从组件到（微）服务

服务器引擎部-王奎

什么是组件

- 定义：
  - 组件（COMPONENT）是对数据和方法的简单封装。
- 模块：
  - （ MODULE）是一套一致而互相有紧密关连的软件组织。它分别包含了程序和数据结构两部分。

完成特定通用功能的程序集合

组件 VS 模块

- 组件和模块都用于指代一组函数或函数的一部分。 模块更多是逻辑性的，例如：ERP系统中的财务模块， HR模块，制造模块..... 另一方面，组件更具有物理性。 在软件中，它可以是一个DLL，OCX，EXE，...
- 没有标准来衡量哪一个大于另一个。 一个组件可以包含模块列表，一个模块也可以包含许多组件。 组件用于在技术视图中对系统建模，模块用于在功能视图中对系统进行建模（系统的功能）

模块是特定系统中的逻辑组成部分或业务功能的集合

WHY COMPONENT

- 基于组件的软件工程（COMPONENT-BASED SOFTWARE ENGINEERING，简称CBSE）或基于组件的开发（COMPONENT-BASED DEVELOPMENT，简称CBD）是一种**软件开发范型**。它是现今**软件复用理论实用化**的研究热点，在组件对象模型的支持下，通过复用已有的组件，软件开发者可以“即插即用”地快速构造应用软件。这样不仅可以节省时间和经费，提高工作效率，而且可以产生更加规范、更加可靠的应用软件。

组件与组织

“organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations.”

–M. Conway

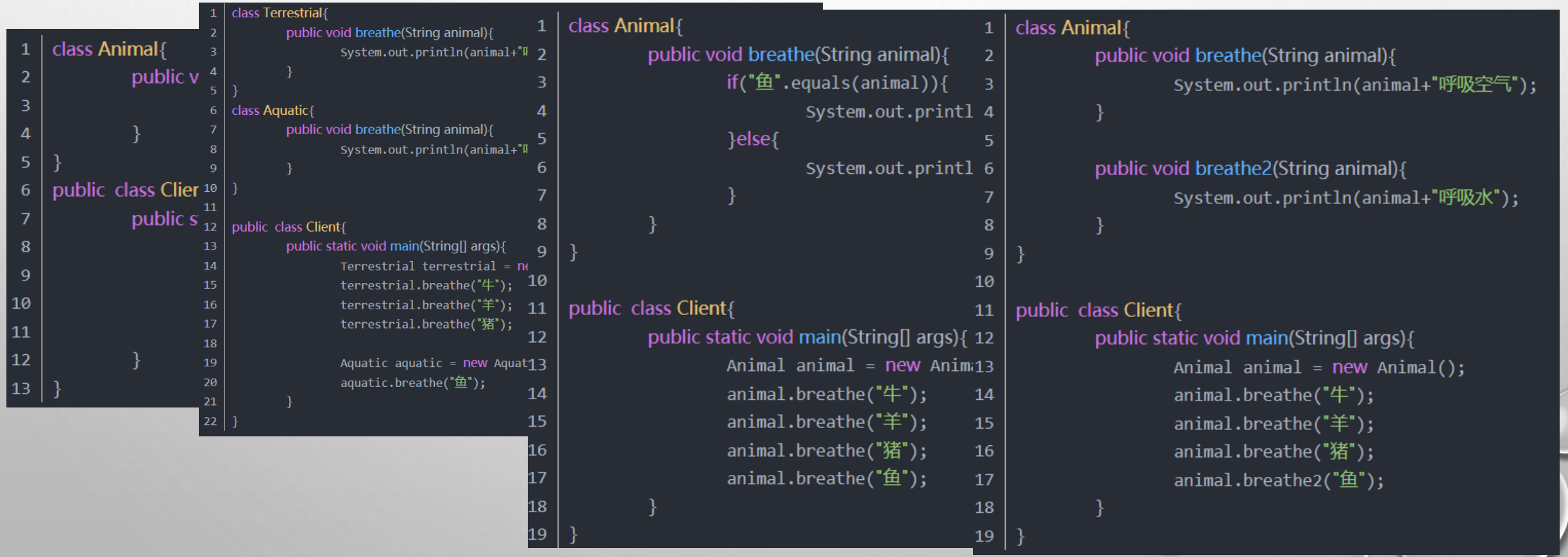
做一件事需要特定的组织结构
组织的结构也会反映到产品中去

任意一个软件都能反映出其制作团队的组织结构，这是因为人们会以反映他们组织形式的方式工作

组件设计原则-SRP

- 单一职责原则（SRP）
  - 一个类应该仅有一个引起它变化的原因。
  - 类T负责两个不同的职责：职责P1，职责P2。当由于职责P1需求发生改变而需要修改类T时，有可能会导致原本运行正常的职责P2功能发生故障。
- 变化的原因"(A REASON FOR CHANGE)。
  - 如果你能够想到多于一个的动机去改变类，那么这个类就具有多于一个的职责。
- 而单一职责原则正是反映了《代码大全》中所说的：软件的首要技术使命--管理复杂度，而找出容易变化改变的区域，隔离变化，就是一种很好的管理复杂度的启发方法。

