论？这里你可以停下来想想。

微服务的拆解和合并，都有一个度需要把握，因为我们在一拆一合之间，都是有成本产生的。如果我们拆解得太细，就必然会导致我们的调用链路增长。调用链路变长，首先影响的就是网络延迟，这个好理解，毕竟你路远了，可能“堵车”的地方也会变多；其次是运维成本的增加，调用链路越长，整个链条就越脆弱，因为其中一环出现问题，都会导致整个调用链条访问失败，而且我们排查问题也变得更加困难。

反过来看，如果我们拆解得太粗，调用链路倒是短了，但是这个微服务的复用性就差了，更别提因为高耦合带来的复杂且冗余的数据库表结构，让我们后续难以维护。我画了个图，你感受下。

带环的是Stateless节点
实心的是Stateful节点



拆之，领域驱动设计

那我们要合理地拆解微服务，应该怎么拆解呢？上节课其实我有提到，目前主流的解决方案就是领域驱动设计，也叫DDD[1]。DDD是Eric Evans在其2004年的同名书中提出来的一个思想，但一直仅仅局限在Java的圈子里，直到2014年，微服务兴起后大家才发现它可以指导微服务的拆分，这才走进了大多数人的视野。用一句话简单总结，DDD就是一套方法论：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。

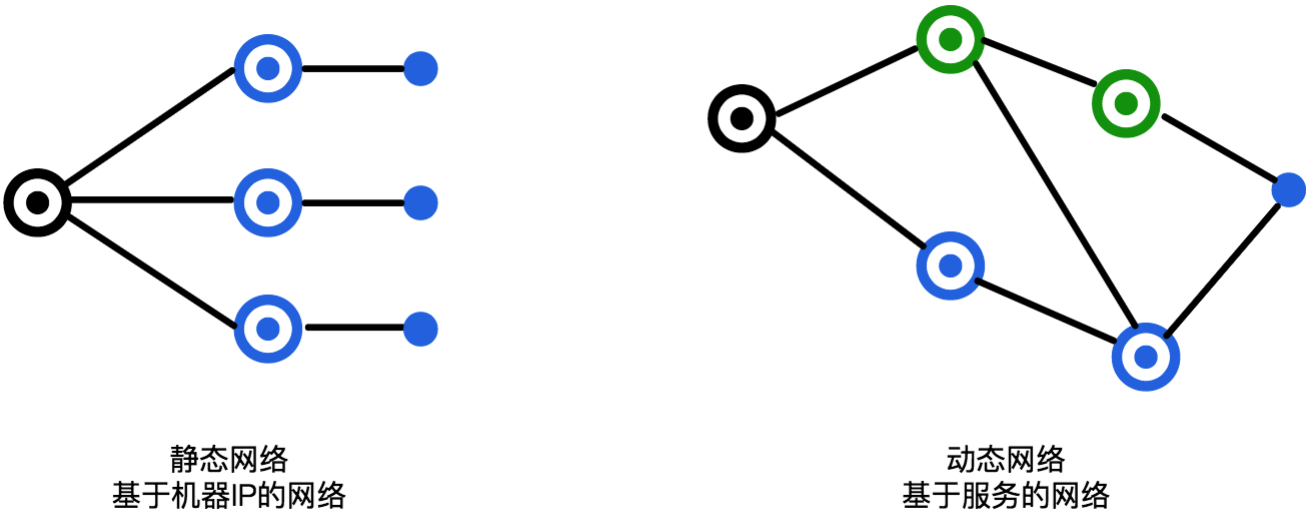
实际我自己在用微服务开发的过程发现，微服务整体应该是一个动态网络结构[2]，随着业务的发展，这个网络结构也会发生变化。DDD能帮助我们前期分析出一个较好的网络结构，但实际上，我们更应该思考的是如何整体优化动态网络：减少核心节点，保护核心节点，降低网络深度等等。

怎么理解动态网络优化呢？我们可以做个思维实验：假设我们将所有的功能都拆解成微服务，任意的微服务节点之间都可以相互调用，调用越频繁它们之间的距离就越近。那么我们考虑一下，当我们网站的访问请求流量稳定后，我们整个微服务节点组成的网络状态是怎么样的？

首先网络节点的相互制约总会让那些相互之间强依赖的、高耦合的节点，越走越近，最后聚集成一团节点。其次那些跟业务逻辑无关的节点，逐渐被边缘化，甚至消失。我们看这些聚集成团的节点，如果团里的点聚合太近了，其实是不适合拆分的，它们整体应该作成一个微服务。等这些节点太近的团合并成一个微服务节点后，我们再看那些聚集在一起、又不太近的节点就是一个微服务了。

所以，我们在启动项目时，不用太过纠结应该如何去拆解微服务。而应该持续关注，并思考每个微服务节点的合理性。就像看待动态网络一样，持续地调整优化，去除核心节点。最终它会伴随你业务的发展阶段，达到各个阶段的稳定动态网络结构。

就像我们上节课“待办任务”Web服务一样，我们可以先简单地将我们的项目后端分为：用户微服务和待办任务微服务。当然这里我们目前的业务太简单了，用DDD去分析，也是大材小用。随着我这个项目的业务发展，我们添加的功能会越来越多。让微服务根据业务一起成长演变就可以了。这并不是说我们就放任微服务不管了，而是从整体网络的角度思考，去看我们的微服务如何演进。



合之，Streaming

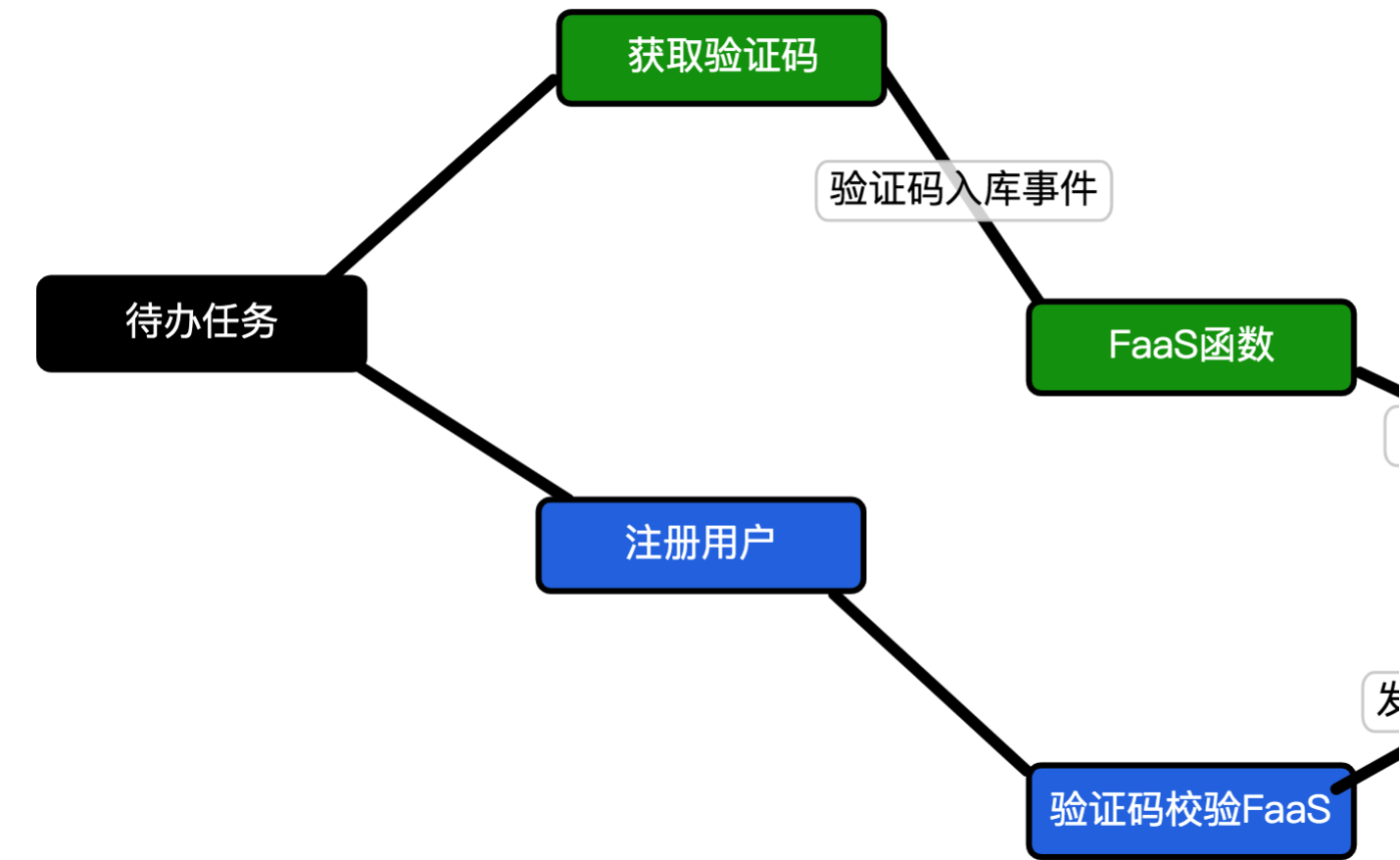
看完拆解，我们再看合并。合并呢，换个高大上的词其实就是前面课程中提到的编排。目前为止，我们整个“待办任务”Web应用架构的设计基本完成了，而且所有节点都是Stateless的了。变成Stateless节点后，其实对于前端的同学来说，一点都不陌生，比如React的单向数据流中的State也要求我们Immutable，Immutable其实就是Stateless。

我们上面已经看到了，拆解后的架构是个动态网络，那我们应该怎么合并或者编排呢？当然你像SFF那样通过传统的函数，将每个HTTP数据的请求结果通过数组或对象加工处理，再将这些结果返回也是可以的。但我在这里想向你介绍另外一种编排思路， workflow。



我们可以将用户的请求想象成我们的呼吸系统，我们的肺就是SFF，而微服务和FaaS节点就是需要氧气的各个器官。我们吸一口气，氧气进入肺部，血液循环将氧气按顺序流经我们每个器官，这就是请求链路。每个器官一接收到新鲜血液，就会吸取氧气返回二氧化碳，最终血液循环将二氧化碳带到肺部呼出，这个就是数据返回链路。我们的各个器官，就被请求链路通过新鲜血液到来的这个事件串联起来了，这个就是事件流，也就是用一个事件去串联FaaS或微服务。

现在我们用[第3课]讲的，PHP发邮件改造一下，举个例子。当用户注册时，我们完全可以将用户的信息和注册验证码存入数据库；PHP发邮件的FaaS触发器改为数据库插入新记录触发事件；用户从邮箱验证获取验证码，把验证码写到输入框后，点击验证，则是另一个HTTP触发器，触发FaaS函数校验验证码通过，修改数据库注册成功，并且返回302跳转到登录成功页面。具体流程可参考下图：



当然现在这个解决方案也有成熟对应的云BaaS服务：Serverless工作流[3]。

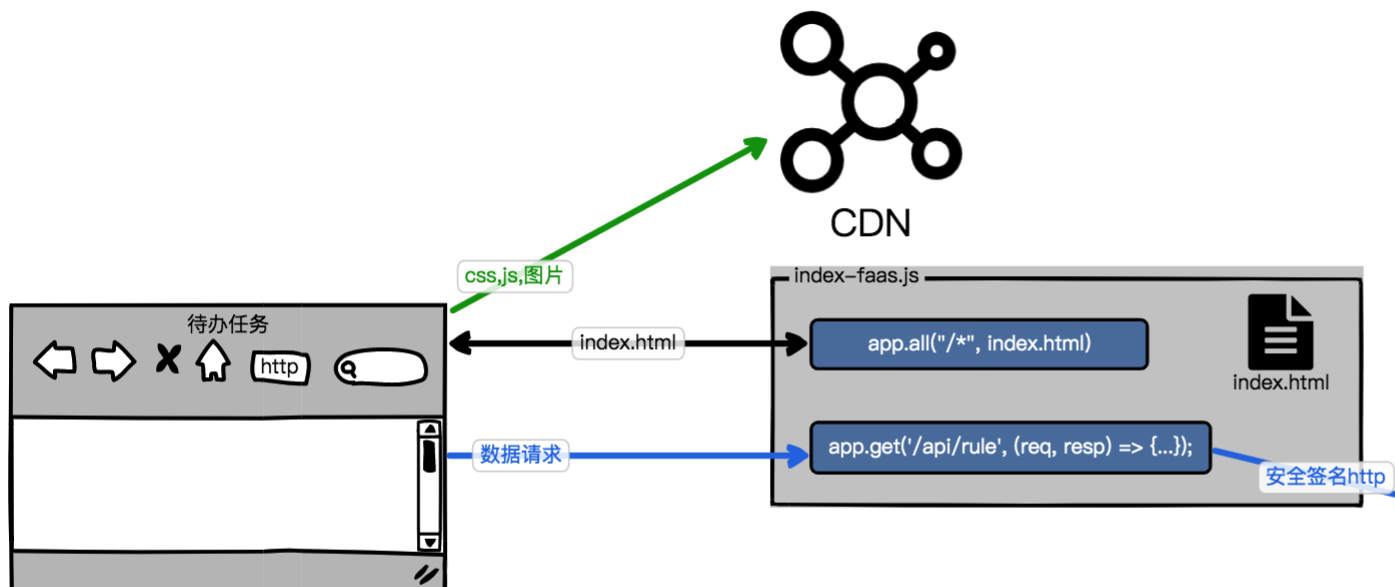
安全门神

理解了拆合的思想，我们就可以将目前“待办项目”的架构再演进一下：静态文件我们用CDN托管，前端项目只负责域名支撑和index.html，剩下的请求直接访问FaaS微服务。这时候，我估计你会问，咱们数据的安全性如何保障呢？是的，到目前为止，我们的FaaS都一直在用匿名模式访问，完全没有任何安全防护可言，也就是说目前我们FaaS服务的接口一直都在互联网上“裸奔”。

