@来源 中间件面试题（2021优化版）

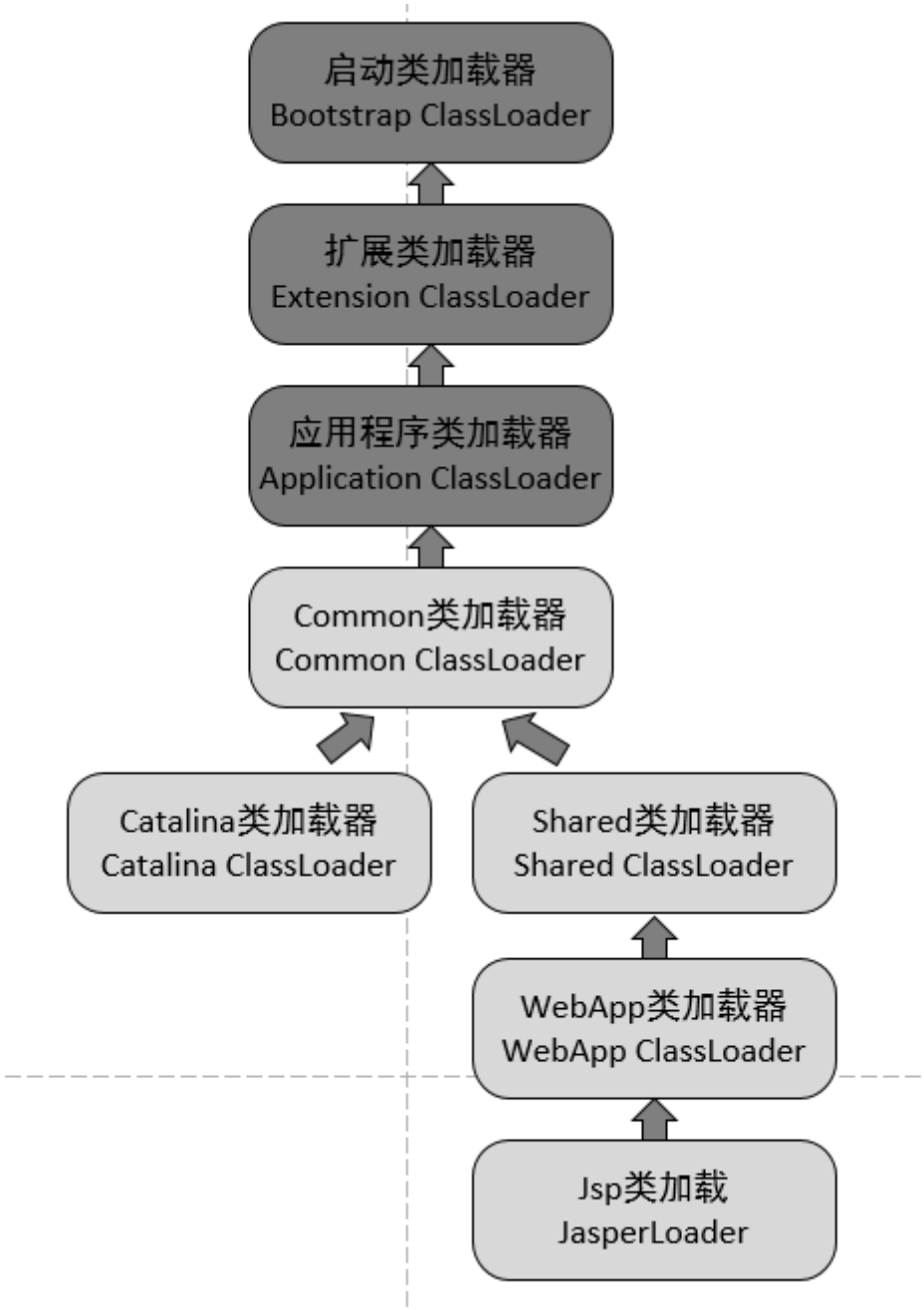
Tomcat

Tomcat是什么?

Tomcat 服务器Apache软件基金会项目中的一个核心项目，是一个免费开源的轻量级 Web 应用服务器，在中小型系统和并发访问用户不大的场合下被普遍使用，是开发和调试 JSP 程序的首选。

Tomcat类加载

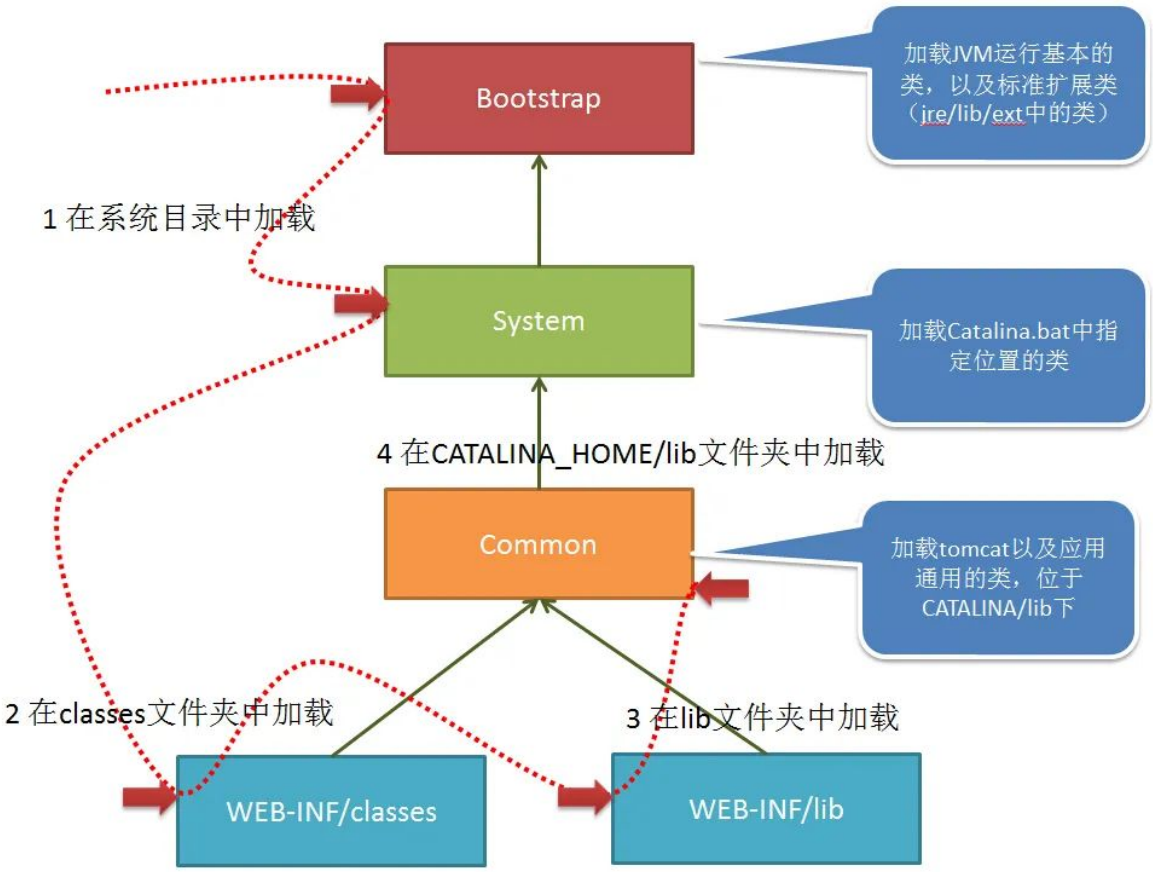
Tomcat整体的类加载图



我们在这张图中看到很多类加载器，除了Jdk自带的类加载器，我们尤其关心Tomcat自身持有的类加载器。仔细一点我们很容易发现：Catalina类加载器和Shared类加载器，他们并不是父子关系，而是兄弟关系。为啥这样设计，我们得分析一下每个类加载器的用途，才能知晓。

1. Common类加载器，负责加载**Tomcat和Web应用的通用类**

2. Catalina类加载器，负责加载**Tomcat专用的类**，而这些被加载的类在Web应用中将不可见
3. Shared类加载器，负责加载**Tomcat下所有的Web应用程序的类**，而这些被加载的类在Tomcat容器将不可见
4. WebApp类加载器，负责加载具体的某个Web应用程序的类，而这些被加载的类在Tomcat和其他的Web应用程序都将不可见
5. Jsp类加载器，每个jsp页面一个类加载器，不同的jsp页面有不同的类加载器，方便实现jsp页面的热插拔



双亲委派模型：如果收到一个类加载的请求，本身不会先加载此类，而是会先将此请求委派给父类加载器去完成，每个层次都是如此，直到启动类加载器中，只有父类都没有加载此文件，那么子类才会尝试自己去加载。

双亲委派模型的好处：保证**核心类库不被覆盖。**如果没有使用双亲委派模型，由各个类加载器自行加载的话，如果用户自己编写了一个称为java.lang.Object的类，并放在程序的ClassPath中，那系统**将会出现多个不同的Object类**， Java类型体系中最基础的行为就无法保证。应用程序也将会变得一片混乱。

那么Tomcat为什么要自定义类加载器呢?

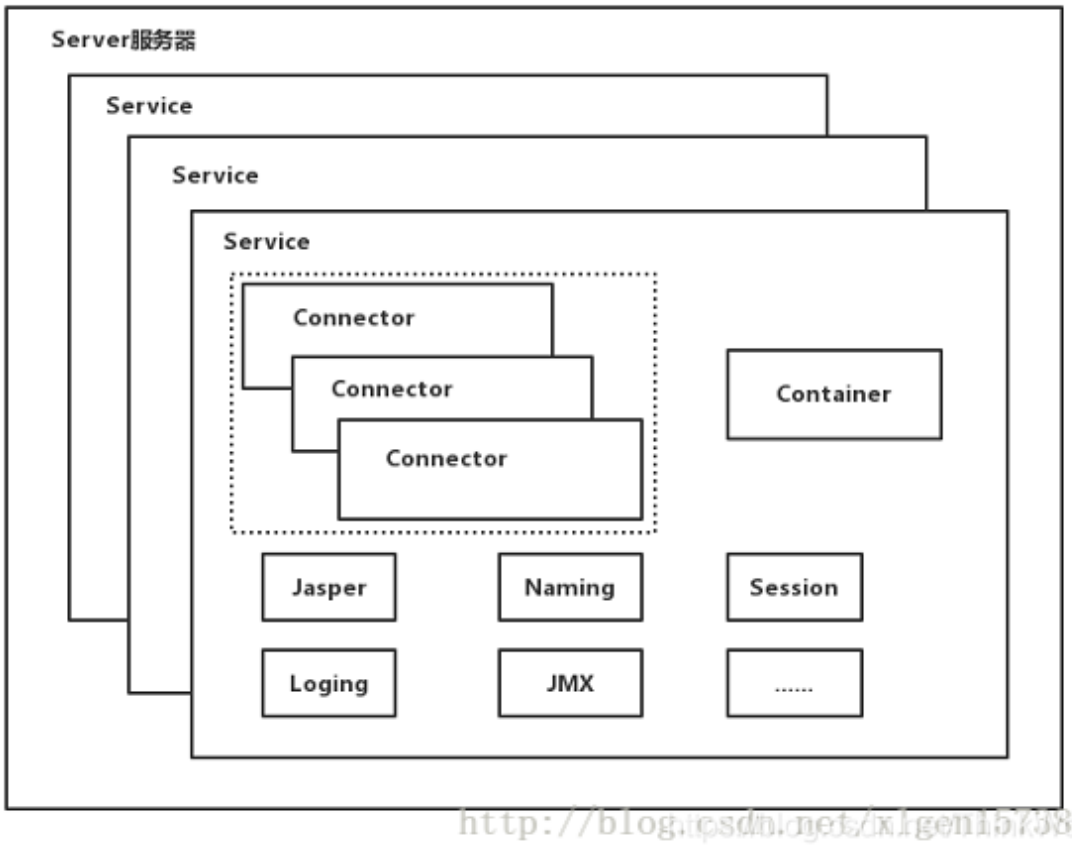
- **隔离不同应用**：部署在同一个Tomcat中的不同应用A和B，例如A用了Spring2.5。B用了Spring3.5，那么这两个应用**如果使用的是同一个类加载器，那么Web应用就会因为jar包覆盖而无法启动。**
- **隔离服务器与不同应用**：服务器需要尽可能地保证自身的安全不受部署的Web应用程序影响。一般来说，服务器所使用的类库应该与应用程序的类库互相独立。
- 性能：部署在同一服务上的两个Web应用程序所使用的Java类库可以互相共享。

Tomcat自定义了WebAppClassLoader类加载器。打破了双亲委派的机制，即如果收到类加载的请求，会尝试自己去加载，如果找不到再交给父加载器去加载，目的就是为了优先加载Web应用自己定义的类。我们知道ClassLoader默认的loadClass方法是以双亲委派的模型进行加载类的，那么Tomcat既然要打破这个规则，就要重写loadClass方法，我们可以看WebAppClassLoader类中重写的loadClass方法。

Web应用默认的类加载顺序（打破了双亲委派规则）：

1. 从本地缓存中查找是否加载过此类，如果已经加载即返回，否则继续下一步。
2. 检查 JVM 的缓存中是否已经加载，防止Web应用覆盖JRE的核心类
3. 从AppClassLoader中查找是否加载过此类，如果加载到即返回，否则继续下一步。
4. 判断是否设置了delegate属性，如果设置为true那么就按照双亲委派机制加载类
5. 默认是设置delegate是false的，那么就会先用WebAppClassLoader进行加载
6. 如果此时在WebAppClassLoader没找到类，那么就委托父类加载器(Common ClassLoader)去加载

Tomcat的基本架构是什么?