会呼吸的鱼



```
class Animal{
        public v

        }
}
public  class Clier
        public s

        }
}
```

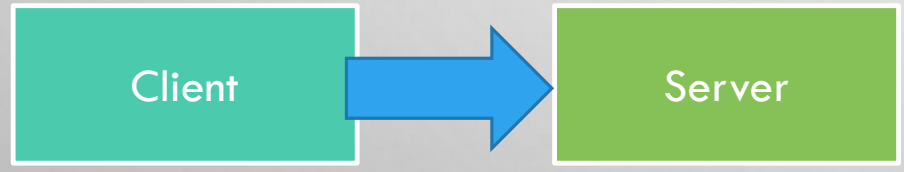
```
class Terrestrial{
        public void breathe(String animal){
                System.out.println(animal+"
        }
}
class Aquatic{
        public void breathe(String animal){
                System.out.println(animal+"
        }
}

public  class Client{
        public static void main(String[] args){
                Terrestrial terrestrial = n
                terrestrial.breathe("牛");
                terrestrial.breathe("羊");
                terrestrial.breathe("猪");

                Aquatic aquatic = new Aquat
                aquatic.breathe("鱼");
        }
}
```

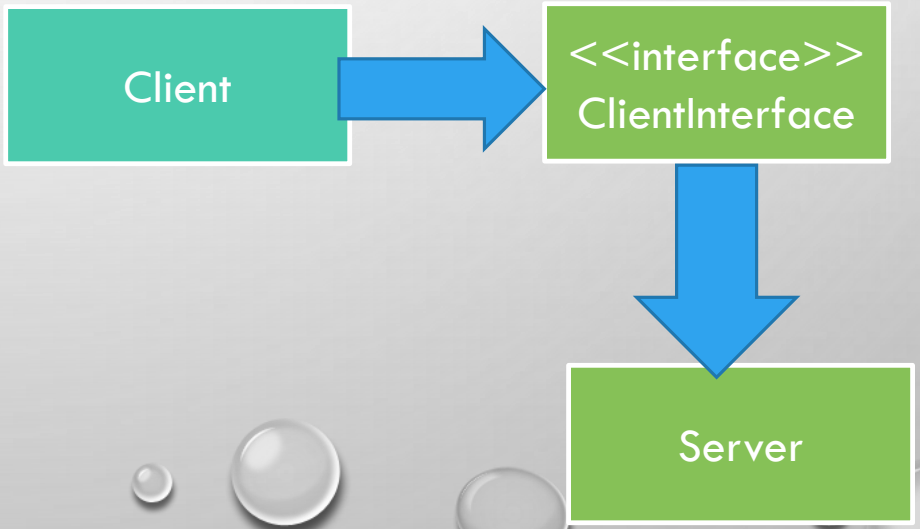
```
class Animal{
        public void breathe(String animal){
                if("鱼".equals(animal)){
                        System.out.printl
                }else{
                        System.out.printl
                }
        }
}

public  class Client{
        public static void main(String[] args){
                Animal animal = new Anim
                animal.breathe("牛");
                animal.breathe("羊");
                animal.breathe("猪");
                animal.breathe("鱼");
        }
}
```

```
class Animal{
        public void breathe(String animal){
                System.out.println(animal+"呼吸空气");
        }

        public void breathe2(String animal){
                System.out.println(animal+"呼吸水");
        }
}

public  class Client{
        public static void main(String[] args){
                Animal animal = new Animal();
                animal.breathe("牛");
                animal.breathe("羊");
                animal.breathe("猪");
                animal.breathe2("鱼");
        }
}
```

组件设计原则-OCP

- 开放-封闭原则（OCP）
  - 对于扩展是开放的（OPEN FOR EXTENSION)
  - 对于更改是封闭的（CLOSED FOR MODIFICATION)