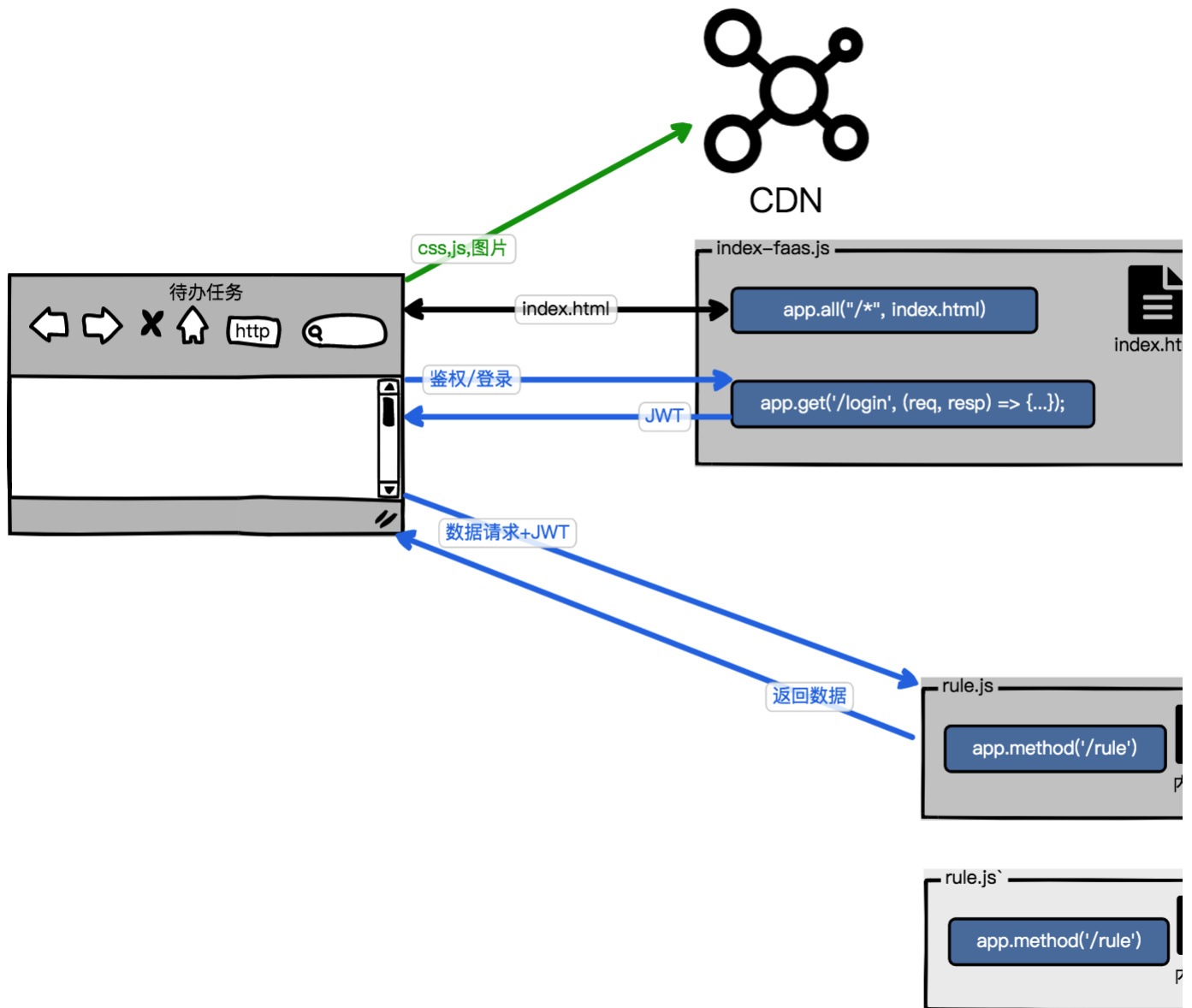
鉴权

其实，FaaS提供的安全防护通常是放在触发器上的。触发器的授权类型或认证方式我们可以设置为：匿名anonymous或函数function。匿名方式就是不需要签名认证，匿名的用户也能访问；而函数方式，则是需要签名认证[4]，这个签名认证的算法，参数需要用到我们账户的访问密钥ak/sk[5]，ak/sk相当于我们云账户的银行卡密码，这么重要的账户信息，我们只能限定在服务端使用，前端代码里绝对不可以出现。

也就是说，我们只能在服务端使用函数安全认证方式。如果是这种方案，我们的“待办任务”架构就演进成下图这样了。



那有没有针对匿名认证方式的安全策略呢？当然有，这里我们同样需要借鉴一下微服务的鉴权设计：JSON Web Token，简称JWT[6]。JWT简单来说，就是将用户身份信息和签名信息，一起传到客户端去了。用户在请求数据时，带上这个JWT，服务端通过校验签名确定用户身份。JWT存在于客户端，JWT验证只需要通过服务端的sk和算法验证签名。同样，我画了张图，以帮助你理解。



要解决后端互调的安全性，我们用VPC或IP白名单，都很容易解决。比较难处理的是前后端的信任问题，JWT正好就提供了一种信任链的解决思路。当然，关于鉴权也有一些云服务商推出了一些更加安全易用的BaaS服务，例如AWS的IAM和Cognito[7]。

安全性是我们考虑架构设计时重要的一环，因为安全架构设计的失败，会直接导致我们资产的损失。鉴权是识别用户身份，防止用户信息泄漏和恶意攻击使用的。但根据我统计的数据，我们在日常99%的问题，都发生在新版本上线的环节。

那我们该怎么稳定持续地快速迭代，发布新版本上线呢？我们可以回想一下[第1课]，小程和小服的例子，小程最后实现NoOps后，小服则只要将代码合并到指定分支就可以发布上线了。那现实中，这点该怎么实现呢？

当我们的项目Serverless化以后，代码的质量变得尤为重要。你可以想想，Serverless化之前，你不小心上线了一个bug，影响的范围最大也就只有一个应用。但是Serverless化之后，如果是核心节点发布了严重的bug上线，那么影响的范围就是所有依赖它的线上应用了。

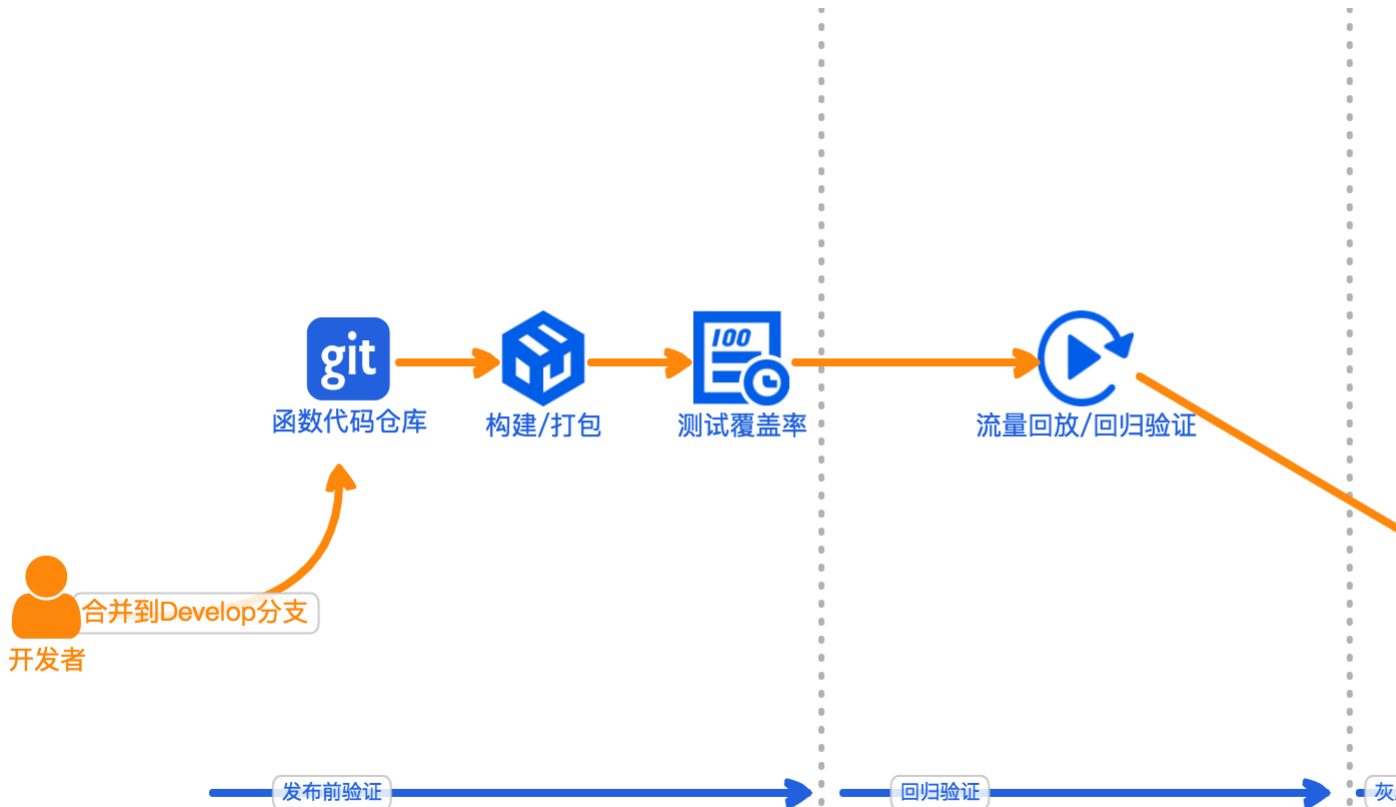
不过，你也不用太担心，微服务和FaaS都具备快速独立迭代的能力。以前我们一个应用的迭代周期通常要一周到两周。但对于Serverless化后的应用来说，每个节点借助独立运维的特性，可以随时随地的发布上线。

综上，我们知道了，微服务和FaaS都是快速迭代的，修复问题很快，但我们也不能每次都等问题出现，再去依赖这个能力呀。有没有什么办法可以提前发现问题，保证我们又快又稳？目前软件工程的最佳做法就是代码流水线的发布管道。

发布管道

发布管道的流水线主要有3个部分：

1. 代码发布前的验证，代码测试覆盖率CI/CD；
2. 模拟流量回归测试通过，发布到灰度环境；
3. 代码正式上线，灰度环境替换正式环境。流水线的每个节点产生的结果，都会作为下一个节点必要的启动参数。



我们先看看上图，我来解释下这个流程。

- 我们的代码合并到指定分支后，通常我会用Develop分支。
- Git的钩子就会触发后续的流水线，开始进入构建打包、测试流程。
- 测试节点做的事情就是跑所有测试Case，并且统计覆盖率。
- 覆盖率验证通过，代码实例用录制流量模拟验证。
- 模拟验证通过，发布代码实例到灰度环境。
- 线上根据灰度策略，将小部分流量导入灰度环境验证灰度版本。
- 在灰度窗口期，比如两个小时，灰度验证没有异常则用灰度版本替换正式版本；反之则立即丢弃这个灰度版本，止损。

这套流程，目前规模大一些的互联网公司发布流程基本都在这样跑，如果你不是很了解，可以自己尝试用我们介绍的Serverless工作流或者云服务商提供的工作流工具[8]动手搭建下。

在这套流程的基础上，很多企业为了追求更高的稳定性，还会设定环境隔离的流水线和安全卡口。比如隔离测试环境和线上环境，测试环境用来复现故障。每次代码进入发布管道，都必须先在测试环境跑通，跑通后安全卡口放行，才能进入线上环境的流水线。

总结

这节课，我们继续讲后端的BaaS化。我们再梳理一下这节课的重要知识点吧。

1. 如何拆解BaaS应用，我们学习了微服务的重要拆解思想DDD：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。另外我也介绍了另一种更适合初创企业的拆分思路：动态网络演进。
2. 拆解完之后，我们就要考虑合并。这里我们介绍了代码编排以外的另一种编排方式：事件流编排，它就是通过一个个事件顺序将我们的微服务或FaaS串联起来。
3. 为了解决拆解后，微服务之间的信任问题。我们先了解了FaaS触发器的安全方案：数字签名。还借鉴了微服务的鉴权做法JWT，将用户鉴权加密信息放在客户端，让鉴权服务变成Stateless。最后，为了让微服务又快又稳地发布版本，我们借鉴了微服务的发布管道：打造自动灰度流水线。

作业

这节课的作业就是我们JWT鉴权的“待办任务”Web应用，你来部署上线。

后端代码GitHub地址：<https://github.com/pusongyang/todolist-backend/tree/lesson06>

前端代码GitHub地址：<https://github.com/pusongyang/todolist-frontend>

演示预览地址：<http://lesson6.jike-serverless.online/list>

期待你的作业，如果今天的内容让你有所收获，也欢迎你把文章分享给身边的朋友，邀请他加入学习。

参考资料

[1] https://en.wikipedia.org/wiki/Domain-driven_design

[2] https://en.wikipedia.org/wiki/Dynamic_network_analysis

[3] <https://www.aliyun.com/product/fnf>

[4] <https://github.com/alibaba/aliyun-ic-nodejs-sdk/blob/master/lib/client.js?spm=a2c4g.11186623.2.15.16e016d7lo8NBQ#L840>

[5] https://help.aliyun.com/document_detail/154851.html?spm=5176.2020520153.0.0.371a415dLXylZ

[6] <https://jwt.io/>

[7] https://docs.aws.amazon.com/zh_cn/cognito/latest/developerguide/what-is-amazon-cognito.html

[8] <https://www.aliyun.com/product/yunxiao/devops?spm=5176.10695662.1173276.1.6c724a38akCjgo>

你好，我是秦粤。上一课中，我们学习了后端BaaS化的重要模块：微服务。现在我们知道微服务的核心理念就是先拆后合，拆解功能是为了提升我们功能的利用率。同步我们也了解了实现微服务的10要素，这10要素要真讲起来够单独开一门课的。如果你不熟悉，我向你推荐杨波老师的《微服务架构核心20讲》课程。

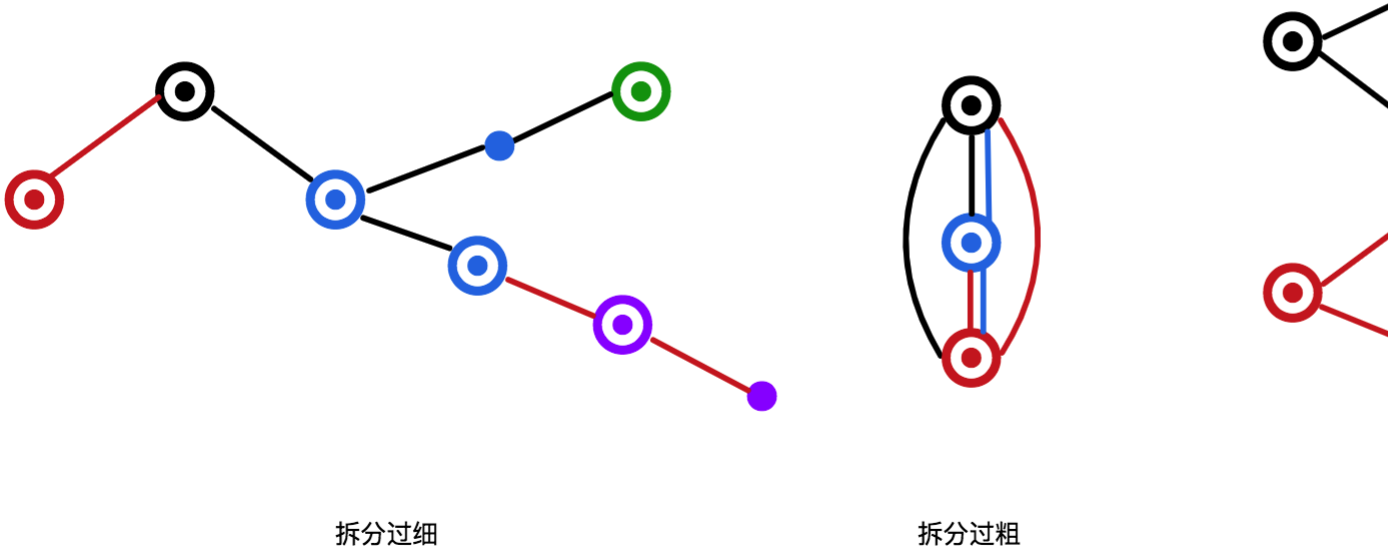
BaaS化的核心其实就是把我们的后端应用封装成RESTful API，然后对外提供服务，而为了后端应用更容易维护，我们需要将后端应用拆解成免运维的微服务。这个逻辑你要理解，这也是为什么我要花这么多篇幅给你谈微服务的关键原因。