Tomcat中只有一个Server，一个Server可以有多个Service，一个Service可以有多个Connector和一个Container；

Server掌管着整个Tomcat的生死大权；

Service 是对外提供服务的；

Connector用于接受请求并将请求封装成Request和Response来具体处理；

Container用于封装和管理Servlet，以及具体处理request请求

Tomcat请求的处理流程

给定一个客户端访问的URL：http://localhost:8080/TestWeb/index.jsp.详细说一下该请求的处理流程.

1. 首先是请求发送给本机8080,被在那里监听HTTP/Connector获得.
2. Connector将该请求发给它本身所在的 Service所在的Engine来处理,并等待Engine来回应.
3. Engine匹配对应的Host和Context,并将请求交给对应的Servlet
4. 构造HttpServletRequest对象和HttpServletResponse对象作为参数,调用Servlet的doGet()方法和doPost()方法.
5. Context把执行完后的HttpServletResponse对象返回给Host,再返回给Engine,Connector
6. Connector把HttpServletResponse对象返回给浏览器的Browser.

Dubbo

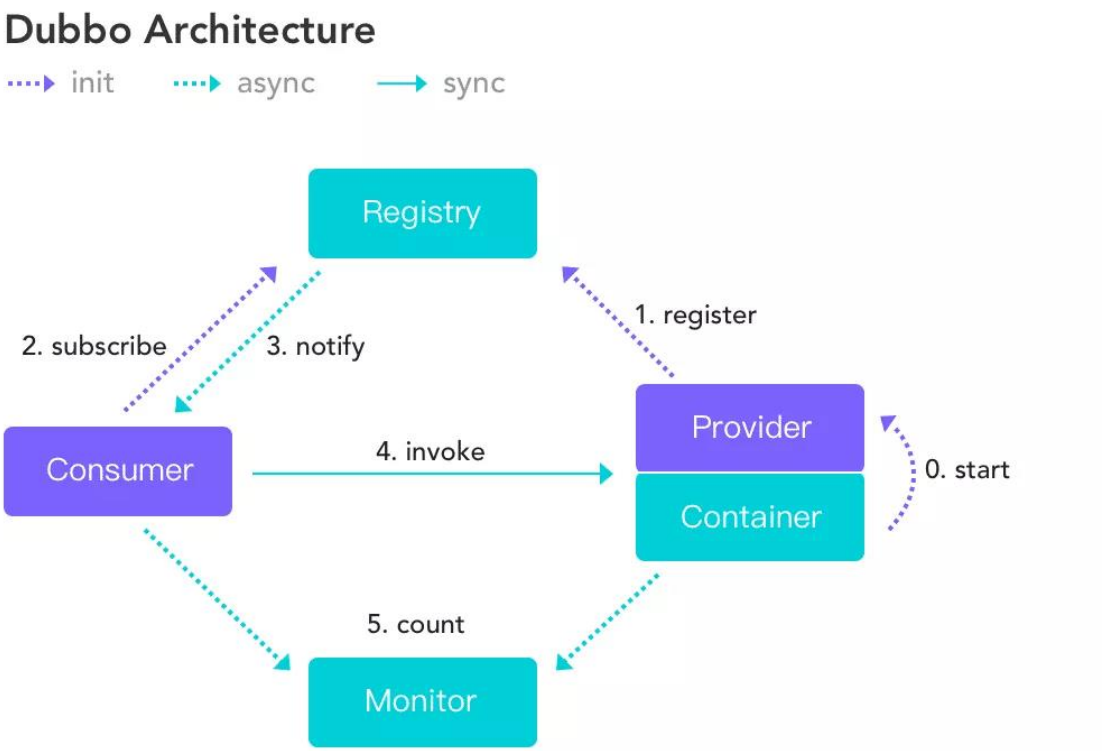
Dubbo 是什么?

Dubbo 是一款高性能、轻量级的开源 RPC 框架，提供服务自动注册、自动发现等高效服务治理方案，可以和 Spring 框架无缝集成。

Dubbo 的使用场景有哪些?

- 透明化的远程方法调用：就像调用本地方法一样调用远程方法，只需简单配置，没有任何API侵入。
- 软负载均衡及容错机制：可在内网替代 F5 等硬件负载均衡器，降低成本，减少单点。
- 服务自动注册与发现：不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的IP地址，并且能够平滑添加或删除服务提供者。

@$Dubbo 服务器注册与发现的流程?



服务容器Container负责启动，加载，运行服务提供者。

服务提供者Provider在启动时，向注册中心注册自己提供的服务。

服务消费者Consumer在启动时，向注册中心订阅自己所需的服务。

注册中心Registry返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

服务消费者Consumer，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

服务消费者Consumer和提供者Provider，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到**监控中心Monitor**。

Dubbo 和 Spring Cloud 有什么关系？Dubbo 和 Spring Cloud 有什么哪些区别?

Dubbo 是 SOA 时代的产物，它的关注点主要在于服务的调用，流量分发、流量监控和熔断。

而 Spring Cloud 诞生于微服务架构时代，考虑的是微服务治理的方方面面，另外由于依托了 Spring、Spring Boot 的优势，两个框架在目标就不一致，**Dubbo 定位服务治理、Spring Cloud 是打造一个分布式的生态。**

Dubbo 底层是使用 Netty 这样的 NIO 框架，是基于 TCP 协议传输的，配合以 Hession 序列化完成 RPC 通信。

Spring Cloud 是基于 Http 协议 Restful 接口调用远程过程的通信，相对来说 Http 请求会有更大的报文，占的带宽也会更多。但是 Restful 相比 RPC 更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的微服务环境下，显得更为合适，至于注重通信速度还是方便灵活性，具体情况具体分析。

@$Dubbo集群提供了哪些负载均衡策略?

Dubbo内置了4种负载均衡策略

1. RandomLoadBalance：随机负载均衡。随机的选择一个。是Dubbo的**默认**负载均衡策略。
2. RoundRobinLoadBalance：轮询负载均衡。轮询选择一个。
3. LeastActiveLoadBalance：最少活跃调用数，相同活跃数的随机。每收到一个请求，活跃数加1，完成请求后则将活跃数减1。活跃调用数越小，表明该服务提供者效率越高，单位时间内可处理更多的请求。
4. ConsistentHashLoadBalance：一致性哈希负载均衡。相同参数的请求总是落在同一台机器上。

@$Dubbo的集群容错方案有哪些?

- Failover Cluster：失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长延迟。
- Failfast Cluster：快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。
- Failsafe Cluster：失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。
- Failback Cluster：失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。
- Forking Cluster：并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数。
- Broadcast Cluster：广播调用所有提供者，逐个调用，任意一台报错则报错 。通常用于通知所有提供者更新缓存或日志等本地资源信息。

默认的容错方案是 Failover Cluster。

消息中间件&RabbitMQ

什么是RabbitMQ?

RabbitMQ是一款开源的，Erlang编写的，基于AMQP协议的消息中间件

为什么使用MQ？MQ的优点

- 异步处理 - 相比于传统的串行、并行方式，提高了系统的吞吐量。
- 应用解耦 - 系统间通过消息通信，不用关心其他系统的处理。
- 流量削锋 - 可以通过消息队列长度控制请求量，可以缓解短时间内的高并发请求。
- 消息通讯 - 消息队列一般都内置了高效的通信机制，因此也可以用在纯消息通讯上。比如实现点对点消息队列，或者聊天室等。
- 日志处理 - 解决大量日志传输。

@$你们公司生产环境用的是什么消息中间件?

这个首先你可以说下你们公司选用的是什么消息中间件，比如用的是RabbitMQ，然后可以初步给一些你**对不同MQ中间件技术的选型分析。**

举个例子：比如说ActiveMQ是老牌的消息中间件，国内很多公司过去运用的还是非常广泛的，功能很强大。

但是问题在于ActiveMQ没法支撑互联网公司的高并发、高负载以及高吞吐的复杂场景，现在在国内互联网公司落地较少。而且使用较多的是一些传统企业，**用ActiveMQ做异步调用和系统解耦。**

然后你可以说说RabbitMQ，他的好处在于可以支撑高并发、高吞吐量、性能很高，同时有非常完善便捷的后台管理界面可以使用。

另外，他还支持集群化、高可用部署架构、消息高可靠支持，功能较为完善。

而且经过调研，国内各大互联网公司落地RabbitMQ集群支撑自身业务的case较多，国内各种中小型互联网公司使用RabbitMQ的实践也比较多。

除此之外，RabbitMQ的开源社区很活跃，较高频率的版本迭代，来修复发现的bug以及进行各种优化，因此综合考虑过后，公司采取了RabbitMQ。

但是RabbitMQ也有一点缺陷，就是他自身是基于erlang语言开发的，所以导致较为难以分析里面的源码，也较难进行深层次的源码定制和改造，需要较为扎实的erlang语言功底。

然后可以聊聊RocketMQ，是阿里开源的，经过阿里生产环境的超高并发、高吞吐的考验，性能卓越，同时还支持分布式事务等特殊场景。

而且RocketMQ是基于Java语言开发的，适合深入阅读源码，有需要可以站在源码层面解决线上问题，包括源码的二次开发和改造。

另外就是Kafka。Kafka提供的消息中间件的功能明显较少一些，相对上述几款MQ中间件要少很多。

但是Kafka的优势在于专为超高吞吐量的实时日志采集、实时数据同步、实时数据计算等场景。

因此Kafka在大数据领域中配合实时计算技术（比如Spark Streaming、Storm、Flink）使用的较多。但是在传统的MQ中间件使用场景中较少采用。

ActiveMQ、RabbitMQ、RocketMQ、Kafka有什么优缺点?

	ActiveMQ	RabbitMQ	RocketMQ	Kafka	ZeroMQ
单机吞吐量	比RabbitMQ低	2.6w/s（消息做持久化）	11.6w/s	17.3w/s	29w/s
开发语言	Java	Erlang	Java	Scala/Java	C
主要维护者	Apache	Mozilla/Spring	Alibaba	Apache	iMatix，创始人已去世
成熟度	成熟	成熟	开源版本不够成熟	比较成熟	只有C、PHP等版本成熟
订阅形式	点对点(p2p)、广播（发布-订阅）	提供了4种：direct, topic ,Headers和fanout。fanout就是广播模式	基于topic/messageTag以及按照消息类型、属性进行正则匹配的发布订阅模式	基于topic以及按照topic进行正则匹配的发布订阅模式	点对点(p2p)
持久化	支持少量堆积	支持少量堆积	支持大量堆积	支持大量堆积	不支持
顺序消息	不支持	不支持	支持	支持	不支持
性能稳定性	好	好	一般	较差	很好
集群方式	支持简单集群模式，比如'主-备'，对高级集群模式支持不好。	支持简单集群，'复制'模式，对高级集群模式支持不好。	常用 多对'Master-Slave' 模式，开源版本需手动切换Slave变成Master	天然的'Leader-Slave'无状态集群，每台服务器既是Master也是Slave	不支持

	ActiveMQ	RabbitMQ	RocketMQ	Kafka	ZeroMQ
管理界面	一般	较好	一般	无	无

综上，各种对比之后，有如下建议：

一般的业务系统要引入 MQ，最早大家都用 ActiveMQ，但是现在确实大家用的不多了，没经过大规模吞吐量场景的验证，社区也不是很活跃，所以大家还是算了吧，我个人不推荐用这个了；

后来大家开始用 RabbitMQ，但是确实 erlang 语言阻止了大量的 Java 工程师去深入研究和掌控它，对公司而言，几乎处于不可控的状态，但是确实人家是开源的，比较稳定的支持，活跃度也高；

不过现在确实越来越多的公司会去用 RocketMQ，确实很不错，毕竟是阿里出品，但社区可能有突然黄掉的风险（目前 RocketMQ 已捐给 Apache，但 GitHub 上的活跃度其实不算高）对自己公司技术实力有绝对自信的，推荐用 RocketMQ，否则回去老老实实用 RabbitMQ 吧，人家有活跃的开源社区，绝对不会黄。

所以**中小型公司**，技术实力较为一般，技术挑战不是特别高，用 **RabbitMQ** 是不错的选择；**大型公司**，基础架构研发实力较强，用 **RocketMQ** 是很好的选择。

如果是**大数据领域**的实时计算、日志采集等场景，用 **Kafka** 是业内标准的，绝对没问题，社区活跃度很高，绝对不会黄，何况几乎是全世界这个领域的事实性规范。