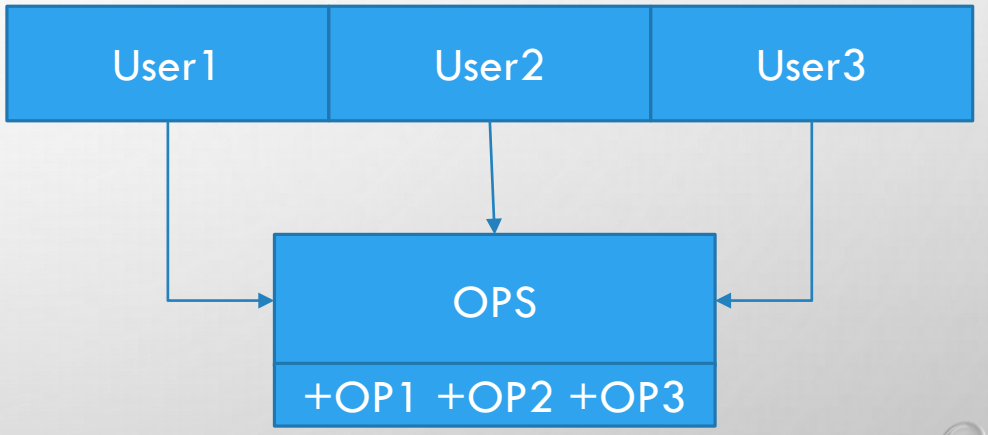


组件设计原则-LSP

- 子类型（SUBTYPE）必须能够替换掉它们的基类型（BASETYPE）
- 正方形是一个矩形
- IS-A 关系是就行为方式而言的， IS-A的含义过于宽泛以至于不能作为子类型的定义。子类型的定义是“可替换性”

ISP接口隔离原则

- 任何层次的软件设计如果依赖与不需要的东西，都会是有害的



DIP依赖反转原则

- 如果想要设计一个灵活的的系统，在源码层次的依赖关系中就应该应用抽象类型，而非具体实现

- C++多态
- C++20 CENCERPTS

概念

概念是具名的要求集合。概念的定义必须出现于命名空间作用域。

概念定义拥有以下形式

```
template < template-parameter-list >
concept concept-name = constraint-expression;
```

制约

制约是逻辑运算和运算数的序列，它指定模板实参上的要求。它们能出现在 *requires 表达式*（见后述）内，而且能直接作为概念之体。

制约有三种类型：

1) 合取
2) 析取
3) 原子制约

通过规范化运算数遵循下列顺序的逻辑与表达式，确定与声明关联的制约：

- 按出现顺序，对每个有制约模板形参引入的制约表达式；
- 模板形参列表后的 *requires* 子句中的制约表达式；
- 尾随 *requires* 子句中的制约表达式。

此顺序确定在检查是否满足时实例化制约的顺序。

有制约声明可以只用相同的语法形式重声明。不要求诊断。

```
template<Incrementable T>
void f(T) requires Decrementable<T>;

template<Incrementable T>
void f(T) requires Decrementable<T>; // OK ：重声明

template<typename T>
    requires Incrementable<T> && Decrementable<T>
void f(T); // 病式，不要求诊断

// 下列二个声明拥有不同的制约：
// 第一声明拥有 Incrementable<T> && Decrementable<T>
// 第二声明拥有 Decrementable<T> && Incrementable<T>
// 尽管它们逻辑上等价

template<Incrementable T>
void g() requires Decrementable<T>;

template<Decrementable T>
void g() requires Incrementable<T>; // 病式，不要求诊断
```

虚函数

- 慢的原因
  - 多了几条汇编指令（运行时得到对应类的函数的地址）
  - 影响CPU流水线
  - 编译器不能内联优化（仅在用父类引用或者指针调用时，不能内联）