上节课我们将“待办任务”Web服务的后端，拆解为用户微服务和待办任务微服务。但为什么要这样拆？是凭感觉，还是有具体的方法论？这里你可以停下来想想。

微服务的拆解和合并，都有一个度需要把握，因为我们在一拆一合之间，都是有成本产生的。如果我们拆解得太细，就必然会导致我们的调用链路增长。调用链路变长，首先影响的就是网络延迟，这个好理解，毕竟你路远了，可能“堵车”的地方也会变多；其次是运维成本的增加，调用链路越长，整个链条就越脆弱，因为其中一环出现问题，都会导致整个调用链条访问失败，而且我们排查问题也变得更加困难。

反过来看，如果我们拆解得太粗，调用链路倒是短了，但是这个微服务的复用性就差了，更别提因为高耦合带来的复杂且冗余的数据库表结构，让我们后续难以维护。我画了个图，你感受下。

带环的是Stateless节点
实心的是Stateful节点



拆之，领域驱动设计

那我们要合理地拆解微服务，应该怎么拆解呢？上节课其实我有提到，目前主流的解决方案就是领域驱动设计，也叫DDD[1]。DDD是Eric Evans在其2004年的同名书中提出来的一个思想，但一直仅仅局限在Java的圈子里，直到2014年，微服务兴起后大家才发现它可以指导微服务的拆分，这才走进了大多数人的视野。用一句话简单总结，DDD就是一套方法论：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。

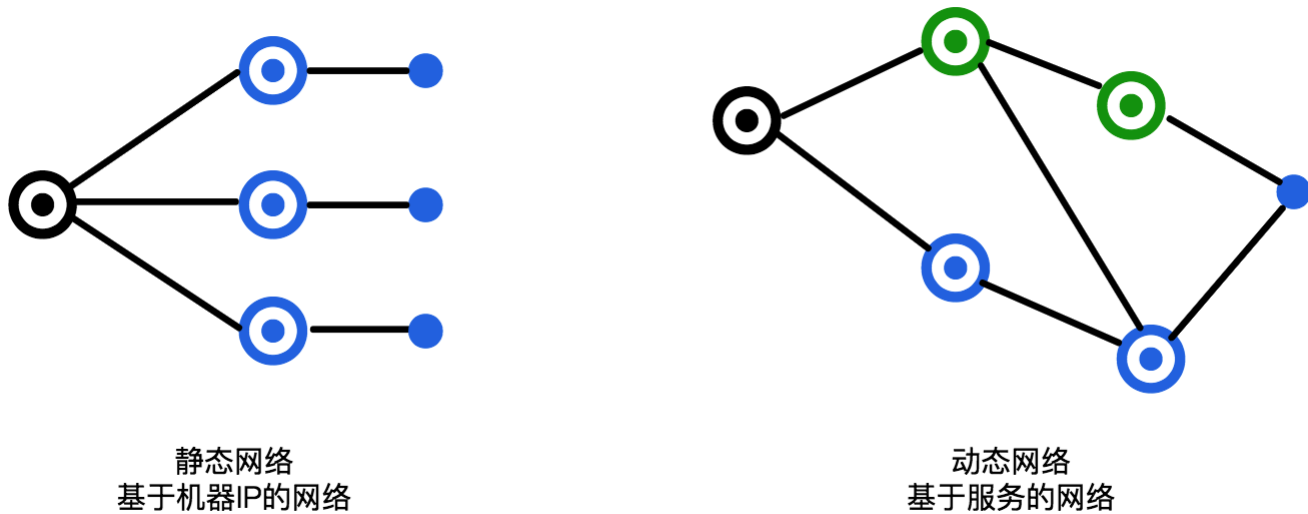
实际我自己在用微服务开发的过程发现，微服务整体应该是一个动态网络结构[2]，随着业务的发展，这个网络结构也会发生变化。DDD能帮助我们前期分析出一个较好的网络结构，但实际上，我们更应该思考的是如何整体优化动态网络：减少核心节点，保护核心节点，降低网络深度等等。

怎么理解动态网络优化呢？我们可以做个思维实验：假设我们将所有的功能都拆解成微服务，任意的微服务节点之间都可以相互调用，调用越频繁它们之间的距离就越近。那么我们考虑一下，当我们网站的访问请求流量稳定后，我们整个微服务节点组成的网络状态是怎么样的？

首先网络节点的相互制约总会让那些相互之间强依赖的、高耦合的节点，越走越近，最后聚集成一团节点。其次那些跟业务逻辑无关的节点，逐渐被边缘化，甚至消失。我们看这些聚集成团的节点，如果团里的点聚合太近了，其实是不适合拆分的，它们整体应该作成一个微服务。等这些节点太近的团合并成一个微服务节点后，我们再看那些聚集在一起、又不太近的节点就是一个微服务了。

所以，我们在启动项目时，不用太过纠结应该如何去拆解微服务。而应该持续关注，并思考每个微服务节点的合理性。就像看待动态网络一样，持续地调整优化，去除核心节点。最终它会伴随你业务的发展阶段，达到各个阶段的稳定动态网络结构。

就像我们上节课“待办任务”Web服务一样，我们可以先简单地将我们的项目后端分为：用户微服务和待办任务微服务。当然这里我们目前的业务太简单了，用DDD去分析，也是大材小用。随着我这个项目的业务发展，我们添加的功能会越来越多。让微服务根据业务一起成长演变就可以了。这并不是说我们就放任微服务不管了，而是从整体网络的角度思考，去看我们的微服务如何演进。



合之，Streaming

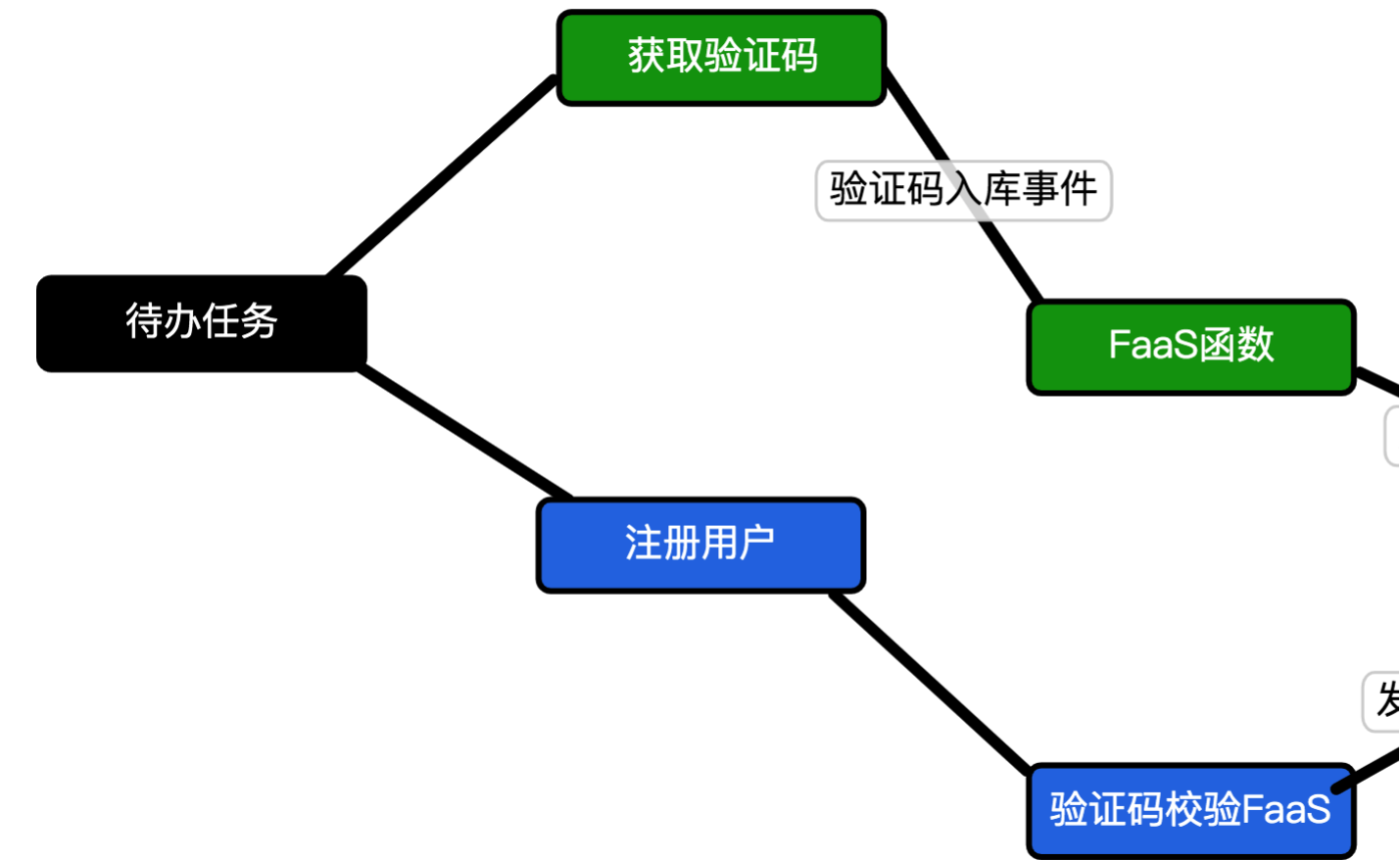
看完拆解，我们再看合并。合并呢，换个高大上的词其实就是前面课程中提到的编排。目前为止，我们整个“待办任务”Web应用架构的设计基本完成了，而且所有节点都是Stateless的了。变成Stateless节点后，其实对于前端的同学来说，一点都不陌生，比如React的单向数据流中的State也要求我们Immutable，Immutable其实就是Stateless。

我们上面已经看到了，拆解后的架构是个动态网络，那我们应该怎么合并或者编排呢？当然你像SFF那样通过传统的函数，将每个HTTP数据的请求结果通过数组或对象加工处理，再将这些结果返回也是可以的。但我在这里想向你介绍另外一种编排思路， workflow。



我们可以将用户的请求想象成我们的呼吸系统，我们的肺就是SFF，而微服务和FaaS节点就是需要氧气的各个器官。我们吸一口气，氧气进入肺部，血液循环将氧气按顺序流经我们每个器官，这就是请求链路。每个器官一接收到新鲜血液，就会吸取氧气返回二氧化碳，最终血液循环将二氧化碳带到肺部呼出，这个就是数据返回链路。我们的各个器官，就被请求链路通过新鲜血液到来的这个事件串联起来了，这个就是事件流，也就是用一个事件去串联FaaS或微服务。

现在我们用[第3课]讲的，PHP发邮件改造一下，举个例子。当用户注册时，我们完全可以将用户的信息和注册验证码存入数据库；PHP发邮件的FaaS触发器改为数据库插入新记录触发事件；用户从邮箱验证获取验证码，把验证码写到输入框后，点击验证，则是另一个HTTP触发器，触发FaaS函数校验验证码通过，修改数据库注册成功，并且返回302跳转到登录成功页面。具体流程可参考下图：



当然现在这个解决方案也有成熟对应的云BaaS服务：Serverless workflow[3]。

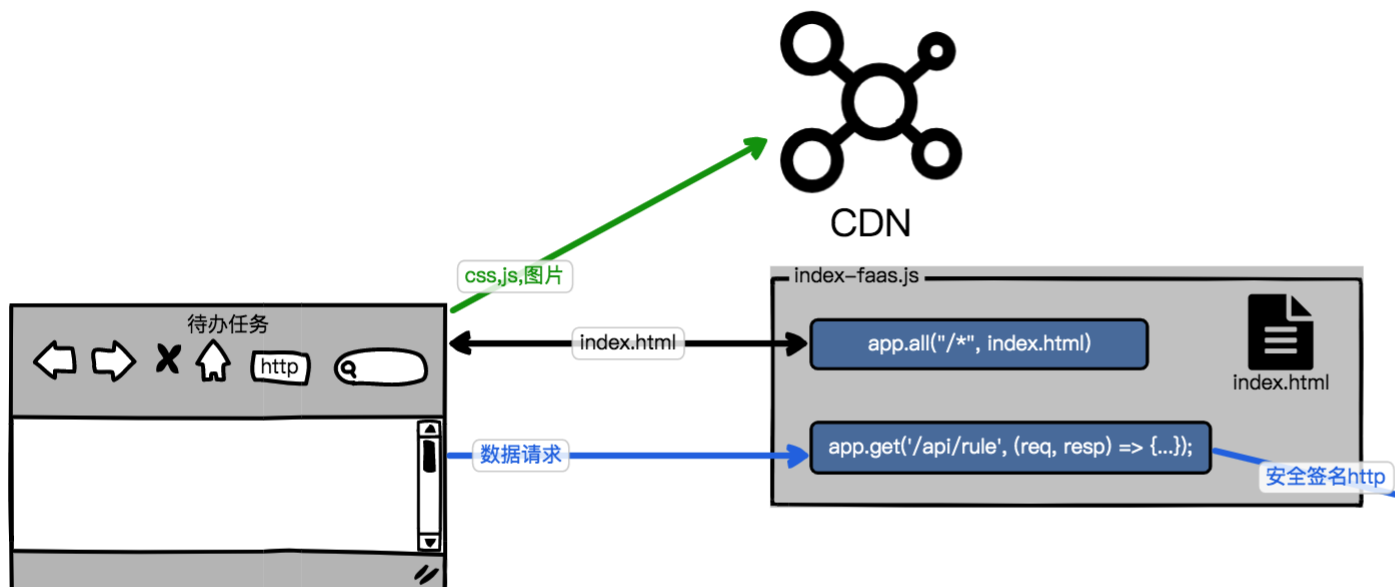
安全门神

理解了拆合的思想，我们就可以将目前“待办项目”的架构再演进一下：静态文件我们用CDN托管，前端项目只负责域名支撑和index.html，剩下的请求直接访问FaaS微服务。这时候，我估计你会问，咱们数据的安全性如何保障呢？是的，到目前为止，我们的FaaS都一直在用匿名模式访问，完全没有任何安全防护可言，也就是说目前我们FaaS服务的接口一直都在互联网上“裸奔”。