@$MQ 有哪些常见问题？如何解决这些问题？

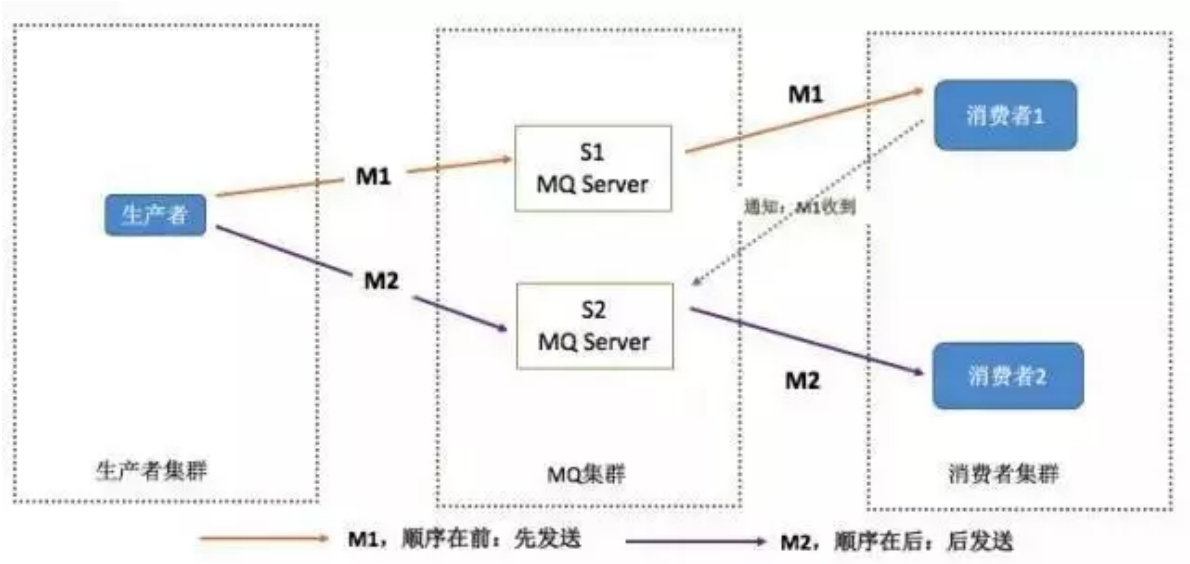
MQ 的常见问题有：

1. 消息的顺序问题
2. 消息的重复问题

消息的顺序问题

消息有序指的是可以按照消息的发送顺序来消费。

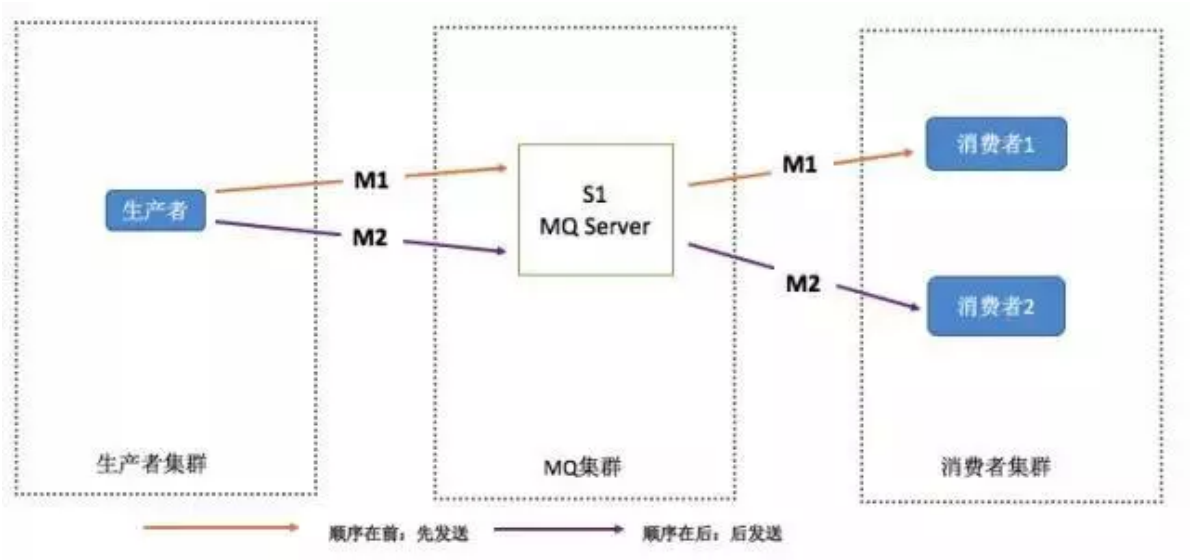
假如生产者产生了 2 条消息：M1、M2，假定 M1 发送到 S1，M2 发送到 S2，如果要保证 M1 先于 M2 被消费，怎么做？



解决方案：

保证生产者 - MQServer - 消费者是一对一对一的关系

RabbitMQ：拆分多个 queue，每个 queue 一个 consumer，就是多一些 queue 而已，确实是麻烦点；或者就一个 queue 但是对应一个 consumer，然后这个 consumer 内部用内存队列做排队，然后分发给底层不同的 worker 来处理。



缺陷：

- 并行度就会成为消息系统的瓶颈（吞吐量不够）
- 更多的异常处理，比如：只要消费端出现问题，就会导致整个处理流程阻塞，我们不得不花费更多的精力来解决阻塞的问题。通过合理的设计或者将问题分解来规避。
- 不关注顺序的应用实际大量存在
- 队列无序并不意味着消息无序，所以从业务层面来保证消息的顺序而不仅仅是依赖于消息系统，是一种更合理的方式。

其他解决方案

方案一：消费端增加消息记录表，暂存不满足业务条件的消息，并采用定时器进行补偿处理，补偿超次进行预警；（该方案对技术营运友好，目前DMS正在使用，同样该方案可以用来解决重复消费问题）

方案二：消费端对不满足业务条件的消息不进行确认，多次消费失败进入死信队列，监听死信队列进行补偿，补偿超次或失败进行预警；

方案三：采用RocketMQ顺序消费机制；（不建议使用，会降低系统吞吐量）

消息的重复问题

造成消息重复的根本原因是：网络不可达。

所以解决这个问题的办法就是绕过这个问题。那么问题就变成了：如果消费端收到两条一样的消息，应该怎样处理?

消费端处理消息的业务逻辑需要保持幂等性。只要保持幂等性，不管来多少条重复消息，最后处理的结果都一样。保证每条消息都有唯一编号和添加一张日志表来记录已经处理成功的消息的 ID，如果新到的消息 ID 已经在日志表中，那么就不再处理这条消息。

@$消息积压怎么处理

消息积压的原因

消息积压的直接原因，一定是系统中**某个部分出现了性能问题，来不及处理上游发送的消息**，才会导致消息积压。

如果日常系统正常运转的时候，没有积压或者只有少量积压很快就消费掉了，但是**某一个时刻**，突然就开始积压消息并且积压持续上涨。这种情况下需要你在短时间内找到消息积压的原因，迅速解决问题才不至于影响业务。

消息积压的处理

排查消息积压原因的方法：能导致积压突然增加，最粗粒度的原因，只有两种：要么是**发送变快**了，要么是**消费变慢**了。

- 大部分消息队列都内置了**监控**的功能，只要通过监控数据，很容易确定是哪种原因。如果是单位时间发送的消息增多，比如说是赶上大促或者抢购，短时间内不太可能优化消费端的代码来提升消费性能，唯一的方法是通过**扩容消费端的实例数**来提升总体的消费能力。如果短时间内没有足够的服务器资源进行扩容，没办法的办法是，将**系统降级**，通过**关闭一些不重要的业务**，**减少发送方发送的数据量**，**最低限度让系统还能正常运转，服务一些重要业务。**
- 还有一种不太常见的情况，你通过监控发现，无论是发送消息的速度还是消费消息的速度和原来都没什么变化，这时候你需要检查一下你的消费端，是不是**消费失败导致的一条消息反复消费**这种情况比较多，这种情况也会拖慢整个系统的消费速度。
- **如果监控到消费变慢**了，你需要检查你的消费实例，分析一下是什么原因导致消费变慢。优先**检查一下日志是否有大量的消费错误**，如果没有错误的话，可以通过打印堆栈信息，看一下你的消费线程是不是卡在什么地方不动了，比如触发了死锁或者卡在等待某些资源上了。

@$如何保证RabbitMQ消息的可靠传输？消息丢失怎么办？

消息不可靠的情况可能是消息丢失，劫持等原因；

丢失又分为：生产者丢失消息、消息列表丢失消息、消费者丢失消息；

生产者丢失消息：**从生产者弄丢数据这个角度来看，RabbitMQ提供transaction和confirm模式来确保生产者不丢消息**；

transaction机制就是说：发送消息前，开启事务（channel.txSelect()），然后发送消息，如果发送过程中出现什么异常，事务就会回滚（channel.txRollback()），如果发送成功则提交事务（channel.txCommit()）。然而，这种方式有个缺点：吞吐量下降；

confirm模式用的居多：一旦channel进入confirm模式，所有在该信道上发布的消息都将会被指派一个唯一的ID（从1开始），一旦消息被投递到所有匹配的队列之后，rabbitMQ就会发送一个ACK给生产者（包含消息的唯一ID），这就使得生产者知道消息已经正确到达目的队列了；如果rabbitMQ没能处理该消息，则会发送一个Nack消息给你，生产者可以进行重试操作。

消息队列丢数据：消息持久化。

处理消息队列丢数据的情况，一般是开启持久化磁盘的配置。

这个持久化配置可以和confirm机制配合使用，你可以在消息持久化磁盘后，再给生产者发送一个Ack信号。

这样，如果消息持久化磁盘之前，rabbitMQ阵亡了，那么生产者收不到Ack信号，生产者会自动重发。

那么如何持久化呢？

这里顺便说一下吧，其实也很容易，就下面两步

1. 将queue的持久化标识durable设置为true，则代表是一个持久的队列
2. 发送消息的时候将deliveryMode=2

这样设置以后，即使rabbitMQ挂了，重启后也能恢复数据

消费者丢失消息：消费者丢数据一般是因为采用了自动确认消息模式，改为手动确认消息即可！

消费者在收到消息之后，处理消息之前，会自动回复RabbitMQ已收到消息；

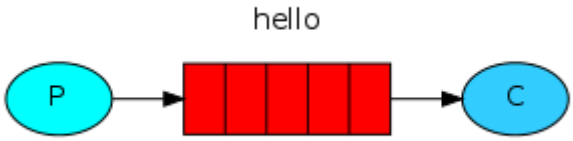
如果这时处理消息失败，就会丢失该消息；

解决方案：处理消息成功后，手动回复确认消息。

@$RabbitMQ 常见工作模式和应用场景

一、简单模式

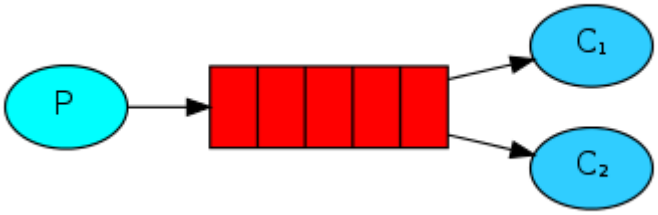
原理：一个生产者，一个消费者。生产者将消息发送到队列，消费者监听消息队列，如果队列中有消息，就进行消费，消费后消息从队列中删除



场景：聊天；有一个oa系统，用户通过接收手机验证码进行注册，页面上点击获取验证码后，将验证码放到消息队列，然后短信服务从队列中获取到验证码，并发送给用户。

二、工作模式

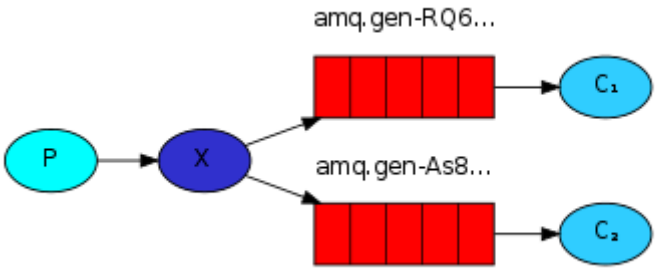
原理：一个生产者，多个消费者，一条消息只能被一个消费者消费。生产者将消息发送到消息队列，多个消费者同时监听一个队列，谁先抢到消息谁负责消费。这样就形成了资源竞争，谁的资源空闲大，争抢到的可能性就大。



场景：红包；有一个电商平台，有两个订单服务，用户下单的时候，任意一个订单服务消费用户的下单请求生成订单即可。不用两个订单服务同时消费用户的下单请求。

三、发布订阅模式

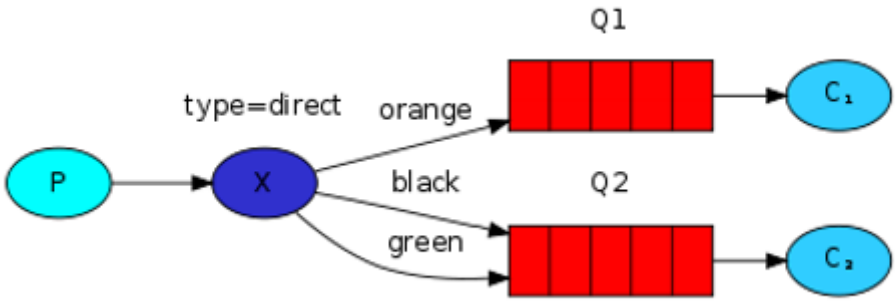
原理：一个生产者，多个消费者，每个消费者都可以收到相同的消息。生产者将消息发送到交换机，交换机类型是fanout，不同的队列注册到交换机上，不同的消费者监听不同的队列，所有消费者都会收到消息。



场景：邮件群发，群聊天，广播(广告)；有一个商城，我们新添加一个商品后，可能同时需要去更新缓存和数据库。

四、路由模式

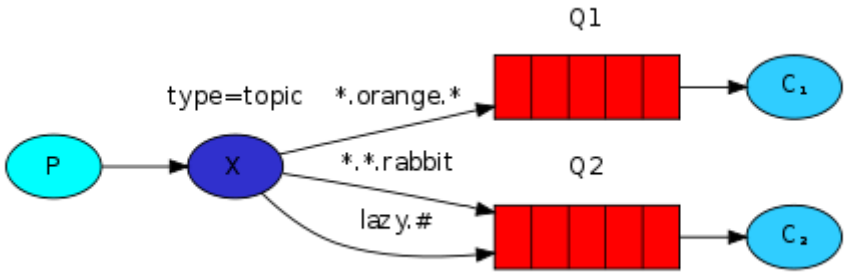
原理：生产者将消息发送给交换机，消息携带具体的routingkey。交换机类型是direct，交换机匹配与之绑定的队列的routingkey，分发到不同的队列上。



场景：还是一样，有一个商城，新添加了一个商品，实时性不是很高，只需要添加到数据库即可，不用刷新缓存。

五、主题模式

原理：路由模式的一种，交换机类型是topic，路由功能添加了模糊匹配。星号（*）代表1个单词，#号（#）代表一个或多个单词。



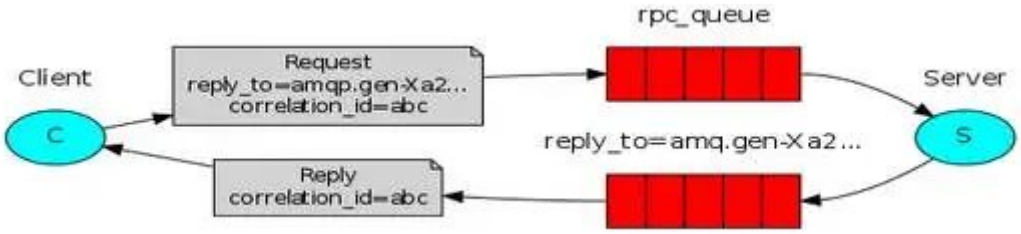
场景：还是一样，有一个商城，新添加了一个商品，实时性不是很高，只需要添加到数据库即可，数据库包含了主数据库mysql1和从数据库mysql2的内容，不用刷新缓存。

六、RPC

1、首先客户端发送一个reply_to和correntionid的请求，发布到RPC队列中；

2、服务器端处理这个请求，并把处理结果发布到一个回调Queue,此Queue的名称应当与reply_to的名称一致

3、客户端从回调Queue中得到先前correntionid设定的值的处理结果。如果碰到和先前不一样的correntionid的值，将会忽略而不是抛出异常。



@$如何保证高可用的？RabbitMQ 的集群

RabbitMQ 是比较有代表性的，因为是基于主从（非分布式）做高可用性的，我们就以 RabbitMQ 为例子讲解第一种 MQ 的高可用性怎么实现。RabbitMQ 有三种模式：单机模式、普通集群模式、镜像集群模式。

单机模式，就是 Demo 级别的，一般就是你本地启动了玩玩儿的?，没人生产用单机模式

普通集群模式，意思就是在多台机器上启动多个 RabbitMQ 实例，每个机器启动一个。你创建的 queue，只会放在一个 RabbitMQ 实例上，但是每个实例都同步 queue 的元数据（元数据可以认为是 queue 的一些配置信息，通过元数据，可以找到 queue 所在实例）。你消费的时候，实际上如果连接到了另外一个实例，那么那个实例会从 queue 所在实例上拉取数据过来。这方案主要是提高吞吐量的，就是说让集群中多个节点来服务某个 queue 的读写操作。

镜像集群模式：这种模式，才是所谓的 RabbitMQ 的高可用模式。跟普通集群模式不一样的是，在镜像集群模式下，你创建的 queue，无论元数据还是 queue 里的消息都会存在于多个实例上，就是说，每个 RabbitMQ 节点都有这个 queue 的一个完整镜像，包含 queue 的全部数据的意思。然后每次你写消息到 queue 的时候，都会自动把消息同步到多个实例的 queue 上。RabbitMQ 有很好的管理控制台，就是在后台新增一个策略，这个策略是镜像集群模式的策略，指定的时候是可以要求数据同步到所有节点的，也可以要求同步到指定数量的节点，再次创建 queue 的时候，应用这个策略，就会自动将数据同步到其他的节点上去了。这样的话，好处在于，你任何一个机器宕机了，没事儿，其它机器（节点）还包含了这个 queue 的完整数据，别的 consumer 都可以到其它节点上去消费数据。坏处在于，第一，这个性能开销也太大了吧，消息需要同步到所有机器上，导致网络带宽压力和消耗很重！RabbitMQ 一个 queue 的数据都是放在一个节点里的，镜像集群下，也是每个节点都放这个 queue 的完整数据。

权限管理

权限管理，一般指根据系统设置的安全规则或者安全策略，用户可以访问而且只能访问自己被授权的资源，不多不少。

认证授权

权限管理包括**身份认证**和**授权**两部分，简称**认证授权**。对于需要访问控制的资源用户首先经过身份认证，认证通过后用户具有该资源的访问权限方可访问。

身份认证

判断一个用户是否为合法用户的处理过程。最常用的简单身份认证方式是系统通过核对用户输入的用户名和密码，看其是否与系统中存储的该用户的用户名和密码一致，来判断用户身份是否正确。对于采用指纹等系统，则出示指纹；对于硬件Key等刷卡系统，则需要刷卡。

认证关键对象

- Subject：主体

  访问系统的用户，主体可以是用户、程序等，进行认证的都称为主体；

- Principal：身份信息

  是主体（subject）进行身份认证的标识，标识必须具有唯一性，如用户名、手机号、邮箱地址等，一个主体可以有多个身份，但是必须有一个主身份（Primary Principal）。

- credential：凭证信息

  是只有主体自己知道的安全信息，如密码、证书等。

授权

授权，即访问控制，控制谁能访问哪些资源。主体进行身份认证后需要分配权限方可访问系统的资源，对于某些资源没有权限是无法访问的。

授权关键对象

授权可简单理解为 **who** 对 **what(which)** 进行 **How** 操作：

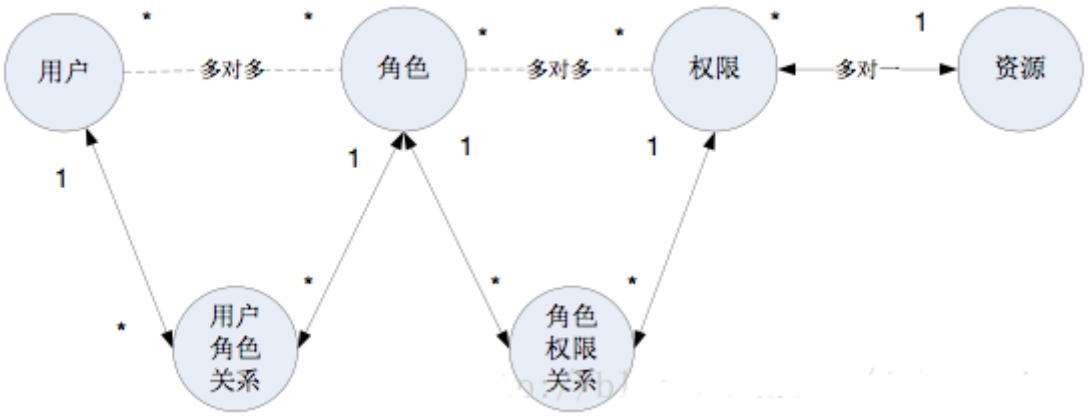
- Who，即主体（Subject），主体需要访问系统中的资源。
- What，即资源（Resource），如系统菜单、页面、按钮、类方法、系统商品信息等。资源包括资源类型和资源实例，比如商品信息为资源类型，类型为t01的商品为资源实例，编号为001的商品信息也属于资源实例。
- How，权限/许可（Permission），规定了主体对资源的操作许可，权限离开资源是没有意义，如用户查询权限、用户添加权限、某个类方法的调用权限、编号为001用户的修改权限等，通过权限可知主体对哪些资源都有哪些操作许可。

  权限分为粗颗粒和细颗粒，粗颗粒权限是指对资源类型的权限，细颗粒权限是对资源实例的权限。

权限模型

主体、资源、权限的数据模型表示。

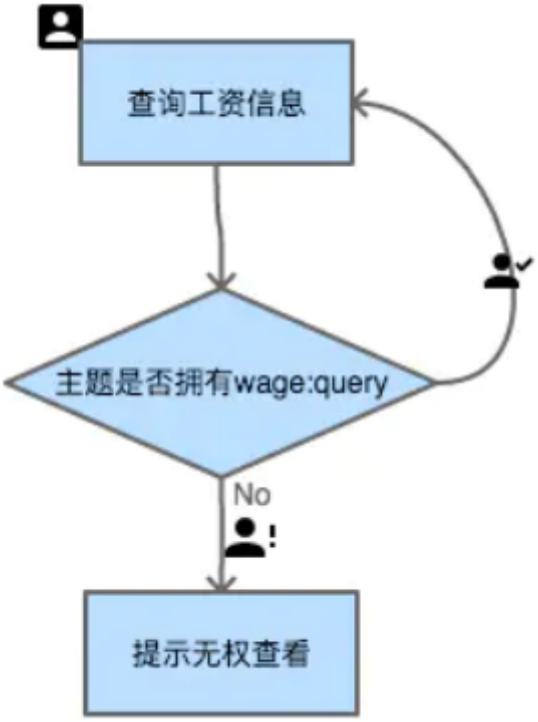
- 主体（账号、密码）
- 角色（角色名称）
- 主体和角色关系（主体id、角色id）
- 权限（权限名称、资源id）
- 角色和权限关系（角色id、权限id）
- 资源（资源id、访问地址）



权限控制

基于角色的访问控制

RBAC基于角色的访问控制（**Role-Based Access Control**）是以角色为中心进行访问控制，比如：主体的角色为总经理可以查询企业运营报表，查询员工工资信息等，访问控制流程如下：



图中的判断逻辑代码可以理解为：

```
if(主体.hasRole("总经理角色id") ||  主体.hasRole("部门经理角色id")){
    查询工资
}
```

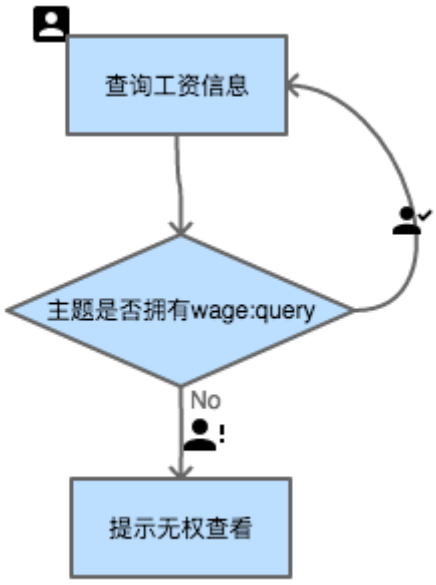
缺点：以角色进行访问控制粒度较粗，如果上图中查询工资所需要的角色变化为总经理和部门经理，此时就需要修改判断逻辑为“**判断主体的角色是否是总经理或部门经理**”，系统可扩展性差。

修改代码如下：

```
if(主体.hasRole("总经理角色id") ||  主体.hasRole("部门经理角色id")){
    查询工资
}
```

基于资源的访问控制

RBAC基于资源的访问控制（**Resource-Based Access Control**）是以资源为中心进行访问控制，比如：主体必须具有查询工资权限才可以查询员工工资信息等，访问控制流程如下：