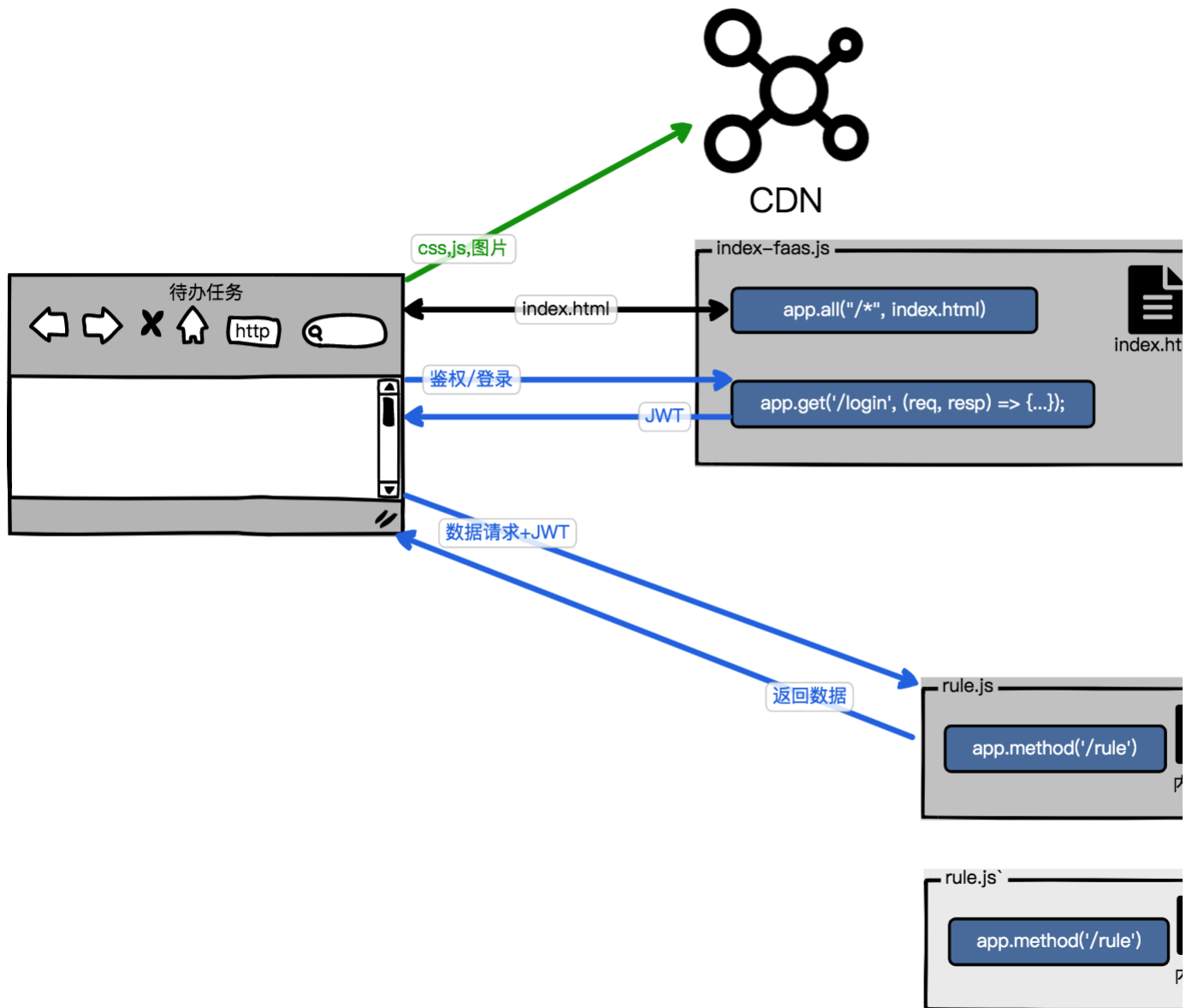
鉴权

其实，FaaS提供的安全防护通常是放在触发器上的。触发器的授权类型或认证方式我们可以设置为：匿名anonymous或函数function。匿名方式就是不需要签名认证，匿名的用户也能访问；而函数方式，则是需要签名认证[4]，这个签名认证的算法，参数需要用到我们账户的访问密钥ak/sk[5]，ak/sk相当于我们云账户的银行卡密码，这么重要的账户信息，我们只能限定在服务端使用，前端代码里绝对不可以出现。

也就是说，我们只能在服务端使用函数安全认证方式。如果是这种方案，我们的“待办任务”架构就演进成下图这样了。



那有没有针对匿名认证方式的安全策略呢？当然有，这里我们同样需要借鉴一下微服务的鉴权设计：JSON Web Token，简称JWT[6]。JWT简单来说，就是将用户身份信息和签名信息，一起传到客户端去了。用户在请求数据时，带上这个JWT，服务端通过校验签名确定用户身份。JWT存在于客户端，JWT验证只需要通过服务端的sk和算法验证签名。同样，我画了张图，以帮助你理解。



要解决后端互调的安全性，我们用VPC或IP白名单，都很容易解决。比较难处理的是前后端的信任问题，JWT正好就提供了一种信任链的解决思路。当然，关于鉴权也有一些云服务商推出了一些更加安全易用的BaaS服务，例如AWS的IAM和Cognito[7]。

安全性是我们考虑架构设计时重要的一环，因为安全架构设计的失败，会直接导致我们资产的损失。鉴权是识别用户身份，防止用户信息泄漏和恶意攻击使用的。但根据我统计的数据，我们在日常99%的问题，都发生在新版本上线的环节。

那我们该怎么稳定持续地快速迭代，发布新版本上线呢？我们可以回想一下[第1课]，小程和小服的例子，小程最后实现NoOps后，小服则只要将代码合并到指定分支就可以发布上线了。那现实中，这点该怎么实现呢？

当我们的项目Serverless化以后，代码的质量变得尤为重要。你可以想想，Serverless化之前，你不小心上线了一个bug，影响的范围最大也就只有一个应用。但是Serverless化之后，如果是核心节点发布了严重的bug上线，那么影响的范围就是所有依赖它的线上应用了。

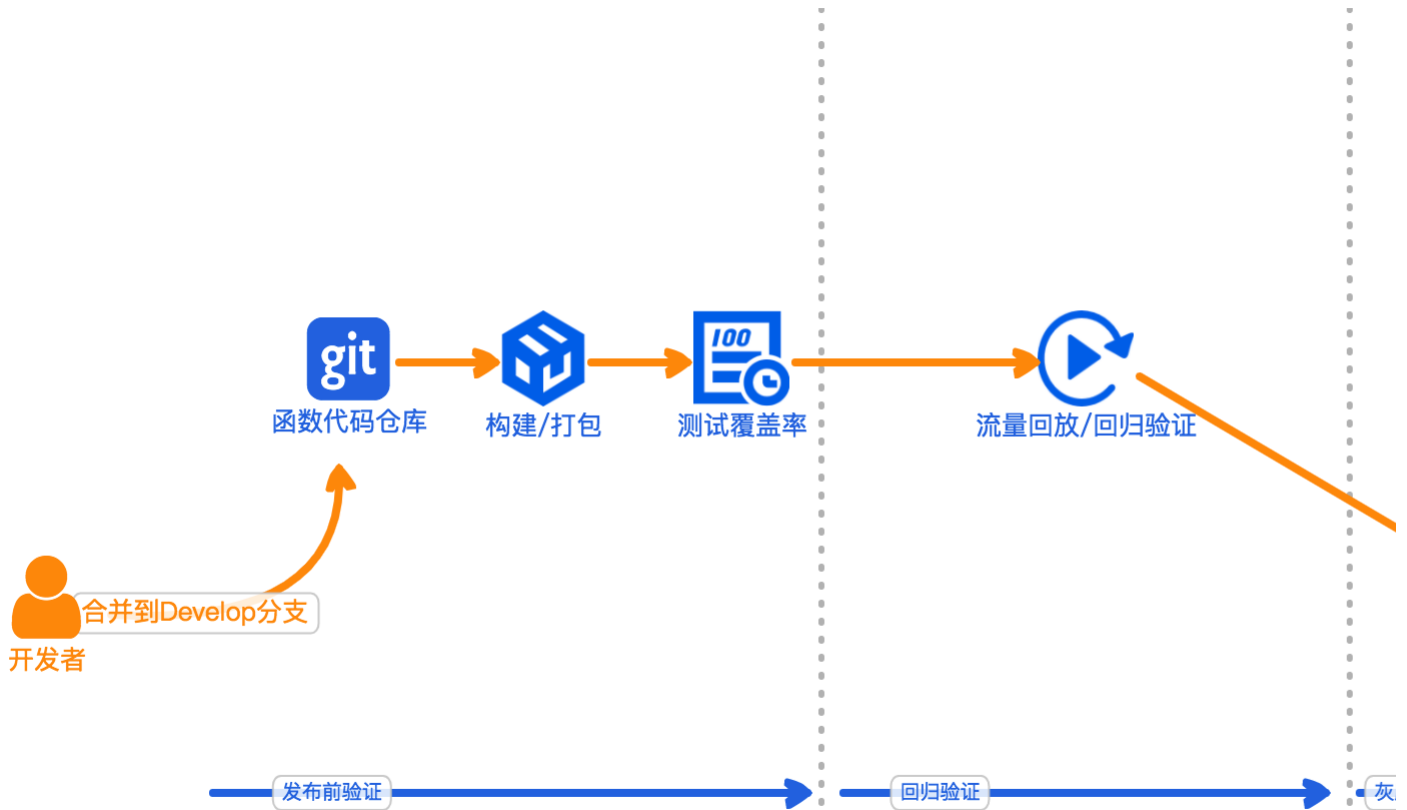
不过，你也不用太担心，微服务和FaaS都具备快速独立迭代的能力。以前我们一个应用的迭代周期通常要一周到两周。但对于Serverless化后的应用来说，每个节点借助独立运维的特性，可以随时随地的发布上线。

综上，我们知道了，微服务和FaaS都是快速迭代的，修复问题很快，但我们也不能每次都等问题出现，再去依赖这个能力呀。有没有什么办法可以提前发现问题，保证我们又快又稳？目前软件工程的最佳做法就是代码流水线的发布管道。

发布管道

发布管道的流水线主要有3个部分：

1. 代码发布前的验证，代码测试覆盖率CI/CD；
2. 模拟流量回归测试通过，发布到灰度环境；
3. 代码正式上线，灰度环境替换正式环境。流水线的每个节点产生的结果，都会作为下一个节点必要的启动参数。



我们先看看上图，我来解释下这个流程。

- 我们的代码合并到指定分支后，通常我会用Develop分支。
- Git的钩子就会触发后续的流水线，开始进入构建打包、测试流程。
- 测试节点做的事情就是跑所有测试Case，并且统计覆盖率。
- 覆盖率验证通过，代码实例用录制流量模拟验证。
- 模拟验证通过，发布代码实例到灰度环境。
- 线上根据灰度策略，将小部分流量导入灰度环境验证灰度版本。
- 在灰度窗口期，比如两个小时，灰度验证没有异常则用灰度版本替换正式版本；反之则立即丢弃这个灰度版本，止损。

这套流程，目前规模大一些的互联网公司发布流程基本都在这样跑，如果你不是很了解，可以自己尝试用我们介绍的Serverless工作流或者云服务商提供的工作流工具[8]动手搭建下。

在这套流程的基础上，很多企业为了追求更高的稳定性，还会设定环境隔离的流水线和安全卡口。比如隔离测试环境和线上环境，测试环境用来复现故障。每次代码进入发布管道，都必须先在测试环境跑通，跑通后安全卡口放行，才能进入线上环境的流水线。

总结

这节课，我们继续讲后端的BaaS化。我们再梳理一下这节课的重要知识点吧。

1. 如何拆解BaaS应用，我们学习了微服务的重要拆解思想DDD：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。另外我也介绍了另一种更适合初创企业的拆分思路：动态网络演进。
2. 拆解完之后，我们就要考虑合并。这里我们介绍了代码编排以外的另一种编排方式：事件流编排，它就是通过一个个事件顺序将我们的微服务或FaaS串联起来。
3. 为了解决拆解后，微服务之间的信任问题。我们先了解了FaaS触发器的安全方案：数字签名。还借鉴了微服务的鉴权做法JWT，将用户鉴权加密信息放在客户端，让鉴权服务变成Stateless。最后，为了让微服务又快又稳地发布版本，我们借鉴了微服务的发布管道：打造自动灰度流水线。

作业

这节课的作业就是我们JWT鉴权的“待办任务”Web应用，你来部署上线。

后端代码GitHub地址：<https://github.com/pusongyang/todolist-backend/tree/lesson06>

前端代码GitHub地址：<https://github.com/pusongyang/todolist-frontend>

演示预览地址：<http://lesson6.jike-serverless.online/list>

期待你的作业，如果今天的内容让你有所收获，也欢迎你把文章分享给身边的朋友，邀请他加入学习。

参考资料

[1] https://en.wikipedia.org/wiki/Domain-driven_design

[2] https://en.wikipedia.org/wiki/Dynamic_network_analysis

[3] <https://www.aliyun.com/product/fnf>

[4] <https://github.com/alibaba/aliyun-ic-nodejs-sdk/blob/master/lib/client.js?spm=a2c4g.11186623.2.15.16e016d7lo8NBQ#L840>

[5] https://help.aliyun.com/document_detail/154851.html?spm=5176.2020520153.0.0.371a415dLXylZ

[6] <https://jwt.io/>

[7] https://docs.aws.amazon.com/zh_cn/cognito/latest/developerguide/what-is-amazon-cognito.html

[8] <https://www.aliyun.com/product/yunxiao/devops?spm=5176.10695662.1173276.1.6c724a38akCjgo>

你好，我是秦粤。上一课中，我们学习了后端BaaS化的重要模块：微服务。现在我们知道微服务的核心理念就是先拆后合，拆解功能是为了提升我们功能的利用率。同步我们也了解了实现微服务的10要素，这10要素要真讲起来够单独开一门课的。如果你不熟悉，我向你推荐杨波老师的《微服务架构核心20讲》课程。

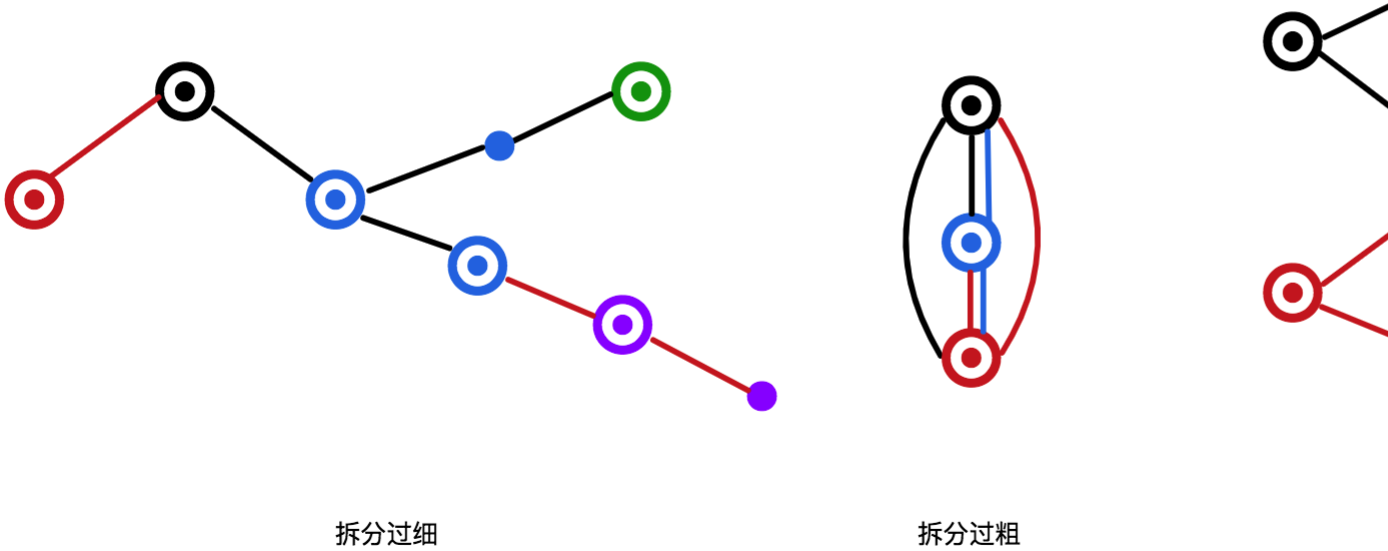
BaaS化的核心其实就是把我们的后端应用封装成RESTful API，然后对外提供服务，而为了后端应用更容易维护，我们需要将后端应用拆解成免运维的微服务。这个逻辑你要理解，这也是为什么我要花这么多篇幅给你谈微服务的关键原因。

上节课我们将“待办任务”Web服务的后端，拆解为用户微服务和待办任务微服务。但为什么要这样拆？是凭感觉，还是有具体的方法论？这里你可以停下来想想。

微服务的拆解和合并，都有一个度需要把握，因为我们在一拆一合之间，都是有成本产生的。如果我们拆解得太细，就必然会导致我们的调用链路增长。调用链路变长，首先影响的就是网络延迟，这个好理解，毕竟你路远了，可能“堵车”的地方也会变多；其次是运维成本的增加，调用链路越长，整个链条就越脆弱，因为其中一环出现问题，都会导致整个调用链条访问失败，而且我们排查问题也变得更加困难。

反过来看，如果我们拆解得太粗，调用链路倒是短了，但是这个微服务的复用性就差了，更别提因为高耦合带来的复杂且冗余的数据库表结构，让我们后续难以维护。我画了个图，你感受下。

带环的是Stateless节点
实心的是Stateful节点



拆之，领域驱动设计

那我们要合理地拆解微服务，应该怎么拆解呢？上节课其实我有提到，目前主流的解决方案就是领域驱动设计，也叫DDD[1]。DDD是Eric Evans在其2004年的同名书中提出来的一个思想，但一直仅仅局限在Java的圈子里，直到2014年，微服务兴起后大家才发现它可以指导微服务的拆分，这才走进了大多数人的视野。用一句话简单总结，DDD就是一套方法论：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。

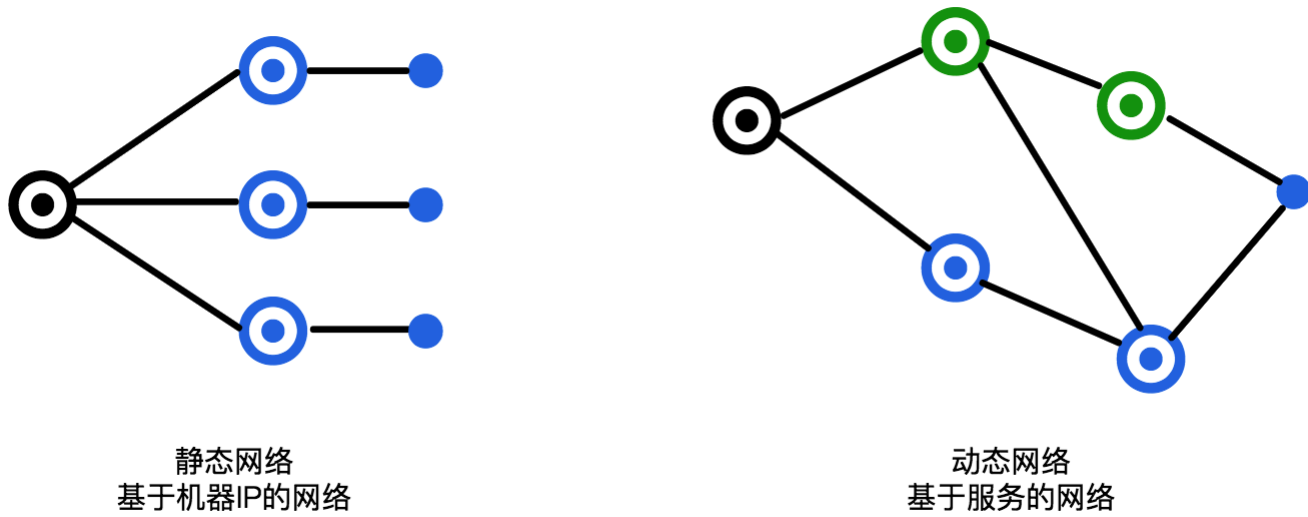
实际我自己在用微服务开发的过程发现，微服务整体应该是一个动态网络结构[2]，随着业务的发展，这个网络结构也会发生变化。DDD能帮助我们前期分析出一个较好的网络结构，但实际上，我们更应该思考的是如何整体优化动态网络：减少核心节点，保护核心节点，降低网络深度等等。

怎么理解动态网络优化呢？我们可以做个思维实验：假设我们将所有的功能都拆解成微服务，任意的微服务节点之间都可以相互调用，调用越频繁它们之间的距离就越近。那么我们考虑一下，当我们网站的访问请求流量稳定后，我们整个微服务节点组成的网络状态是怎么样的？

首先网络节点的相互制约总会让那些相互之间强依赖的、高耦合的节点，越走越近，最后聚集成一团节点。其次那些跟业务逻辑无关的节点，逐渐被边缘化，甚至消失。我们看这些聚集成团的节点，如果团里的点聚合太近了，其实是不适合拆分的，它们整体应该作成一个微服务。等这些节点太近的团合并成一个微服务节点后，我们再看那些聚集在一起、又不太近的节点就是一个微服务了。

所以，我们在启动项目时，不用太过纠结应该如何去拆解微服务。而应该持续关注，并思考每个微服务节点的合理性。就像看待动态网络一样，持续地调整优化，去除核心节点。最终它会伴随你业务的发展阶段，达到各个阶段的稳定动态网络结构。

就像我们上节课“待办任务”Web服务一样，我们可以先简单地将我们的项目后端分为：用户微服务和待办任务微服务。当然这里我们目前的业务太简单了，用DDD去分析，也是大材小用。随着我这个项目的业务发展，我们添加的功能会越来越多。让微服务根据业务一起成长演变就可以了。这并不是说我们就放任微服务不管了，而是从整体网络的角度思考，去看我们的微服务如何演进。



合之，Streaming

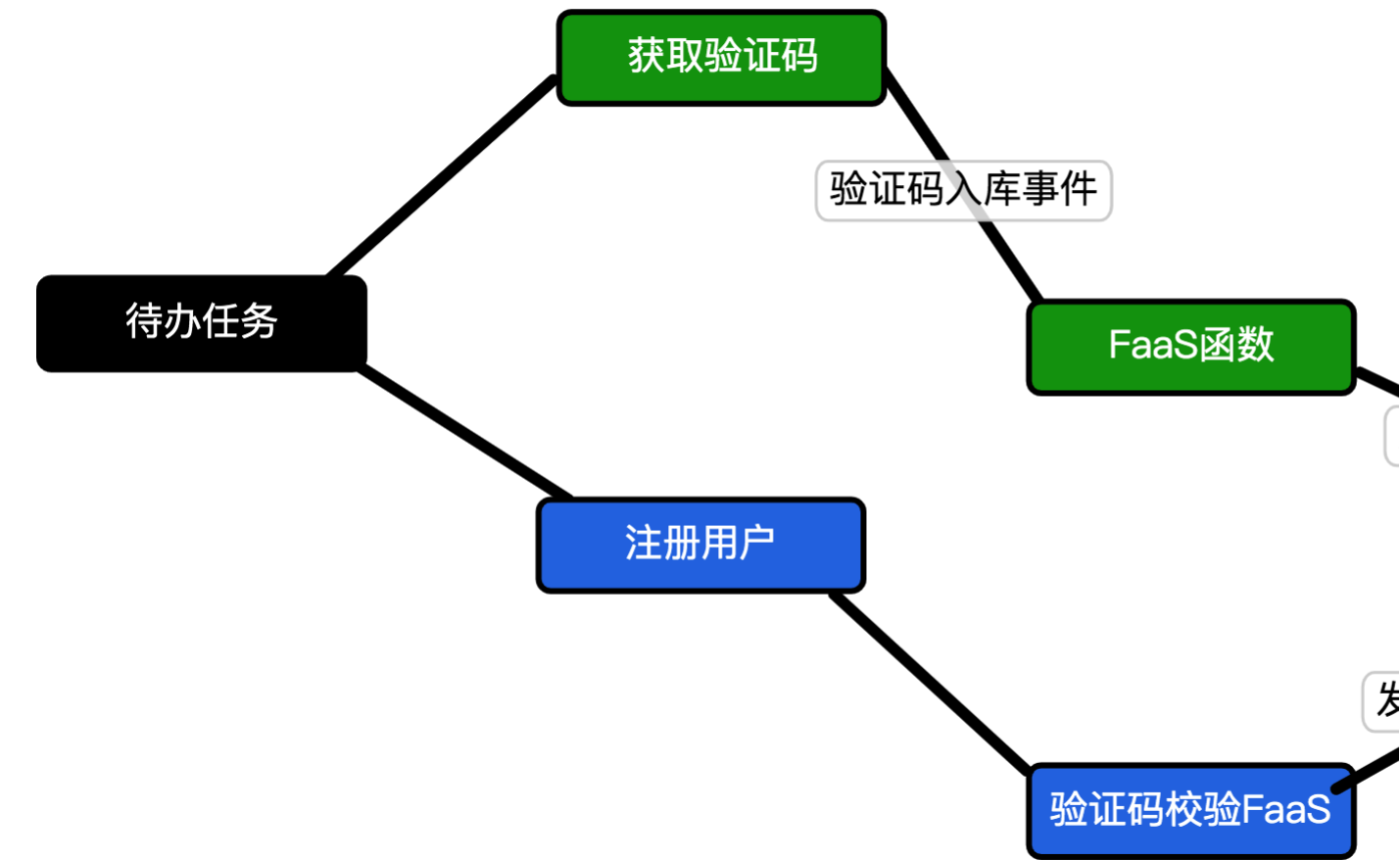
看完拆解，我们再看合并。合并呢，换个高大上的词其实就是前面课程中提到的编排。目前为止，我们整个“待办任务”Web应用架构的设计基本完成了，而且所有节点都是Stateless的了。变成Stateless节点后，其实对于前端的同学来说，一点都不陌生，比如React的单向数据流中的State也要求我们Immutable，Immutable其实就是Stateless。

我们上面已经看到了，拆解后的架构是个动态网络，那我们应该怎么合并或者编排呢？当然你像SFF那样通过传统的函数，将每个HTTP数据的请求结果通过数组或对象加工处理，再将这些结果返回也是可以的。但我在这里想向你介绍另外一种编排思路， workflow。



我们可以将用户的请求想象成我们的呼吸系统，我们的肺就是SFF，而微服务和FaaS节点就是需要氧气的各个器官。我们吸一口气，氧气进入肺部，血液循环将氧气按顺序流经我们每个器官，这就是请求链路。每个器官一接收到新鲜血液，就会吸取氧气返回二氧化碳，最终血液循环将二氧化碳带到肺部呼出，这个就是数据返回链路。我们的各个器官，就被请求链路通过新鲜血液到来的这个事件串联起来了，这个就是事件流，也就是用一个事件去串联FaaS或微服务。

现在我们用[第3课]讲的，PHP发邮件改造一下，举个例子。当用户注册时，我们完全可以将用户的信息和注册验证码存入数据库；PHP发邮件的FaaS触发器改为数据库插入新记录触发事件；用户从邮箱验证获取验证码，把验证码写到输入框后，点击验证，则是另一个HTTP触发器，触发FaaS函数校验验证码通过，修改数据库注册成功，并且返回302跳转到登录成功页面。具体流程可参考下图：



当然现在这个解决方案也有成熟对应的云BaaS服务：Serverless工作流[3]。

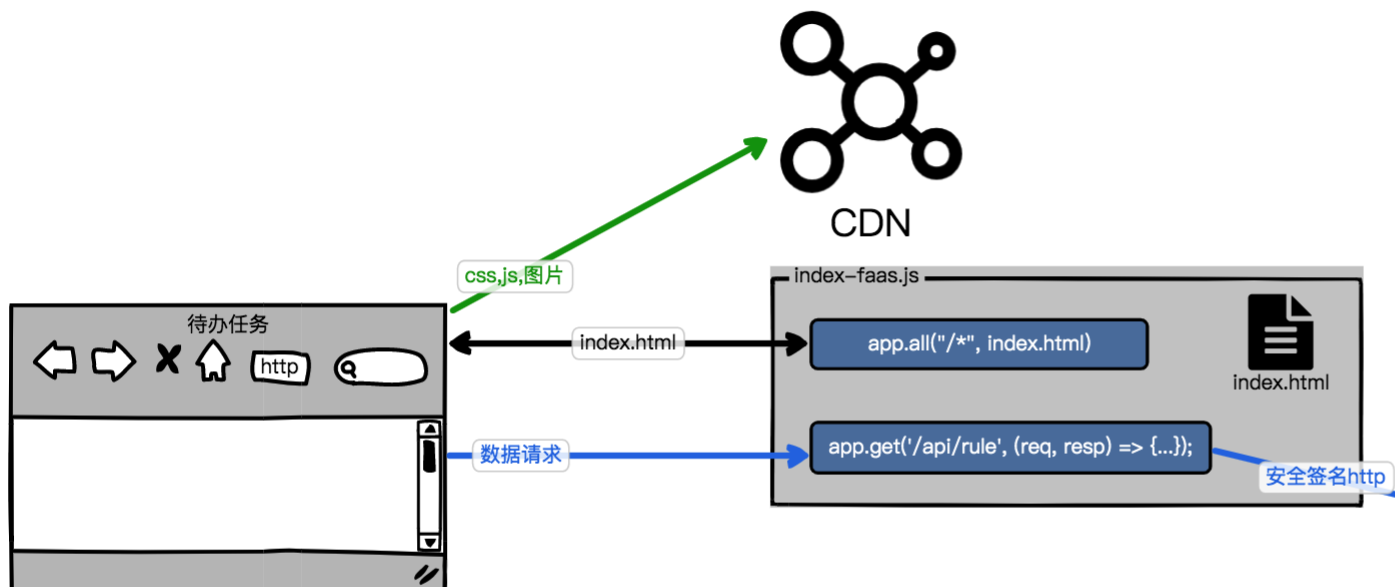
安全门神

理解了拆合的思想，我们就可以将目前“待办项目”的架构再演进一下：静态文件我们用CDN托管，前端项目只负责域名支撑和index.html，剩下的请求直接访问FaaS微服务。这时候，我估计你会问，咱们数据的安全性如何保障呢？是的，到目前为止，我们的FaaS都一直在用匿名模式访问，完全没有任何安全防护可言，也就是说目前我们FaaS服务的接口一直都在互联网上“裸奔”。