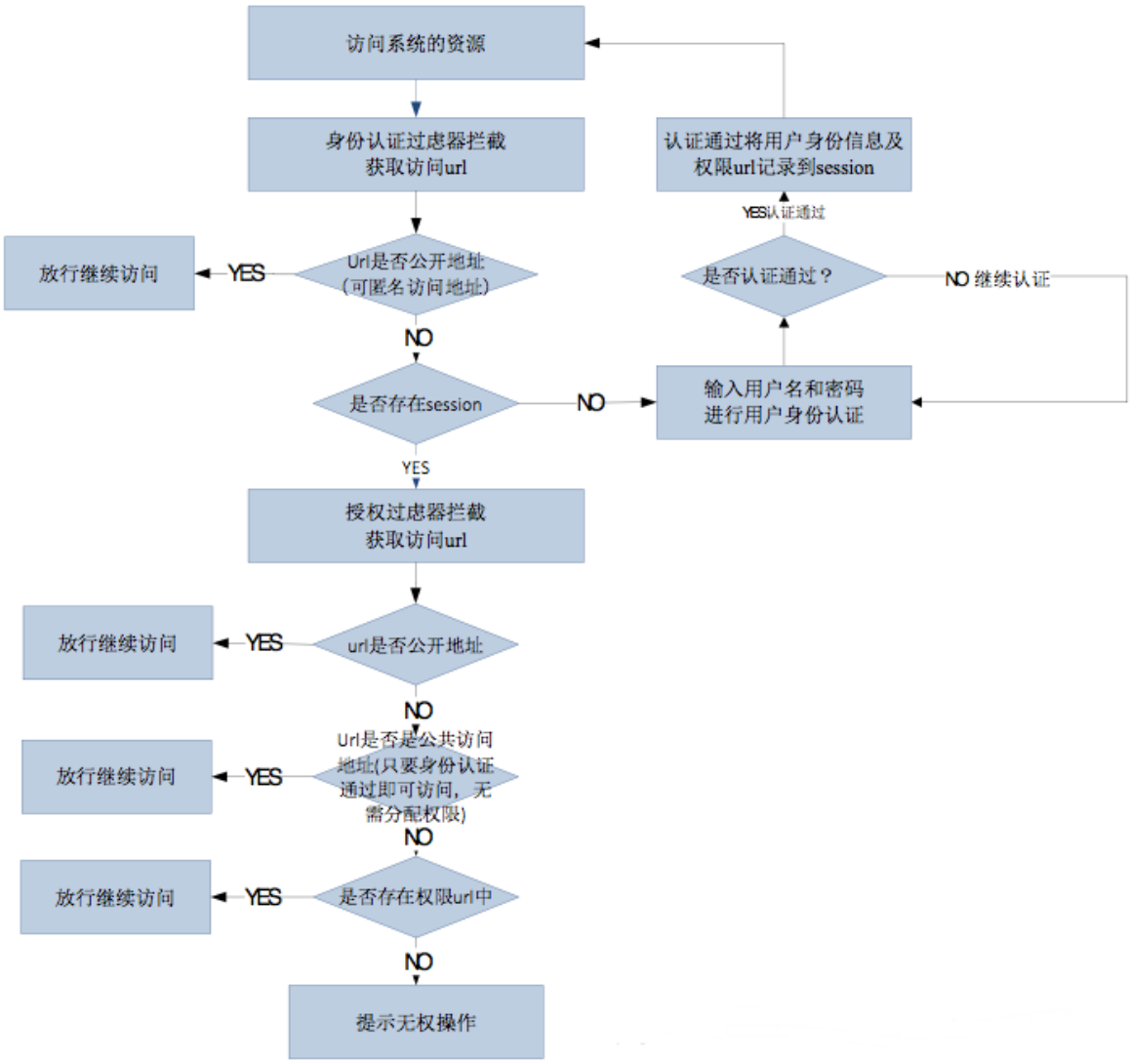
可以理解为：

```
if(主体.hasPermission("wage:query")){
    查询工资
}
```

优点：系统设计时定义好查询工资的**权限标识**，即使查询工资所需要的角色变化为总经理和部门经理也**只需要将"查询工资信息权限"添加到"部门经理角色"的权限列表中**，判断逻辑不用修改，系统可扩展性强。

基于url的访问控制

基于url拦截是企业中常用的权限管理方法，实现思路是：**将系统操作的每个url配置在权限表中，将权限对应到角色，将角色分配给用户，用户访问系统功能通过Filter进行过虑，过虑器获取到用户访问的url，只要访问的url是用户分配角色中的url则放行继续访问。**



粗颗粒度和细颗粒度

什么是粗颗粒度和细颗粒度

对**资源类型**的管理称为粗颗粒度权限管理，即只控制到菜单、按钮、方法，粗粒度的例子比如：用户具有用户管理的权限，具有导出订单明细的权限。

对**资源实例**的控制称为细颗粒度权限管理，即控制到数据级别的权限，比如：用户只允许修改本部门的员工信息，用户只允许导出自己创建的订单明细。

如何实现粗颗粒度和细颗粒度

- **对于粗颗粒度的权限管理可以很容易做系统架构级别的功能，即系统功能操作使用统一的粗颗粒度的权限管理。**
- **对于细颗粒度的权限管理不建议做成系统架构级别的功能，因为对数据级别的控制是系统的业务需求，随着业务需求的变更业务功能变化的可能性很大，建议对数据级别的权限控制在业务层个性化开发**，比如：用户只允许修改自己创建的商品信息可以在service接口添加校验实现，service接口需要传入当前操作人的标识，与商品信息创建人标识对比，不一致则不允许修改商品信息。

Shiro

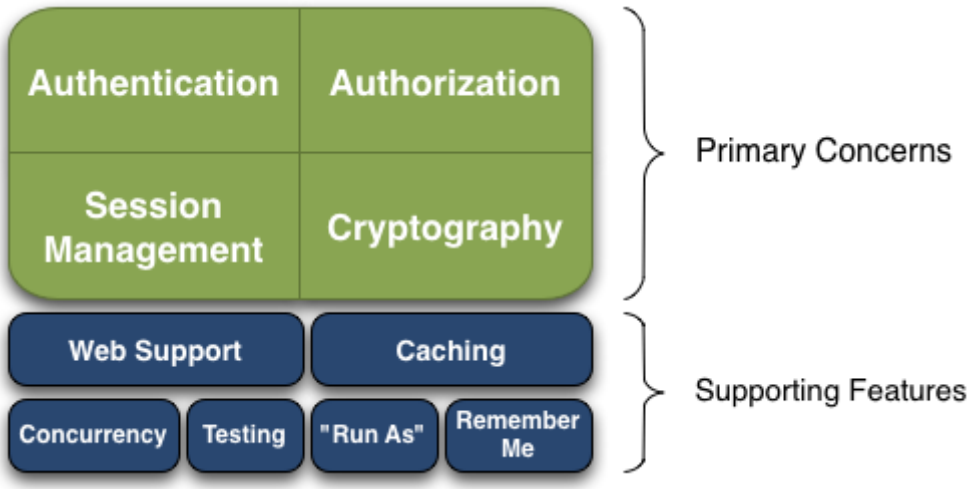
Apache Shiro 是一个功能强大，使用简单的Java安全框架，它为开发人员提供一个直观而全面的认证，授权，加密及会话管理的解决方案。

Shiro能做什么呢?

- 验证用户身份
- 用户访问权限控制
- 在非 web 或 EJB 容器的环境下可以任意使用Session API
- 可以响应认证、访问控制，或者 Session 生命周期中发生的事件
- 可将一个或以上用户安全数据源数据组合成一个复合的用户 "view"(视图)
- 支持单点登录(SSO)功能
- 支持提供“Remember Me”服务，获取用户关联信息而无需登录

Shiro基本功能

Apache Shiro是一个全面的、蕴含丰富功能的安全框架。



Authentication（认证）, Authorization（授权）, Session Management（会话管理）, Cryptography（加密）被 Shiro 框架的开发团队称之为应用安全的四大基石。

- **Authentication（认证）**：身份认证/登录，验证用户是不是拥有相应的身份。
- **Authorization（授权）**：授权，即权限验证，验证某个已认证的用户是否拥有某个权限
- **Session Management（会话管理）**：会话管理，即用户登录后就是一次会话，在没有退出之前，它的所有信息都在会话中
- **Cryptography（加密）**：加密，保护数据的安全性，如密码加密存储到数据库，而不是明文存储。

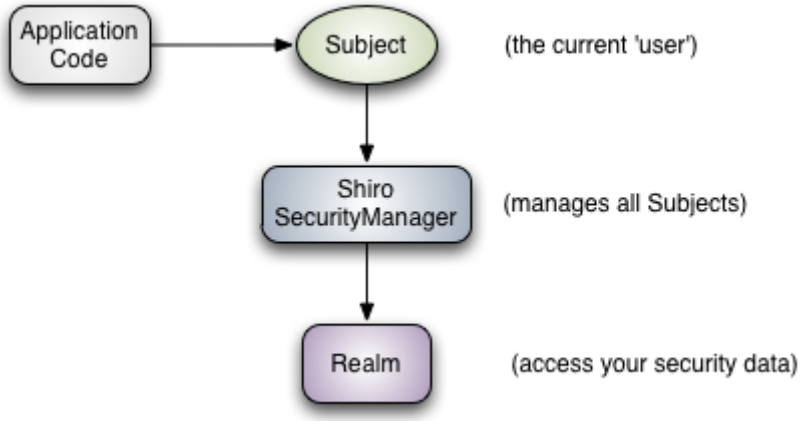
还有其他的功能来支持和加强这些不同应用环境下安全领域的关注点。特别是对以下的功能支持:

- Web Support：Web支持，可以非常容易的集成到 web 环境。
- Caching：缓存，比如用户登录后，其用户信息、拥有的角色/权限不必每次去查，这样可以提高效率。
- Concurrency：shiro 支持多线程应用的并发验证，即如在一个线程中开启另一个线程，能把权限自动传播过去。
- Testing：提供测试支持。
- Run As：允许一个用户假装为另一个用户（如果他们允许）的身份进行访问。
- Remember Me：记住我，这个是非常常见的功能，即一次登录后，下次再来的话不用登录了。

> 注意：Shiro不会去维护用户、维护权限；这些需要我们自己去设计/提供，然后通过相应的接口注入给Shiro即可。

Shiro运行原理

从应用程序角度的来观察如何使用Shiro完成工作



- Subject：主体，任何与应用交互的用户。
- SecurityManager：相当于 SpringMVC 中的 DispatcherServlet 或者 Struts2 中的 StrutsPreparedAndExcutorFilter。**它是 Shiro 的核心，所有具体的交互都通过 SecurityManager 进行控制。它管理着所有 Subject、且负责进行认证和授权、及会话、缓存的管理。**
- Realm：可以有1个或多个 Realm，可以认为是**安全实体数据源，即用于获取安全实体的。**

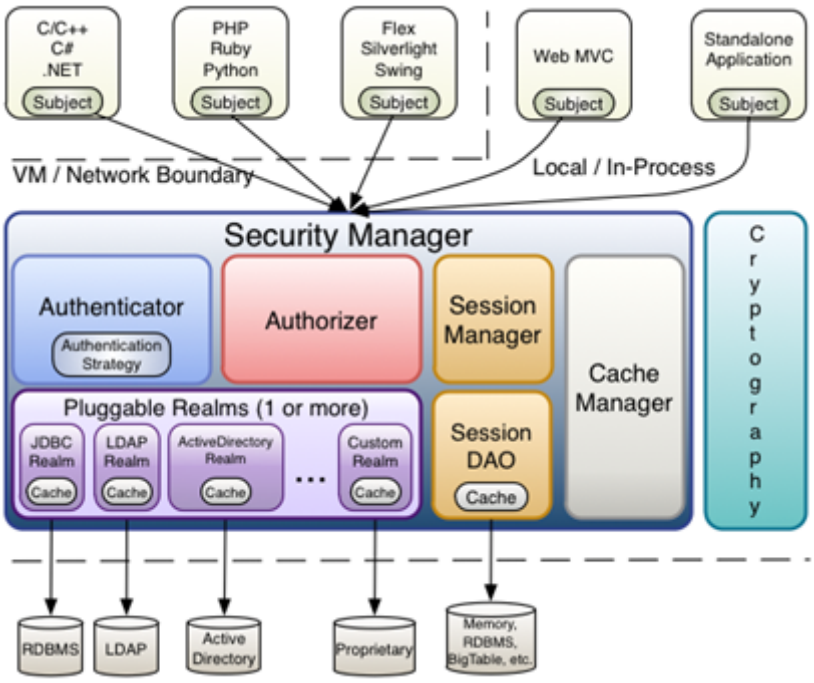
我们需要实现Realms的Authentication 和 Authorization。其中 Authentication 是用来验证用户身份，Authorization 是权限验证

也就是说对于我们而言，最简单的一个Shiro应用：

- 应用代码通过Subject来进行认证和授权，而Subject又委托给SecurityManager；
- 我们需要给Shiro的SecurityManager注入Realm，从而让SecurityManager能得到合法的用户及其权限进行判断。

从以上也可以看出，Shiro不提供维护用户权限，而是通过Realm让开发人员自己注入。

Shiro内部架构



shiro组件如下：

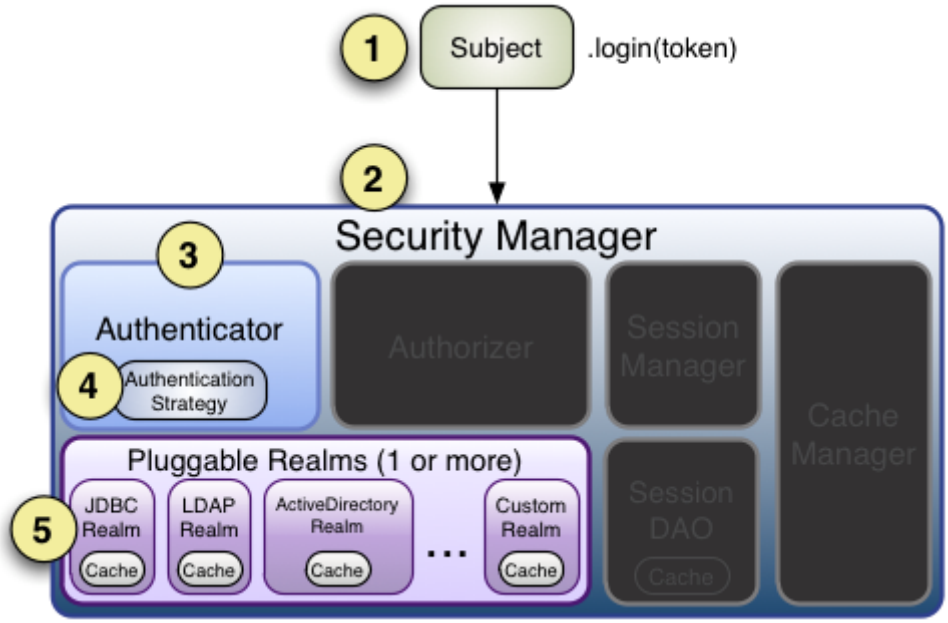
1. Subject：主体，任何与应用交互的用户。

2. SecurityManager：相当于 SpringMVC 中的 DispatcherServlet 或者 Struts2 中的 StrutsPreparedAndExcutorFilter。**它是 Shiro 的核心，所有具体的交互都通过 SecurityManager 进行控制。它管理着所有 Subject、且负责进行认证和授权、及会话、缓存的管理。**
3. Authenticator：**认证器，负责主体认证，用户可以自定义实现，自定义认证策略**
4. Authrizer：**授权器，或者叫访问控制器。它用来决定主体是否有权限进行相应的操作**
5. Realm：可以有1个或多个 Realm，可以认为是**安全实体数据源，即用于获取安全实体的。**
6. SessionManager：会话管理器，**管理session的生命周期**（可以实现单点登录）
7. SessionDAO：数据访问对象，用于会话的 CRUD。
8. CacheManager：缓存管理器。它来管理如用户、角色、权限等的缓存的。
9. Cryptography：密码模块，Shiro 提供了一些常见的加密组件用于如密码加密/解密的。