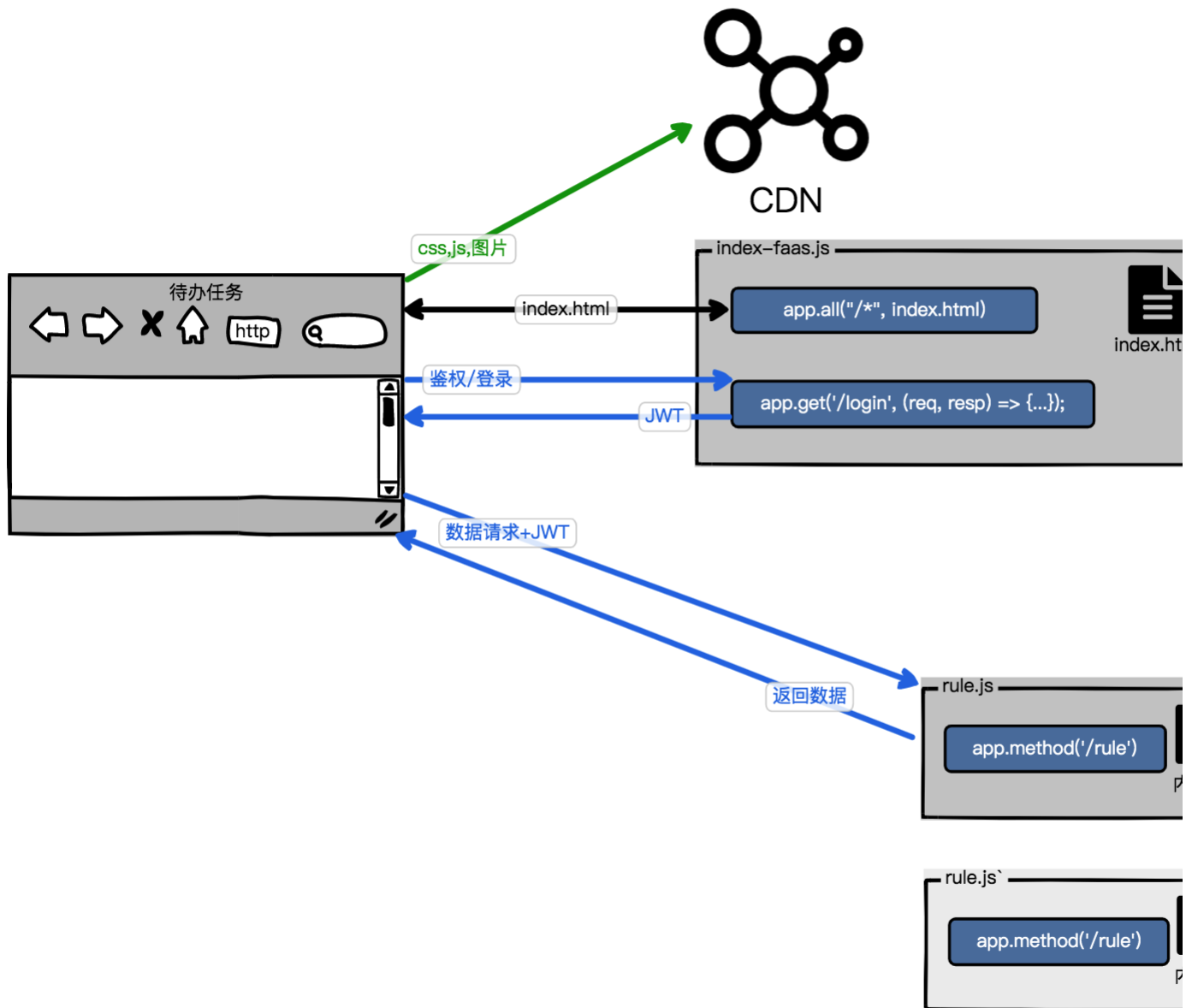
鉴权

其实，FaaS提供的安全防护通常是放在触发器上的。触发器的授权类型或认证方式我们可以设置为：匿名anonymous或函数function。匿名方式就是不需要签名认证，匿名的用户也能访问；而函数方式，则是需要签名认证[4]，这个签名认证的算法，参数需要用到我们账户的访问密钥ak/sk[5]，ak/sk相当于我们云账户的银行卡密码，这么重要的账户信息，我们只能限定在服务端使用，前端代码里绝对不可以出现。

也就是说，我们只能在服务端使用函数安全认证方式。如果是这种方案，我们的“待办任务”架构就演进成下图这样了。



那有没有针对匿名认证方式的安全策略呢？当然有，这里我们同样需要借鉴一下微服务的鉴权设计：JSON Web Token，简称JWT[6]。JWT简单来说，就是将用户身份信息和签名信息，一起传到客户端去了。用户在请求数据时，带上这个JWT，服务端通过校验签名确定用户身份。JWT存在于客户端，JWT验证只需要通过服务端的sk和算法验证签名。同样，我画了张图，以帮助你理解。



要解决后端互调的安全性，我们用VPC或IP白名单，都很容易解决。比较难处理的是前后端的信任问题，JWT正好就提供了一种信任链的解决思路。当然，关于鉴权也有一些云服务商推出了一些更加安全易用的BaaS服务，例如AWS的IAM和Cognito[7]。

安全性是我们考虑架构设计时重要的一环，因为安全架构设计的失败，会直接导致我们资产的损失。鉴权是识别用户身份，防止用户信息泄漏和恶意攻击使用的。但根据我统计的数据，我们在日常99%的问题，都发生在新版本上线的环节。

那我们该怎么稳定持续地快速迭代，发布新版本上线呢？我们可以回想一下[第1课]，小程和小服的例子，小程最后实现NoOps后，小服则只要将代码合并到指定分支就可以发布上线了。那现实中，这点该怎么实现呢？

当我们的项目Serverless化以后，代码的质量变得尤为重要。你可以想想，Serverless化之前，你不小心上线了一个bug，影响的范围最大也就只有一个应用。但是Serverless化之后，如果是核心节点发布了严重的bug上线，那么影响的范围就是所有依赖它的线上应用了。

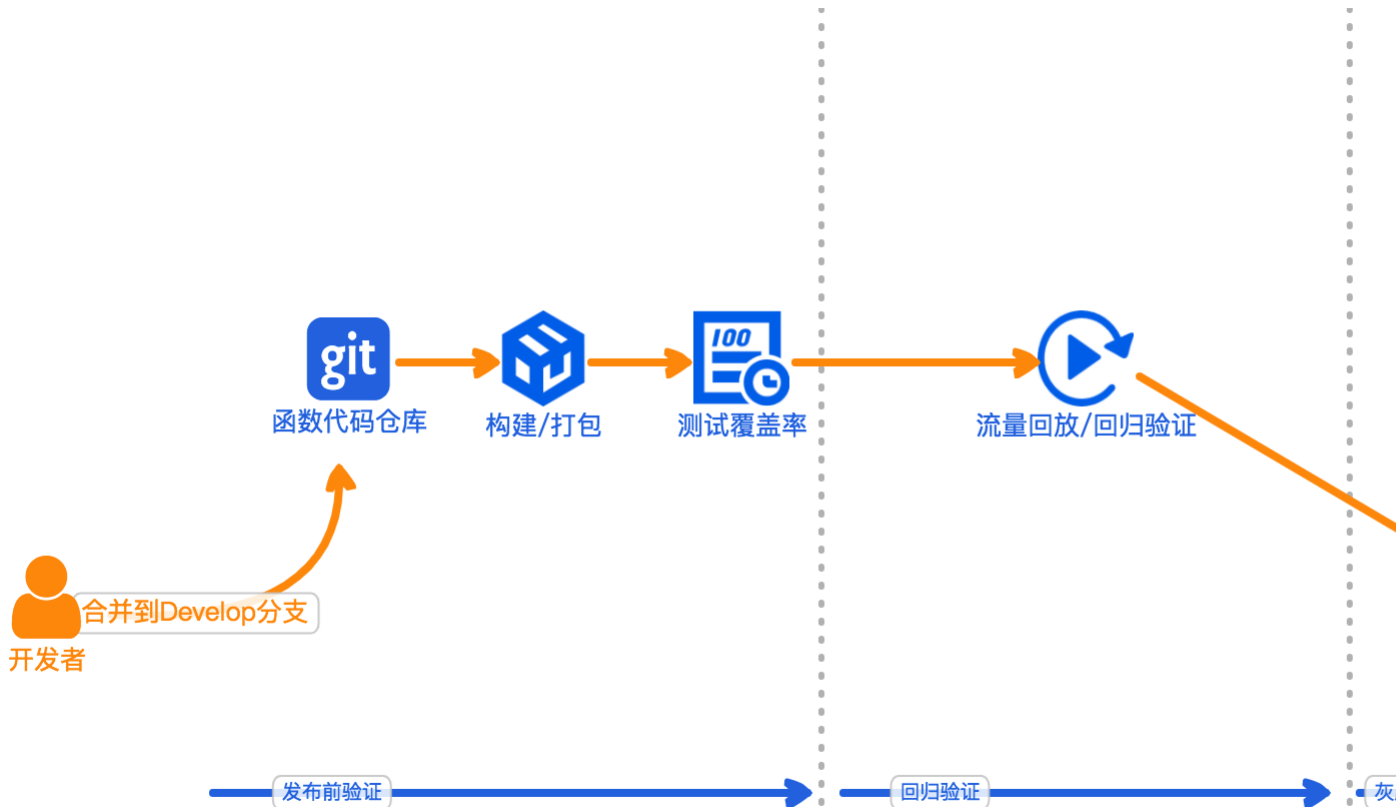
不过，你也不用太担心，微服务和FaaS都具备快速独立迭代的能力。以前我们一个应用的迭代周期通常要一周到两周。但对于Serverless化后的应用来说，每个节点借助独立运维的特性，可以随时随地地发布上线。

综上，我们知道了，微服务和FaaS都是快速迭代的，修复问题很快，但我们也不能每次都等问题出现，再去依赖这个能力呀。有没有什么办法可以提前发现问题，保证我们又快又稳？目前软件工程的最佳做法就是代码流水线的发布管道。

发布管道

发布管道的流水线主要有3个部分：

1. 代码发布前的验证，代码测试覆盖率CI/CD；
2. 模拟流量回归测试通过，发布到灰度环境；
3. 代码正式上线，灰度环境替换正式环境。流水线的每个节点产生的结果，都会作为下一个节点必要的启动参数。



我们先看看上图，我来解释下这个流程。

- 我们的代码合并到指定分支后，通常我会用Develop分支。
- Git的钩子就会触发后续的流水线，开始进入构建打包、测试流程。
- 测试节点做的事情就是跑所有测试Case，并且统计覆盖率。
- 覆盖率验证通过，代码实例用录制流量模拟验证。
- 模拟验证通过，发布代码实例到灰度环境。
- 线上根据灰度策略，将小部分流量导入灰度环境验证灰度版本。
- 在灰度窗口期，比如两个小时，灰度验证没有异常则用灰度版本替换正式版本；反之则立即丢弃这个灰度版本，止损。

这套流程，目前规模大一些的互联网公司发布流程基本都在这样跑，如果你不是很了解，可以自己尝试用我们介绍的Serverless工作流或者云服务商提供的工作流工具[8]动手搭建下。

在这套流程的基础上，很多企业为了追求更高的稳定性，还会设定环境隔离的流水线和安全卡口。比如隔离测试环境和线上环境，测试环境用来复现故障。每次代码进入发布管道，都必须先在测试环境跑通，跑通后安全卡口放行，才能进入线上环境的流水线。

总结

这节课，我们继续讲后端的BaaS化。我们再梳理一下这节课的重要知识点吧。

1. 如何拆解BaaS应用，我们学习了微服务的重要拆解思想DDD：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。另外我也介绍了另一种更适合初创企业的拆分思路：动态网络演进。
2. 拆解完之后，我们就要考虑合并。这里我们介绍了代码编排以外的另一种编排方式：事件流编排，它就是通过一个个事件顺序将我们的微服务或FaaS串联起来。
3. 为了解决拆解后，微服务之间的信任问题。我们先了解了FaaS触发器的安全方案：数字签名。还借鉴了微服务的鉴权做法JWT，将用户鉴权加密信息放在客户端，让鉴权服务变成Stateless。最后，为了让微服务又快又稳地发布版本，我们借鉴了微服务的发布管道：打造自动灰度流水线。

作业

这节课的作业就是我们JWT鉴权的“待办任务”Web应用，你来部署上线。

后端代码GitHub地址：<https://github.com/pusongyang/todolist-backend/tree/lesson06>

前端代码GitHub地址：<https://github.com/pusongyang/todolist-frontend>

演示预览地址：<http://lesson6.jike-serverless.online/list>

期待你的作业，如果今天的内容让你有所收获，也欢迎你把文章分享给身边的朋友，邀请他加入学习。

参考资料

- [1] https://en.wikipedia.org/wiki/Domain-driven_design
- [2] https://en.wikipedia.org/wiki/Dynamic_network_analysis
- [3] <https://www.aliyun.com/product/fnf>
- [4] <https://github.com/alibaba/aliyun-fc-nodejs-sdk/blob/master/lib/client.js?spm=a2c4g.11186623.2.15.16e016d7lo8NBQ#L840>
- [5] https://help.aliyun.com/document_detail/154851.html?spm=5176.2020520153.0.0.371a415dLXylZ
- [6] <https://jwt.io/>
- [7] https://docs.aws.amazon.com/zh_cn/cognito/latest/developerguide/what-is-amazon-cognito.html
- [8] <https://www.aliyun.com/product/yunxiao/devops?spm=5176.10695662.1173276.1.6c724a38akCjgo>

你好，我是秦粤。上一课中，我们学习了后端BaaS化的重要模块：微服务。现在我们知道微服务的核心理念就是先拆后合，拆解功能是为了提升我们功能的利用率。同步我们也了解了实现微服务的10要素，这10要素要真讲起来够单独开一门课的。如果你不熟悉，我向你推荐杨波老师的《微服务架构核心20讲》课程。

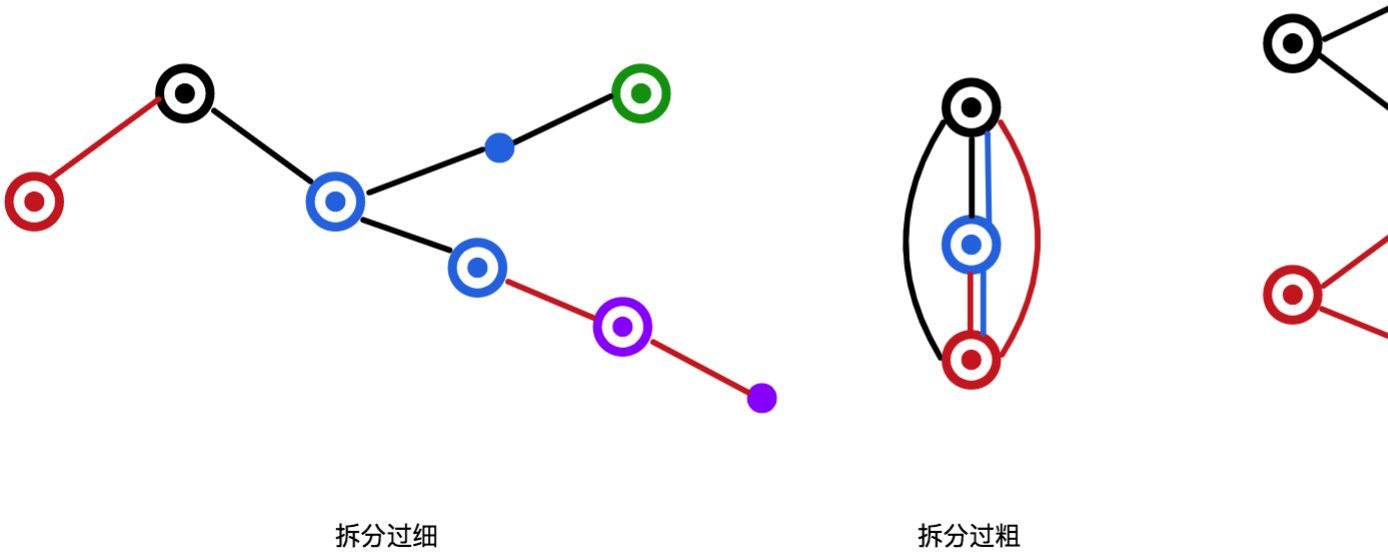
BaaS化的核心其实就是把我们的后端应用封装成RESTful API，然后对外提供服务，而为了后端应用更容易维护，我们需要将后端应用拆解成免运维的微服务。这个逻辑你要理解，这也是为什么我要花这么多篇幅给你谈微服务的关键原因。

上节课我们将“待办任务”Web服务的后端，拆解为用户微服务和待办任务微服务。但为什么要这样拆？是凭感觉，还是有具体的方法论？这里你可以停下来想想。

微服务的拆解和合并，都有一个度需要把握，因为我们在一拆一合之间，都是有成本产生的。如果我们拆解得太细，就必然会导致我们的调用链路增长。调用链路变长，首先影响的就是网络延迟，这个好理解，毕竟你路远了，可能“堵车”的地方也会变多；其次是运维成本的增加，调用链路越长，整个链条就越脆弱，因为其中一环出现问题，都会导致整个调用链条访问失败，而且我们排查问题也变得更加困难。

反过来看，如果我们拆解得太粗，调用链路倒是短了，但是这个微服务的复用性就差了，更别提因为高耦合带来的复杂且冗余的数据库表结构，让我们后续难以维护。我画了个图，你感受一下。

带环的是Stateless节点
实心的是Stateful节点



拆之，领域驱动设计

那我们要合理地拆解微服务，应该怎么拆解呢？上节课其实我有提到，目前主流的解决方案就是领域驱动设计，也叫DDD[1]。DDD是Eric Evans在其2004年的同名书中提出来的一个思想，但一直仅仅局限在Java的圈子里，直到2014年，微服务兴起后大家才发现它可以指导微服务的拆分，这才走进了大多数人的视野。用一句话简单总结，DDD就是一套方法论：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。

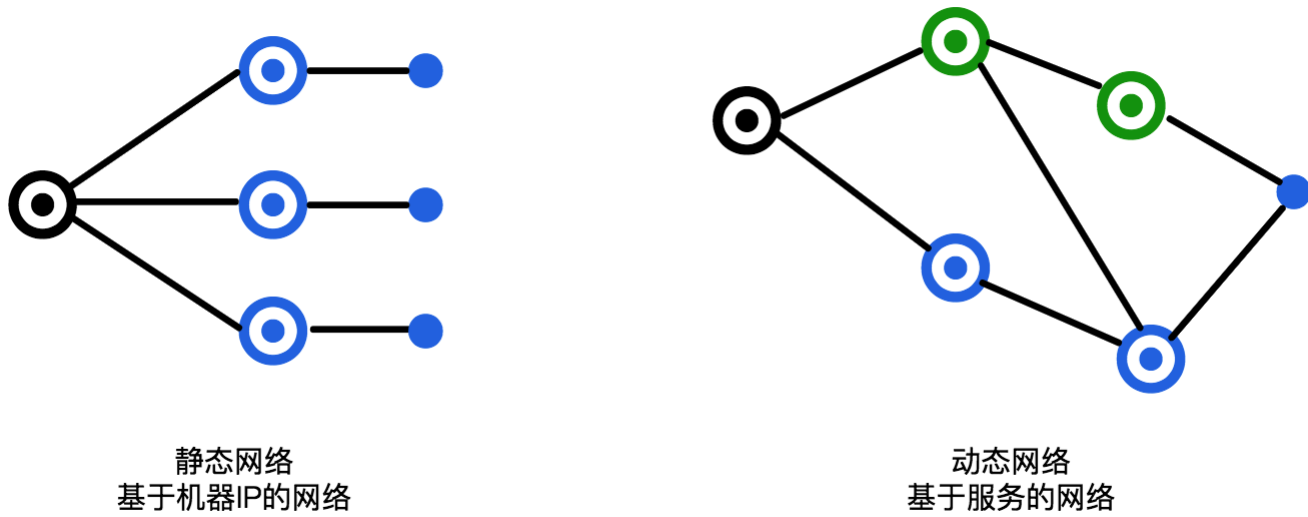
实际我自己在用微服务开发的过程发现，微服务整体应该是一个动态网络结构[2]，随着业务的发展，这个网络结构也会发生变化。DDD能帮助我们前期分析出一个较好的网络结构，但实际上，我们更应该思考的是如何整体优化动态网络：减少核心节点，保护核心节点，降低网络深度等等。

怎么理解动态网络优化呢？我们可以做个思维实验：假设我们将所有的功能都拆解成微服务，任意的微服务节点之间都可以相互调用，调用越频繁它们之间的距离就越近。那么我们考虑一下，当我们网站的访问请求流量稳定后，我们整个微服务节点组成的网络状态是怎么样的？

首先网络节点的相互制约总会让那些相互之间强依赖的、高耦合的节点，越走越近，最后聚集成一团节点。其次那些跟业务逻辑无关的节点，逐渐被边缘化，甚至消失。我们看这些聚集成团的节点，如果团里的点聚合太近了，其实是不适合拆分的，它们整体应该作成是一个微服务。等这些节点太近的团合并成一个微服务节点后，我们再看那些聚集在一起、又不太近的节点就是一个微服务了。

所以，我们在启动项目时，不用太过纠结应该如何去拆解微服务。而应该持续关注，并思考每个微服务节点的合理性。就像看待动态网络一样，持续地调整优化，去除核心节点。最终它会伴随你业务的发展阶段，达到各个阶段的稳定动态网络结构。

就像我们上节课“待办任务”Web服务一样，我们可以先简单地将我们的项目后端分为：用户微服务和待办任务微服务。当然这里我们目前的业务太简单了，用DDD去分析，也是大材小用。随着我这个项目的业务发展，我们添加的功能会越来越多。让微服务根据业务一起成长演变就可以了。这并不是说我们就放任微服务不管了，而是从整体网络的角度思考，去看我们的微服务如何演进。



合之，Streaming

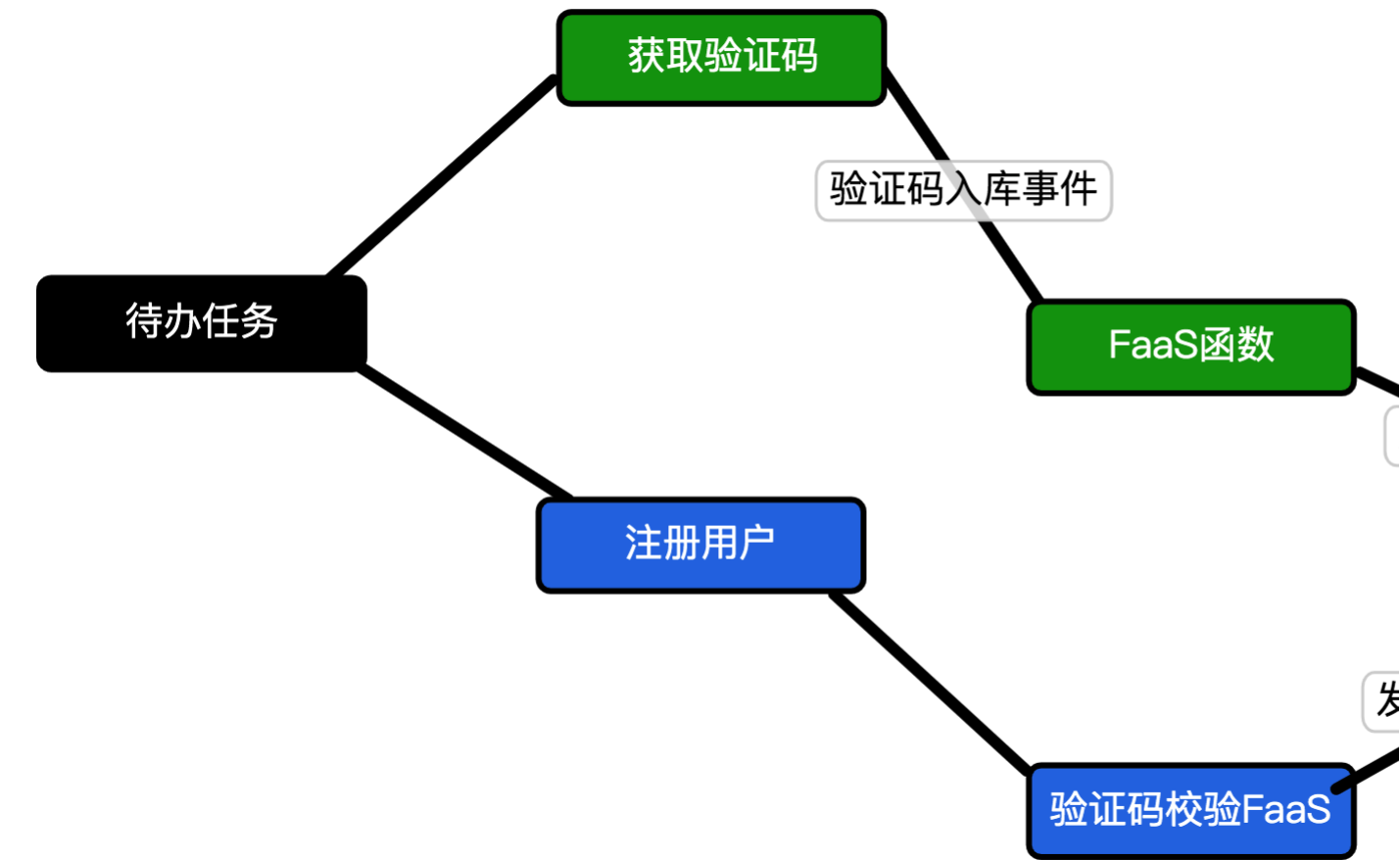
看完拆解，我们再看合并。合并呢，换个高大上的词其实就是前面课程中提到的编排。目前为止，我们整个“待办任务”Web应用架构的设计基本完成了，而且所有节点都是Stateless的了。变成Stateless节点后，其实对于前端的同学来说，一点都不陌生，比如React的单向数据流中的State也要求我们Immutable，Immutable其实就是Stateless。

我们上面已经看到了，拆解后的架构是个动态网络，那我们应该怎么合并或者编排呢？当然你像SFF那样通过传统的函数，将每个HTTP数据的请求结果通过数组或对象加工处理，再将这些结果返回也是可以的。但我在这里想向你介绍另外一种编排思路， workflow。



我们可以将用户的请求想象成我们的呼吸系统，我们的肺就是SFF，而微服务和FaaS节点就是需要氧气的各个器官。我们吸一口气，氧气进入肺部，血液循环将氧气按顺序流经我们每个器官，这就是请求链路。每个器官一接收到新鲜血液，就会吸取氧气返回二氧化碳，最终血液循环将二氧化碳带到肺部呼出，这个就是数据返回链路。我们的各个器官，就被请求链路通过新鲜血液到来的这个事件串联起来了，这个就是事件流，也就是用一个事件去串联FaaS或微服务。

现在我们用[第3课]讲的，PHP发邮件改造一下，举个例子。当用户注册时，我们完全可以将用户的信息和注册验证码存入数据库；PHP发邮件的FaaS触发器改为数据库插入新记录触发事件；用户从邮箱验证获取验证码，把验证码写到输入框后，点击验证，则是另一个HTTP触发器，触发FaaS函数校验验证码通过，修改数据库注册成功，并且返回302跳转到登录成功页面。具体流程可参考下图：



当然现在这个解决方案也有成熟对应的云BaaS服务：Serverless工作流[3]。

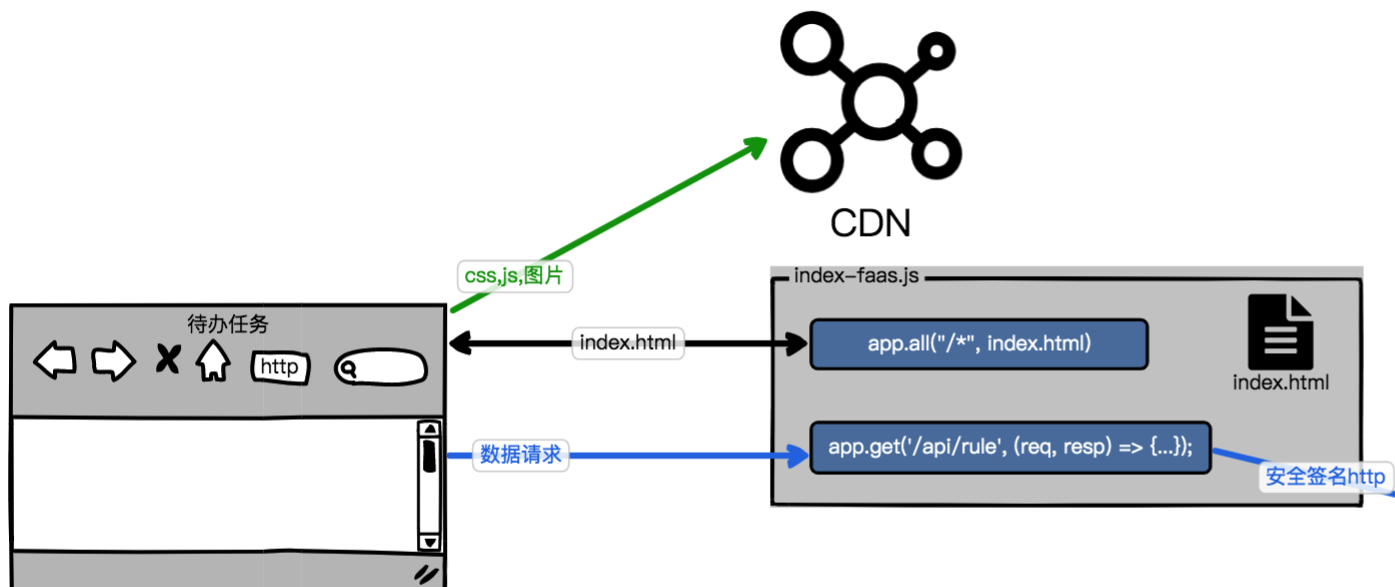
安全门神

理解了拆合的思想，我们就可以将目前“待办项目”的架构再演进一下：静态文件我们用CDN托管，前端项目只负责域名支撑和index.html，剩下的请求直接访问FaaS微服务。这时候，我估计你会问，咱们数据的安全性如何保障呢？是的，到目前为止，我们的FaaS都一直在用匿名模式访问，完全没有任何安全防护可言，也就是说目前我们FaaS服务的接口一直都在互联网上“裸奔”。