通过上面的各个组件我们可以完成认证、授权、会话管理、加密/解密、记住我等安全相关功能。

@$Shiro认证授权流程

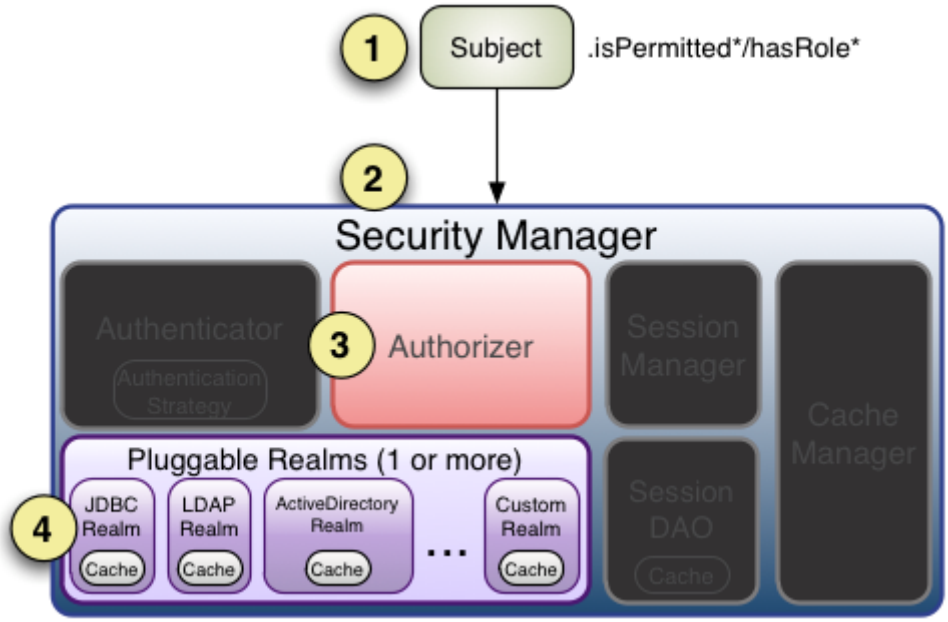
身份认证流程



流程如下：

1. 首先调用 Subject.login(token) 进行登录，其会自动委托给 Security Manager，调用之前必须通过 SecurityUtils.setSecurityManager() 设置；
2. SecurityManager 负责真正的身份验证逻辑；它会委托给 Authenticator 进行身份验证；
3. Authenticator 才是真正的身份验证者，Shiro API 中核心的身份认证入口点，此处可以自定义插入自己的实现；
4. Authenticator 可能会委托给相应的 AuthenticationStrategy 进行多 Realm 身份验证，默认 ModularRealmAuthenticator 会调用 AuthenticationStrategy 进行多 Realm 身份验证；
5. Authenticator 会把相应的 token 传入 Realm，从 Realm 获取身份验证信息，如果没有返回 / 抛出异常表示身份验证失败了。此处可以配置多个 Realm，将按照相应的顺序及策略进行访问。

授权流程



流程如下：

1. 首先调用 `Subject.isPermitted*/hasRole*` 接口，其会委托给 SecurityManager，而 SecurityManager 接着会委托给 Authorizer；
2. Authorizer 是真正的授权者，如果我们调用如 isPermitted(“user:view”)，其首先会通过 PermissionResolver 把字符串转换成相应的 Permission 实例；
3. 在进行授权之前，Authorizer 会调用相应的 Realm 获取 Subject 相应的角色/权限用于匹配传入的角色/权限，如果有多个 Realm，会委托给 ModularRealmAuthorizer 进行循环判断，如果匹配如 `isPermitted*/hasRole*` 会返回 true，否则返回 false 表示授权失败。继承 AuthorizingRealm 而不是实现 Realm 接口；
4. 推荐继承 AuthorizingRealm，重写认证和授权方法：AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token)：表示获取用户认证信息；AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals)：表示根据用户身份获取授权信息。这种方式的好处是当只需要身份验证时只需要获取身份验证信息而不需要获取授权信息。

@$Shiro和Spring Security比较

shiro和spring security都是安全框架，都具有认证授权，加密等功能，主要区别如下：

1. Shiro 的配置和使用比较简单，Spring Security 上手复杂些
2. Spring Security 有更好的社区支持，社区资源相对比 Shiro 更加丰富
3. Shiro 依赖性低，不需要任何框架和容器，可以独立运行。Spring Security 依赖Spring容器，与Spring整合更加方便
4. Spring Security 功能比 Shiro 更加丰富些，例如安全维护方面

OAuth2 协议

什么是第三方登录

什么是第三方登录

很多网站登录时，允许使用第三方网站的身份来进行登录，这称为“第三方登录”。比如知乎和慕课网等，可以使用微信，QQ，或微博来进行登录。一个网站想接入第三方登录，需要用到OAuth2这个协议。

什么是OAuth2

OAuth2是一个关于授权的开放网络标准，用来授权第三方应用，获取用户的数据。其最终的目的是为了给第三方应用颁发一个有时效性的令牌access_token，第三方应用根据这个access_token就可以去获取用户的相关资源，如头像，昵称，email这些信息。现在大家用的基本是2.0的版本。

协议流程

在详细介绍OAuth2协议流程之前，先来简单了解几个角色，方便后续的理解。

- Client，客户端，第三方应用，可以是浏览器、移动设备或者服务器；
- Resource Owner，资源所有者，拥有该资源的最终用户，他有访问资源的账号密码；
- Authorization Server，授权服务器，一般和资源服务器是同一家公司的应用，主要是用来处理授权，给客户端颁发令牌
- Resource Server，资源服务器，托管受保护资源的服务器
- User-agent，用户代理，一般为web浏览器，在手机上就是app

了解了上面这些角色之后，来看下OAuth2.0的运行流程是怎么样的。

```
+--------+                               +---------------+
|        |--(A)- Authorization Request ->|   Resource    |
|        |                               |     Owner     |
|        |<-(B)-- Authorization Grant ---|               |
|        |                               +---------------+
|        |
|        |                               +---------------+
|        |--(C)-- Authorization Grant -->| Authorization |
| Client |                               |     Server    |
|        |<-(D)----- Access Token -------|               |
|        |                               +---------------+
|        |
|        |                               +---------------+
|        |--(E)----- Access Token ------>|    Resource   |
|        |                               |     Server    |
```

```
|       |<-(F)--- Protected Resource ---|               |
+--------+                               +---------------+
```

(A). 客户端向用户(Resource Owner)发送一个授权请求

(B). 用户同意给客户端(Client)授权，并返回一个授权码（code）

(C). 客户端使用刚才的授权码（code）去向授权服务器(Authorization Server)授权

(D). 授权服务器校验通过后，会给客户端发放令牌(Access Token)

(E). 客户端拿着令牌(Access Token)，去向资源服务器(Resource Server)申请获取资源

(F). 资源服务器确认令牌之后，给客户端返回受保护的资源(Protected Resource)

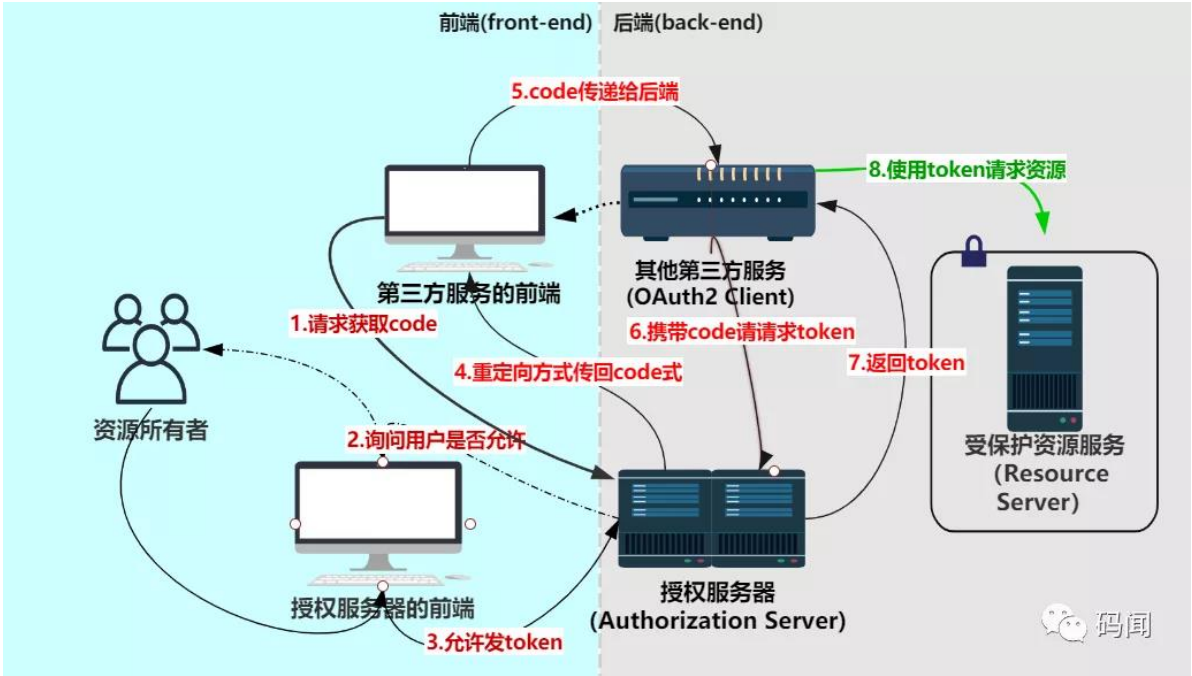
授权方式

在OAuth2当中，定义了四种授权方式，针对不同的业务场景：

- Authorization Code（授权码模式）：正宗的OAuth2的授权模式，客户端先将用户导向授权服务器，登录后获取授权码，然后进行授权，最后根据授权码获取访问令牌；
- Implicit（简化模式）：和授权码模式相比，取消了获取授权码的过程，直接获取访问令牌；
- Resource Owner Password Credentials（密码模式）：客户端直接向用户获取用户名和密码，之后向授权服务器获取访问令牌；
- Client Credentials（客户端模式）：客户端直接通过客户端认证（比如client_id和client_secret）从授权服务器获取访问令牌。

授权码模式

OAuth2授权码模式完整流程图：



Netty

Netty 是什么?

Netty是一个异步事件驱动的网络应用程序框架，用于快速开发高性能通信的服务器和客户端。Netty是基于nio的，它封装了jdk的nio，让我们使用起来更加方便灵活。

Netty 高性能表现在哪些方面？Netty 的优势有哪些?

- 使用简单：封装了 NIO 的很多细节，提供了易于使用调用接口。
- 性能高，IO 线程模型：Netty 是一款基于 NIO开发的网络通信框架，对比BIO，并发性能得到了很大提高。通过与其他业界主流的 NIO 框架对比，Netty 的综合性能最优。
- 内存零拷贝：尽量减少不必要的内存拷贝，实现了更高效率的传输。
- 功能强大：预置了多种编解码功能，支持多种主流协议。支持 protobuf 等高性能序列化协议。
- 定制能力强：可以通过 ChannelHandler 对通信框架进行灵活地扩展。
- 稳定：Netty 修复了已经发现的所有 NIO 的 bug，让开发人员可以专注于业务本身。
- 社区活跃：Netty 是活跃的开源项目，版本迭代周期短，bug 修复速度快。

Netty的线程模型?

Netty通过Reactor模型基于多路复用器接收并处理用户请求，内部实现了两个线程池，boss线程池和work线程池，其中boss线程池的线程负责处理请求的accept事件，当接收到accept事件的请求时，把对应的socket封装到一个NioSocketChannel中，并交给work线程池，其中work线程池负责请求的read和write事件，由对应的Handler处理。

单线程模型：所有I/O操作都由一个线程完成，即多路复用、事件分发和处理都是在一个Reactor线程上完成的。既要接收客户端的连接请求,向服务端发起连接，又要发送/读取请求或应答/响应消息。一个 NIO 线程同时处理成百上千的链路，性能上无法支撑，速度慢，若线程进入死循环，整个程序不可用，对于高负载、大并发的应用场景不合适。

多线程模型：有一个NIO 线程（Acceptor） 只负责监听服务端，接收客户端的TCP 连接请求；NIO 线程池负责网络IO 的操作，即消息的读取、解码、编码和发送；1 个NIO 线程可以同时处理N 条链路，但是 1 个链路只对应1 个NIO 线程，这是为了防止发生并发操作问题。但在并发百万客户端连接或需要安全认证时，一个Acceptor 线程可能会存在性能不足问题。

主从多线程模型：Acceptor 线程用于绑定监听端口，接收客户端连接，将SocketChannel 从主线程池的 Reactor 线程的多路复用器上移除，重新注册到Sub 线程池的线程上，用于处理I/O 的读写等操作，从而保证mainReactor只负责接入认证、握手等操作；

TCP 粘包/拆包的原因及解决方法?

TCP是以流的方式来处理数据，一个完整的包可能会被TCP拆分成多个包进行发送，也可能把小的封装成一个大的数据包发送。

TCP粘包/分包的原因

应用程序写入的字节大小大于套接字发送缓冲区的大小，会发生拆包现象，而应用程序写入数据小于套接字缓冲区大小，网卡将应用多次写入的数据发送到网络上，这将会发生粘包现象；

解决方法

- 消息定长：FixedLengthFrameDecoder类
- 包尾增加特殊字符分割：行分隔符类：LineBasedFrameDecoder；或自定义分隔符类 ：DelimiterBasedFrameDecoder
- 将消息分为消息头和消息体：LengthFieldBasedFrameDecoder类。分为有头部的拆包与粘包、长度字段在前且有头部的拆包与粘包、多扩展头部的拆包与粘包。

什么是 Netty 的零拷贝?

Netty 的零拷贝主要包含三个方面：

- Netty 的接收和发送 ByteBuffer 采用 DIRECT BUFFERS，使用堆外直接内存进行 Socket 读写，不需要进行字节缓冲区的二次拷贝。如果使用传统的堆内存（HEAP BUFFERS）进行 Socket 读写，

JVM 会将堆内存 Buffer 拷贝一份到直接内存中，然后才写入 Socket 中，相比于堆外直接内存，消息在发送过程中多了一次缓冲区的内存拷贝。

- Netty 提供了组合 Buffer 对象，可以聚合多个 ByteBuffer 对象，用户可以像操作一个 Buffer 那样方便的对组合 Buffer 进行操作，避免了传统通过内存拷贝的方式将几个小 Buffer 合并成一个大的 Buffer。
- Netty 的文件传输采用了 transferTo 方法，它可以直接将文件缓冲区的数据发送到目标 Channel，避免了传统通过循环 write 方式导致的内存拷贝问题。

Netty常见使用场景

Netty常见的使用场景如下：

- 互联网行业 在分布式系统中，各个节点之间需要远程服务调用，高性能的RPC框架必不可少，Netty作为异步高新能的通信框架,往往作为基础通信组件被这些RPC框架使用。典型的应用有：阿里分布式服务框架Dubbo的RPC框架使用Dubbo协议进行节点间通信，Dubbo协议默认使用Netty作为基础通信组件，用于实现各进程节点之间的内部通信。
- 游戏行业 无论是手游服务端还是大型的网络游戏，Java语言得到了越来越广泛的应用。Netty作为高性能的基础通信组件，它本身提供了TCP/UDP和HTTP协议栈。非常方便定制和开发私有协议栈，账号登录服务器，地图服务器之间可以方便的通过Netty进行高性能的通信
- 大数据领域 经典的Hadoop的高性能通信和序列化组件Avro的RPC框架，默认采用Netty进行跨界点通信，它的Netty Service基于Netty框架二次封装实现

工作原理架构

初始化并启动Netty服务端过程如下：

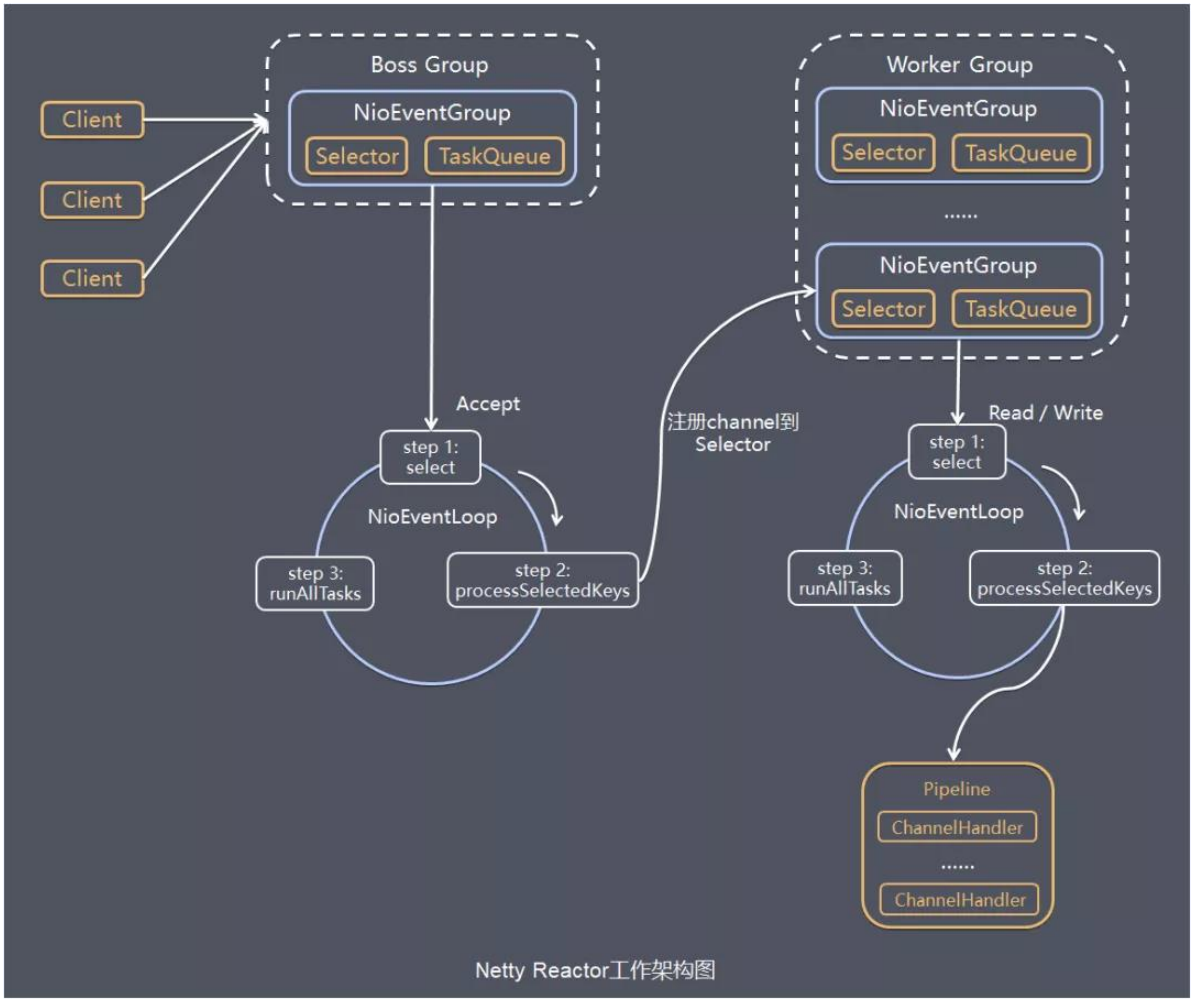
```
public static void main(String[] args) {
   // 创建mainReactor
   NioEventLoopGroup boosGroup = new NioEventLoopGroup();
   // 创建工作线程组
   NioEventLoopGroup workerGroup = new NioEventLoopGroup();

   final ServerBootstrap serverBootstrap = new ServerBootstrap();
   serverBootstrap
       // 组装NioEventLoopGroup
      .group(boosGroup, workerGroup)
       // 设置channel类型为NIO类型
      .channel(NioServerSocketChannel.class)
       // 设置连接配置参数
      .option(ChannelOption.SO_BACKLOG, 1024)
      .childOption(ChannelOption.SO_KEEPALIVE, true)
      .childOption(ChannelOption.TCP_NODELAY, true)
       // 配置入站、出站事件handler
      .childHandler(new ChannelInitializer<NioSocketChannel>() {
           @Override
           protected void initChannel(NioSocketChannel ch) {
               // 配置入站、出站事件channel
               ch.pipeline().addLast(...);
               ch.pipeline().addLast(...);
           }
      });

   // 绑定端口
   int port = 8080;
   serverBootstrap.bind(port).addListener(future -> {
       if (future.isSuccess()) {
```

```
            System.out.println(new Date() + ": 端口[" + port + "]绑定成功!");
        } else {
            System.out.println("端口[" + port + "]绑定失败!");
        }
    });
}
```

基本过程如下：

1. 初始化创建2个NioEventLoopGroup，其中boosGroup用于Accetpt连接建立事件并分发请求，workerGroup用于处理I/O读写事件和业务逻辑
2. 基于ServerBootstrap(服务端启动引导类)，配置EventLoopGroup、Channel类型，连接参数、配置入站、出站事件handler
3. 绑定端口，开始工作

结合上面的介绍的Netty Reactor模型，介绍服务端Netty的工作架构图：



Zookeeper

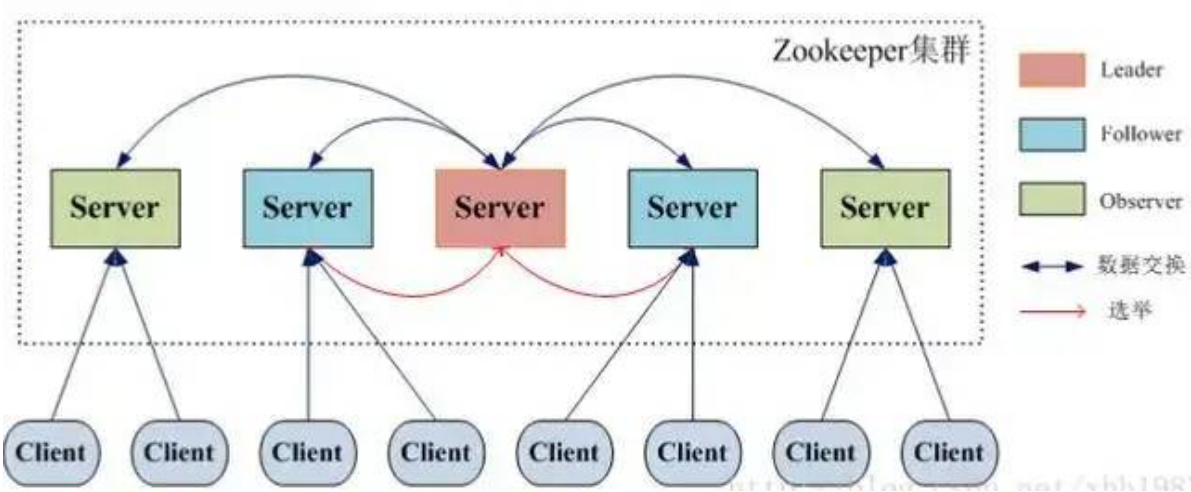
ZooKeeper 是什么？Zookeeper的用途，使用场景

ZooKeeper 是一个开源的分布式协调服务。它是一个为分布式应用提供一致性服务的软件，分布式应用程序可以基于 Zookeeper 实现诸如数据发布/订阅、负载均衡、命名服务、分布式协调/通知、集群管理、Master 选举、分布式锁和分布式队列等功能。

ZooKeeper 的目标就是封装好复杂易出错的关键服务，将简单易用的接口和性能高效、功能稳定的系统提供给用户。

zookeeper原理？选举的原理是什么？

Zookeeper集群



Zookeeper的角色

角色		描述
领导者（Leader）		领导者负责进行投票的发起和决议，更新系统状态
学习者（Learner）	跟随者（Follower）	Follower 用于接收客户请求并向客户端返回结果，在选主过程中参与投票
	观察者（ObServer）	ObServer 可以接收客户端连接，将写请求转发给 leader 节点。但 ObServer 不参加投票过程，只同步 leader 的状态。ObServer 的目的是为了扩展系统，提高读取速度
客户端（Client）		请求发起方

Zookeeper工作原理

Zookeeper的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。

Zab协议有两种模式，它们分 别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和 leader的状态同步以后，恢复模式就结束了。恢复模式结束后，Zab进入广播模式，状态同步保证了leader和Server具有相同的系统状态。

为了保证事务的顺序一致性，zookeeper采用了递增的事务id号（zxid）来标识事务。所有的提议（proposal）都在被提出的时候加上 了zxid。实现中zxid是一个64位的数字，它高32位是epoch用来标识leader关系是否改变，每次一个leader被选出来，它都会有一个 新的epoch，标识当前属于那个leader的统治时期。低32位用于递增计数。