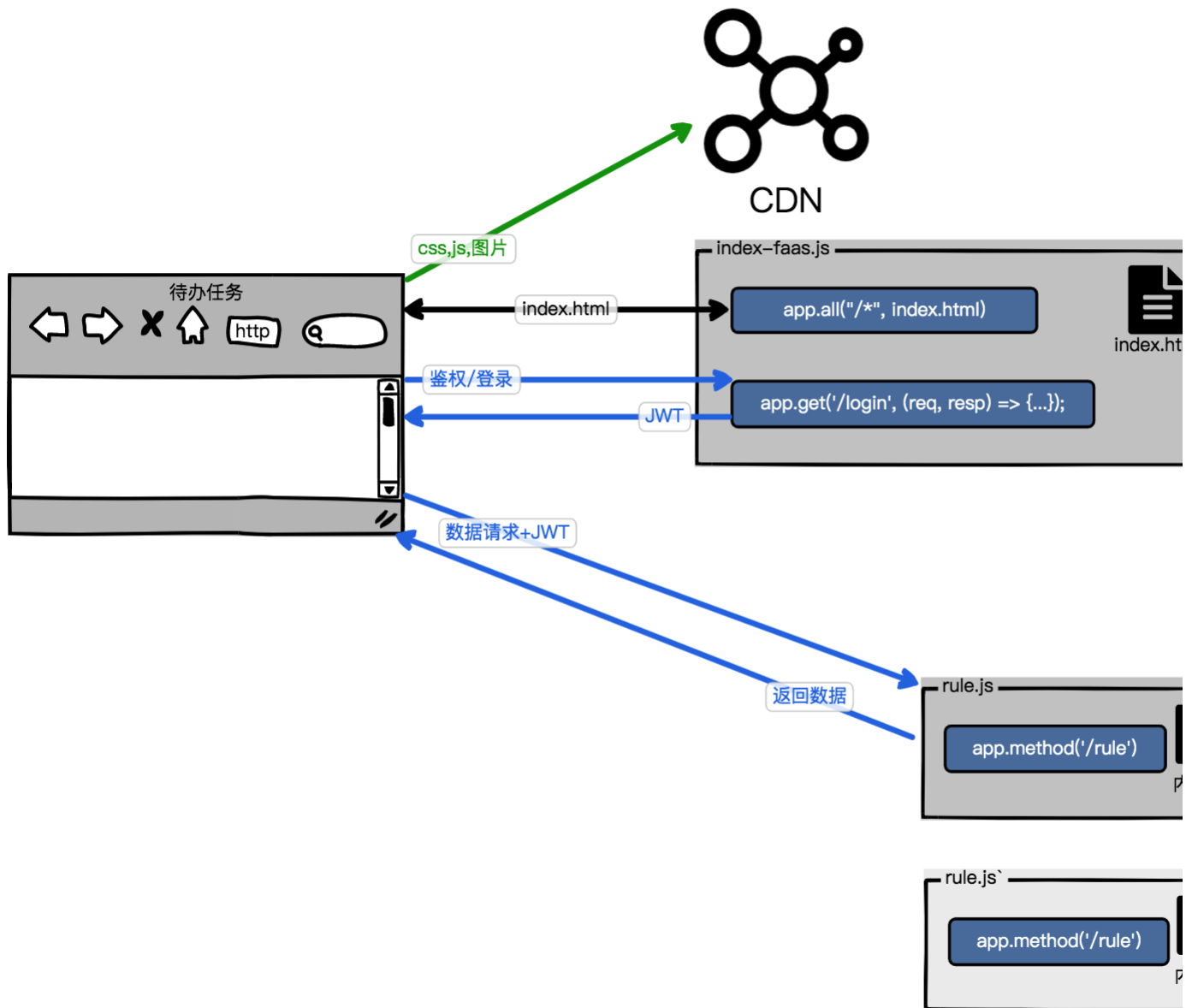
鉴权

其实，FaaS提供的安全防护通常是放在触发器上的。触发器的授权类型或认证方式我们可以设置为：匿名anonymous或函数function。匿名方式就是不需要签名认证，匿名的用户也能访问；而函数方式，则是需要签名认证[4]，这个签名认证的算法，参数需要用到我们账户的访问密钥ak/sk[5]，ak/sk相当于我们云账户的银行卡密码，这么重要的账户信息，我们只能限定在服务端使用，前端代码里绝对不可以出现。

也就是说，我们只能在服务端使用函数安全认证方式。如果是这种方案，我们的“待办任务”架构就演进成下图这样了。



那有没有针对匿名认证方式的安全策略呢？当然有，这里我们同样需要借鉴一下微服务的鉴权设计：JSON Web Token，简称JWT[6]。JWT简单来说，就是将用户身份信息和签名信息，一起传到客户端去了。用户在请求数据时，带上这个JWT，服务端通过校验签名确定用户身份。JWT存在于客户端，JWT验证只需要通过服务端的sk和算法验证签名。同样，我画了张图，以帮助你理解。



要解决后端互调的安全性，我们用VPC或IP白名单，都很容易解决。比较难处理的是前后端的信任问题，JWT正好就提供了一种信任链的解决思路。当然，关于鉴权也有一些云服务商推出了一些更加安全易用的BaaS服务，例如AWS的IAM和Cognito[7]。

安全性是我们考虑架构设计时重要的一环，因为安全架构设计的失败，会直接导致我们资产的损失。鉴权是识别用户身份，防止用户信息泄漏和恶意攻击使用的。但根据我统计的数据，我们在日常99%的问题，都发生在新版本上线的环节。

那我们该怎么稳定持续地快速迭代，发布新版本上线呢？我们可以回想一下[第1课]，小程和小服的例子，小程最后实现NoOps后，小服则只要将代码合并到指定分支就可以发布上线了。那现实中，这点该怎么实现呢？

当我们的项目Serverless化以后，代码的质量变得尤为重要。你可以想想，Serverless化之前，你不小心上线了一个bug，影响的范围最大也就只有一个应用。但是Serverless化之后，如果是核心节点发布了严重的bug上线，那么影响的范围就是所有依赖它的线上应用了。

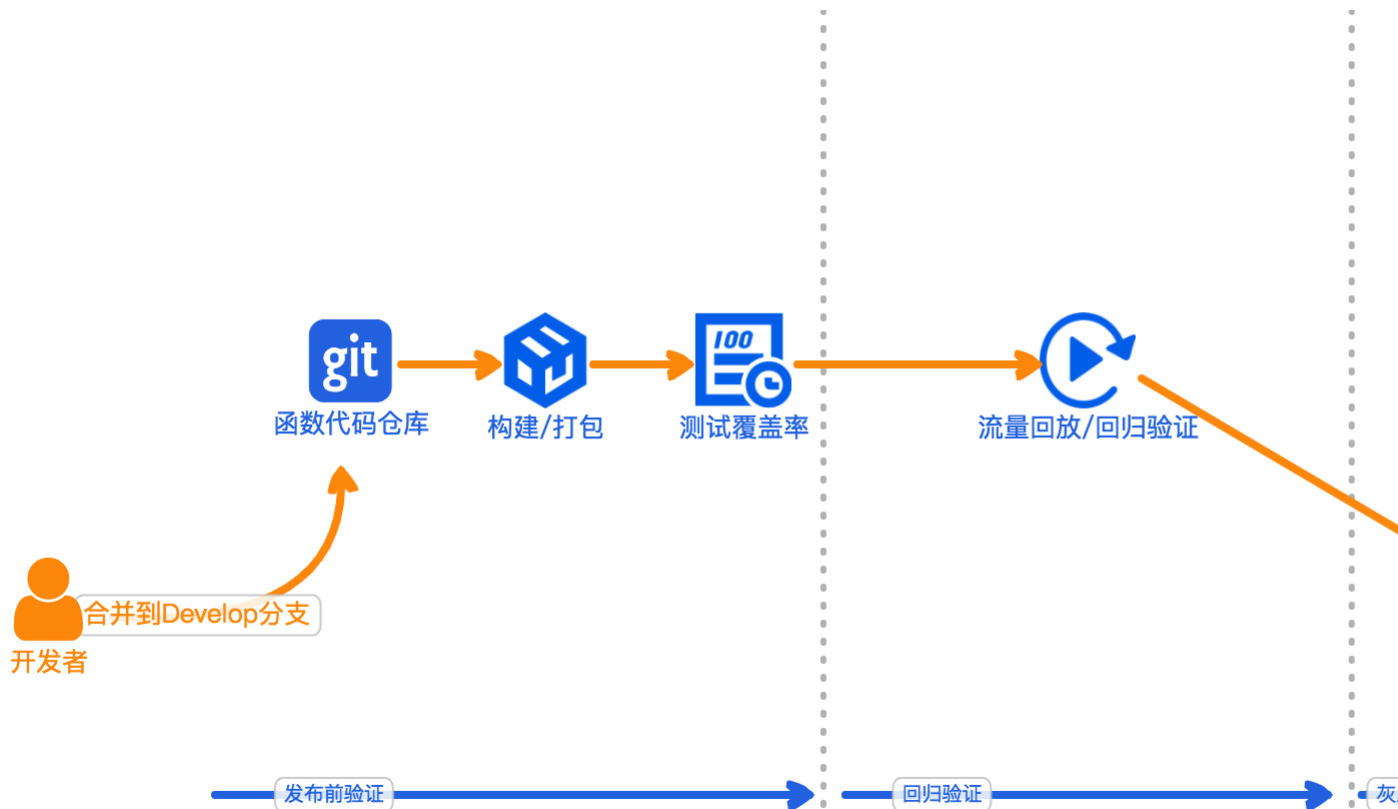
不过，你也不用太担心，微服务和FaaS都具备快速独立迭代的能力。以前我们一个应用的迭代周期通常要一周到两周。但对于Serverless化后的应用来说，每个节点借助独立运维的特性，可以随时随地地发布上线。

综上，我们知道了，微服务和FaaS都是快速迭代的，修复问题很快，但我们也不能每次都等问题出现，再去依赖这个能力呀。有没有什么办法可以提前发现问题，保证我们又快又稳？目前软件工程的最佳做法就是代码流水线的发布管道。

发布管道

发布管道的流水线主要有3个部分：

1. 代码发布前的验证，代码测试覆盖率CI/CD；
2. 模拟流量回归测试通过，发布到灰度环境；
3. 代码正式上线，灰度环境替换正式环境。流水线的每个节点产生的结果，都会作为下一个节点必要的启动参数。



我们先看看上图，我来解释下这个流程。

- 我们的代码合并到指定分支后，通常我会用Develop分支。
- Git的钩子就会触发后续的流水线，开始进入构建打包、测试流程。
- 测试节点做的事情就是跑所有测试Case，并且统计覆盖率。
- 覆盖率验证通过，代码实例用录制流量模拟验证。
- 模拟验证通过，发布代码实例到灰度环境。
- 线上根据灰度策略，将小部分流量导入灰度环境验证灰度版本。
- 在灰度窗口期，比如两个小时，灰度验证没有异常则用灰度版本替换正式版本；反之则立即丢弃这个灰度版本，止损。

这套流程，目前规模大一些的互联网公司发布流程基本都在这样跑，如果你不是很了解，可以自己尝试用我们介绍的Serverless工作流或者云服务商提供的工作流工具[8]动手搭建下。

在这套流程的基础上，很多企业为了追求更高的稳定性，还会设定环境隔离的流水线和安全卡口。比如隔离测试环境和线上环境，测试环境用来复现故障。每次代码进入发布管道，都必须先在测试环境跑通，跑通后安全卡口放行，才能进入线上环境的流水线。

总结

这节课，我们继续讲后端的BaaS化。我们再梳理一下这节课的重要知识点吧。

1. 如何拆解BaaS应用，我们学习了微服务的重要拆解思想DDD：通过对业务分层抽象，分析定义出领域模型，用领域模型驱动我们设计系统，最终将复杂的业务模型拆解为独立运维的领域模型。另外我也介绍了另一种更适合初创企业的拆分思路：动态网络演进。
2. 拆解完之后，我们就要考虑合并。这里我们介绍了代码编排以外的另一种编排方式：事件流编排，它就是通过一个个事件顺序将我们的微服务或FaaS串联起来。
3. 为了解决拆解后，微服务之间的信任问题。我们先了解了FaaS触发器的安全方案：数字签名。还借鉴了微服务的鉴权做法JWT，将用户鉴权加密信息放在客户端，让鉴权服务变成Stateless。最后，为了让微服务又快又稳地发布版本，我们借鉴了微服务的发布管道：打造自动灰度流水线。

作业

这节课的作业就是我们JWT鉴权的“待办任务”Web应用，你来部署上线。

后端代码GitHub地址：<https://github.com/pusongyang/todolist-backend/tree/lesson06>

前端代码GitHub地址：<https://github.com/pusongyang/todolist-frontend>

演示预览地址：<http://lesson6.jike-serverless.online/list>

期待你的作业，如果今天的内容让你有所收获，也欢迎你把文章分享给身边的朋友，邀请他加入学习。

参考资料

- [1] https://en.wikipedia.org/wiki/Domain-driven_design
- [2] https://en.wikipedia.org/wiki/Dynamic_network_analysis
- [3] <https://www.aliyun.com/product/fnf>
- [4] <https://github.com/alibaba/aliyun-fc-nodejs-sdk/blob/master/lib/client.js?spm=a2c4g.11186623.2.15.16e016d7lo8NBQ#L840>
- [5] https://help.aliyun.com/document_detail/154851.html?spm=5176.2020520153.0.0.371a415dLXylZ
- [6] <https://jwt.io/>
- [7] https://docs.aws.amazon.com/zh_cn/cognito/latest/developerguide/what-is-amazon-cognito.html
- [8] <https://www.aliyun.com/product/yunxiao/devops?spm=5176.10695662.1173276.1.6c724a38akCjgo>