每个Server在工作过程中有三种状态：

- LOOKING：当前Server不知道leader是谁，正在搜寻
- LEADING：当前Server即为选举出来的leader
- FOLLOWING：leader已经选举出来，当前Server与之同步

leader选举原理

半数通过

当leader崩溃或者leader失去大多数的follower，这时候zk进入恢复模式，恢复模式需要重新选举出一个新的leader，让所有的 Server都恢复到一个正确的状态。

Zk的选举算法有两种：一种是基于basic paxos实现的，另外一种是基于fast paxos算法实现的。系统默认的选举算法为fast paxos。

- A提案说，我要选自己，B你同意吗？C你同意吗？B说，我同意选A；C说，我同意选A。(注意，这里超过半数了，其实在现实世界选举已经成功了。但是计算机世界是很严格，另外要理解算法，要继续模拟下去。)
- 接着B提案说，我要选自己，A你同意吗；A说，我已经超半数同意当选，你的提案无效；C说，A已经超半数同意当选，B提案无效。
- 接着C提案说，我要选自己，A你同意吗；A说，我已经超半数同意当选，你的提案无效；B说，A已经超半数同意当选，C的提案无效。

选举已经产生了Leader，后面的都是follower，只能服从Leader的命令。而且这里还有个小细节，就是其实谁先启动谁当头。

Zookeeper Watcher 机制 -- 数据变更通知

Zookeeper 允许客户端向服务端的某个 Znode 注册一个 Watcher 监听，当服务端的一些指定事件触发了这个 Watcher，服务端会向指定客户端发送一个事件通知来实现分布式的通知功能，然后客户端根据 Watcher 通知状态和事件类型做出业务上的改变。

工作机制:

1. 客户端注册 watcher
2. 服务端处理 watcher
3. 客户端回调 watcher

Watcher 特性总结

1. 一次性

   无论是服务端还是客户端，一旦一个 Watcher 被 触 发 ，Zookeeper 都会将其从相应的存储中移除。这样的设计有效的减轻了服务端的压力，不然对于更新非常频繁的节点，服务端会不断的向客户端发送事件通知，无论对于网络还是服务端的压力都非常大。

2. 客户端串行执行

   客户端 Watcher 回调的过程是一个串行同步的过程。

3. 轻量

   - Watcher 通知非常简单，只会告诉客户端发生了事件，而不会说明事件的具体内容。
   - 客户端向服务端注册 Watcher 的时候，并不会把客户端真实的 Watcher 对象实体传递到服务端，仅仅是在客户端请求中使用 boolean 类型属性进行了标记。

4. watcher event 异步发送 watcher 的通知事件从 server 发送到 client 是异步的，这就存在一个问题，不同的客户端和服务器之间通过 socket 进行通信，由于网络延迟或其他因素导致客户端在不通的时刻监听到事件，由于 Zookeeper 本身提供了 ordering guarantee，即客户端监听事件后，才会感知它所监视 znode发生了变化。所以我们使用 Zookeeper 不能期望能够监控到节点每次的变化。Zookeeper 只能保证最终的一致性，而无法保证强一致性。

5. 注册 watcher getData、exists、getChildren

6. 触发 watcher create、delete、setData

7. 当一个客户端连接到一个新的服务器上时，watch 将会被以任意会话事件触发。当与一个服务器失去连接的时候，是无法接收到 watch 的。而当 client 重新连接时，如果需要的话，所有先前注册过的 watch，都会被重新注册。通常这是完全透明的。只有在一个特殊情况下，watch 可能会丢失：对于一个未创建的 znode的 exist watch，如果在客户端断开连接期间被创建了，并且随后在客户端连接上之前又删除了，这种情况下，这个 watch 事件可能会被丢失。

客户端注册 Watcher 实现

1. 调用 getData()/getChildren()/exist()三个 API，传入 Watcher 对象
2. 标记请求 request，封装 Watcher 到 WatchRegistration
3. 封装成 Packet 对象，发服务端发送 request
4. 收到服务端响应后，将 Watcher 注册到 ZKWatcherManager 中进行管理
5. 请求返回，完成注册。

服务端处理 Watcher 实现

1. 服务端接收 Watcher 并存储

   接收到客户端请求，处理请求判断是否需要注册 Watcher，需要的话将数据节点的节点路径和 ServerCnxn（ServerCnxn 代表一个客户端和服务端的连接，实现了 Watcher 的 process 接口，此时可以看成一个 Watcher 对象）存储在WatcherManager 的 WatchTable 和 watch2Paths 中去。

2. Watcher 触发

   以服务端接收到 setData() 事务请求触发 NodeDataChanged 事件为例：

   1. 封装 WatchedEvent

      将通知状态（SyncConnected）、事件类型（NodeDataChanged）以及节点路径封装成一个 WatchedEvent 对象

   2. 查询 Watcher

      从 WatchTable 中根据节点路径查找 Watcher

   3. 没找到；说明没有客户端在该数据节点上注册过 Watcher

   4. 找到；提取并从 WatchTable 和 Watch2Paths 中删除对应 Watcher（从这里可以看出 Watcher 在服务端是一次性的，触发一次就失效了）

3. 调用 process 方法来触发 Watcher

   这里 process 主要就是通过 ServerCnxn 对应的 TCP 连接发送 Watcher 事件通知。

zk 节点宕机如何处理?

Zookeeper 本身也是集群，推荐配置不少于 3 个服务器。Zookeeper 自身也要保证当一个节点宕机时，其他节点会继续提供服务。

如果是一个 Follower 宕机，还有 2 台服务器提供访问，因为 Zookeeper 上的数据是有多个副本的，数据并不会丢失；如果是一个 Leader 宕机，Zookeeper 会选举出新的 Leader。

ZK 集群的机制是只要超过半数的节点正常，集群就能正常提供服务。只有在 ZK节点挂得太多，只剩一半或不到一半节点能工作，集群才失效。

所以

3 个节点的 cluster 可以挂掉 1 个节点(leader 可以得到 2 票>1.5)

2 个节点的 cluster 就不能挂掉任何 1 个节点了(leader 可以得到 1 票<=1)

WebSocket

为什么需要 WebSocket?

因为 HTTP 协议有一个缺陷：通信只能由客户端发起，服务器不能主动联系客户端。

举例来说，我们想了解今天的天气，只能是客户端向服务器发出请求，服务器返回查询结果。HTTP 协议做不到服务器主动向客户端推送信息。

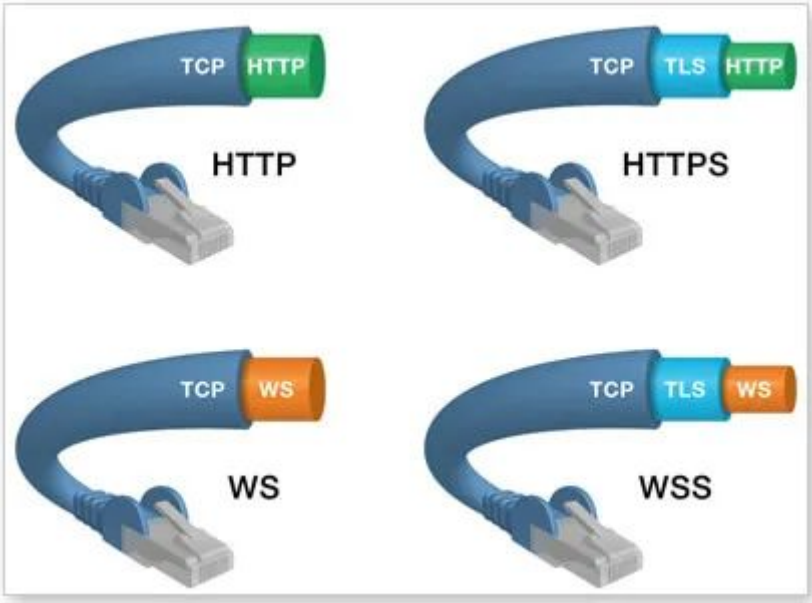
这种单向请求的特点，注定了如果服务器有连续的状态变化，客户端要获知就非常麻烦。我们只能使用轮询：每隔一段时候，就发出一个询问，了解服务器有没有新的信息。

轮询的缺点：轮询的效率低，非常浪费资源（因为必须不停连接，或者 HTTP 连接始终打开）。因此，工程师们一直在思考，有没有更好的方法。WebSocket 就是这样发明的。

什么是WebSocket

WebSocket 协议在2008年诞生，2011年成为国际标准。所有浏览器都已经支持了。WebSocket是HTML5的一个新协议。

WebSocket最大的特点就是实现**全双工通信**：客户端能够实时推送消息给服务端，服务端也能够实时推送消息给客户端，属于服务器推送技术的一种。



其他特点包括：

1. 建立在 TCP 协议之上，服务器端的实现比较容易。
2. 与 HTTP 协议有着良好的兼容性。默认端口也是80和443，并且握手阶段采用 HTTP 协议，因此握手时不容易屏蔽，能通过各种 HTTP 代理服务器完成。
3. 数据格式比较轻量，性能开销小，通信高效。
4. 可以发送文本，也可以发送二进制数据。
5. 没有同源限制，客户端可以与任意服务器通信。
6. 协议标识符是ws（如果加密，则为wss），服务器网址就是 URL。

WebSocket 底层原理

TCP是持久连接、全双工

TCP是持久连接，建立TCP连接是3次握手，关闭TCP连接是4次挥手。TCP连接是由通信双方（应用层）来决定什么时候关闭，其本身是一个持久连接。TCP连接可以进行全双工通信，因为双方都知道对方是谁

HTTP只能单向通信、无状态

Http协议只能单向通信的原因是：Server服务端没有保存Http客户端的信息（无状态的），想要通信的时候找不到人。而Http1.1协议新增的keep-alive Header之后，Server会保存连接，即长连接。但是每次都是http请求，一堆没用的信息（http head），浪费资源。而且本质没有变化，都需要客户端请求才能获得数据，增加了keep-alive请求头只是可以通过一条通道请求多次。

WebSocket 底层原理

WebSocket协议实现全双工通信、以及持久连接的一个前提是，它是**基于TCP的。**

WebSocket协议也需要通过已建立的TCP连接来传输数据。具体实现上是通过http协议建立通道，然后在此基础上用真正的WebSocket协议进行通信。

WebSocket 本质上跟 HTTP 完全不一样，只不过**为了兼容性**，WebSocket 的握手是以 HTTP 的形式发起的，

总结重点

WebSocket是一个网络上的应用层协议，它依赖于HTTP协议的第一次握手，握手成功后，数据就通过TCP/IP协议传输了。

WebSocket = “HTTP第1次握手” + TCP的“全双工“通信 的网络协议。

主要过程：

- 首先，通过HTTP第一次握手保证连接成功。
- 其次，再通过TCP实现浏览器与服务器全双工(full-duplex)通信。（通过不断发ping包、pang包保持心跳）

最终，使得 “**服务端**” 拥有 “**主动**” 发消息给 “**客户端**” 的能力。

WebSocket分为握手阶段和数据传输阶段，即进行了HTTP一次握手 + 双工的TCP连接。

1、握手阶段

首先，客户端发送消息：

```
GET /chat HTTP/1.1
Host: server.qishare.org
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://qishare.org
Sec-WebSocket-Version: 13
```

然后，服务端返回消息：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

- Sec-WebSocket-Version表明客户端所使用的协议版本
- 响应的状态码101，表示切换了协议，说明利用http建立传输层的TCP连接，之后便与http协议无关的
- Sec-WebSocket-Key是一个Base64编码值，由浏览器随机生成。是一种验证服务端支不支持websocket的算法
- Sec-Websocket-accept=base64(sha1(key)+常量)，如果返回的accept和算出来的相同，说明服务端支持
- 如果返回成功，Websocket就会回调`onopen`事件

2、传输阶段

WebSocket是以`frame`的形式传输数据的。比如会将一条消息分为几个`frame`，按照先后顺序传输出去。

这样做会有几个好处：

- 较大的数据可以**分片传输**，不用考虑到数据大小导致的长度标志位不足够的情况。
- 和HTTP的chunk一样，可以边生成数据边传递消息，即提高传输效率。

WebSocket协议的优缺点

优点：

- WebSocket协议一旦建议后，互相沟通所消耗的请求头是很小的
- 服务器可以主动向客户端推送消息了

缺点：

- 少部分浏览器不支持，浏览器支持的程度与方式有区别

WebSocket 应用场景

1. **IM（即时通讯）**

典型例子：微信、QQ等，当然，用户量如果非常大的话，仅仅依靠WebSocket肯定是不够的，各大厂应该也有自己的一些优化的方案与措施。但对于用户量不是很大的即时通讯需求，使用WebSocket是一种不错的方案。

2. **游戏（多人对战）**

典型例子：王者荣耀等（应该都玩过）

3. **协同编辑（共享文档）**

多人同时编辑同一份文档时，可以实时看到对方的操作。这时，就用上了WebSocket。

4. **直播/视频聊天**

对音频/视频需要较高的实时性。

5. **股票/基金等金融交易平台**

对于股票/基金的交易来说，每一秒的价格可能都会发生变化。

6. **IoT（物联网 / 智能家居）**

例如，我们的App需要实时的获取智能设备的数据与状态。这时，就需要用到WebSocket。

只要是一些对 “**实时性**” 要求比较高的需求，可能就会用到WebSocket。

其他

nginx请求转发策略

如果不用quartz这样的定时任务框架，怎么处理定时任务

jwt由什么组成，有什么优缺点

生成唯一id的方法