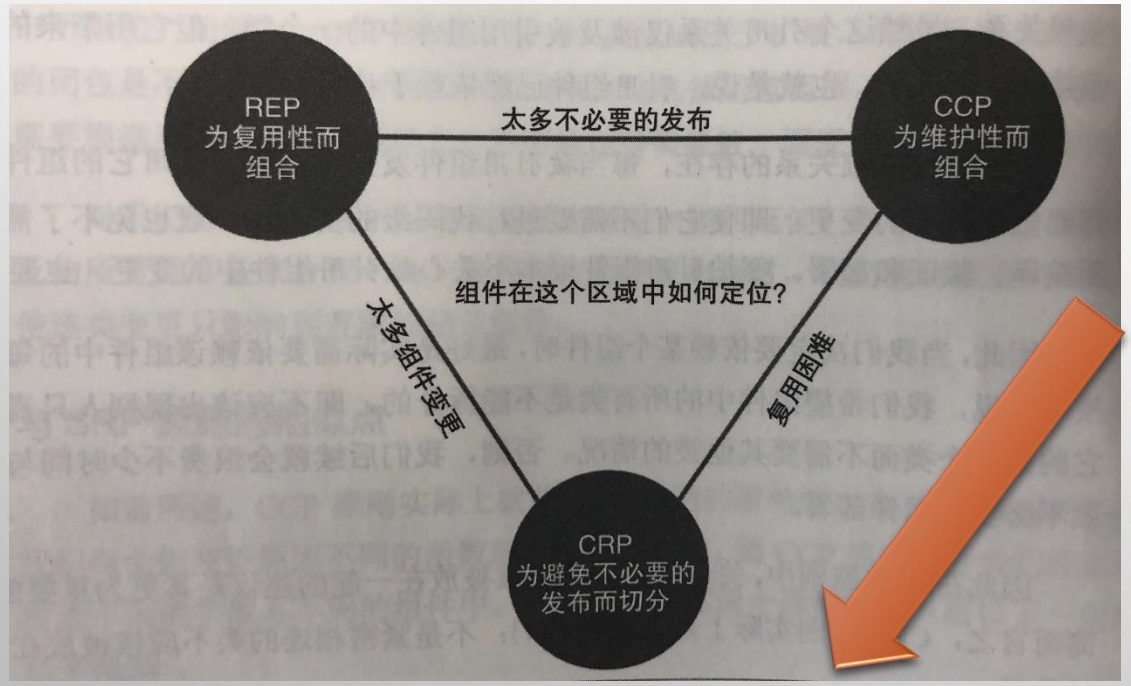
1层继承的测试结果：

	push_back	at
Vector	0.263s	0.04s
VirtualVector	0.331s	0.222s
倍数	1.25	5.55

6层继承的测试结果：

	push_back	at
Vector	0.262s	0.041s
VirtualVector	0.334s	0.223s
倍数	1.27	5.43

组件设计原则-目的



CCP(The Common Closure Principle)
共同闭包原则
将哪些会同时修改，并且修改目的相同的类放到同一个组件中，反之就应该放到不同组件中

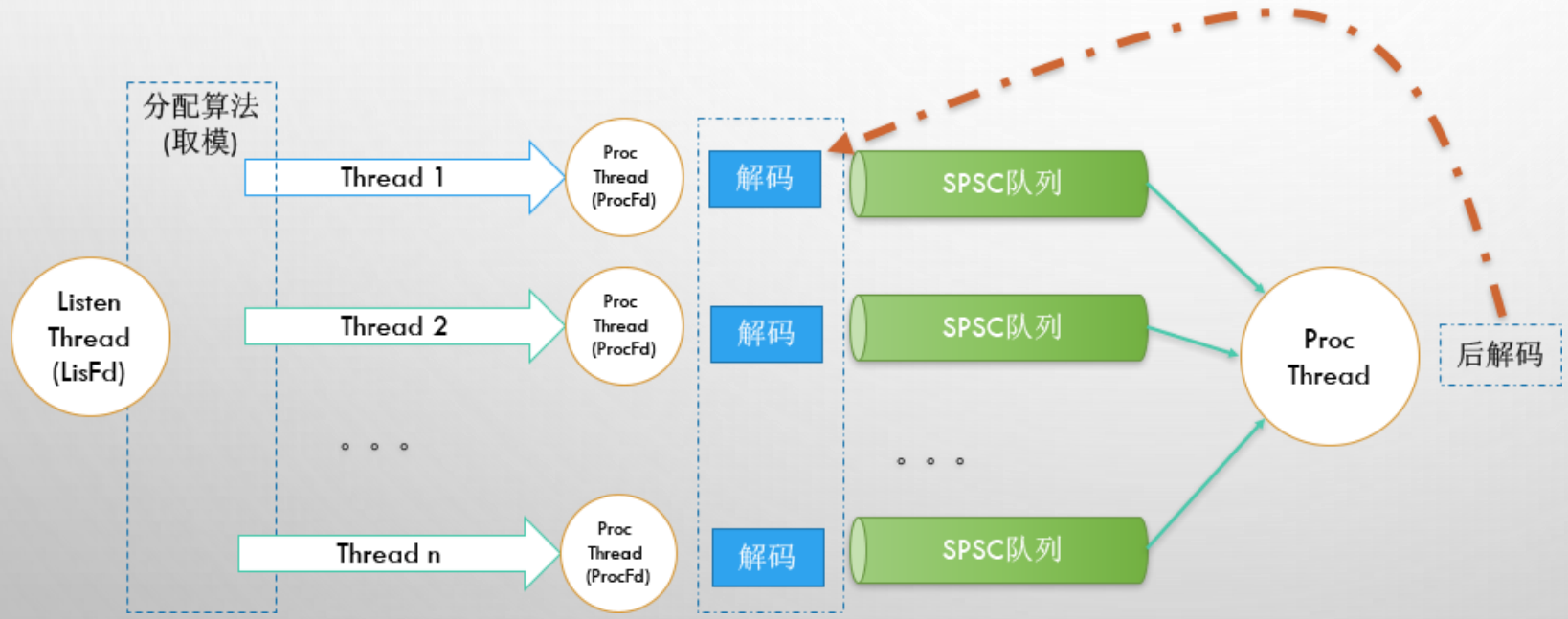
REP(The Reuse/Release Equivalence Principle)
复用/发布等同原则
软件复用的最小粒度应等同于其发布的最小粒度

CRP(The Common Reuse Principle)
共同复用原则
不要强迫一个组件的用户依赖他们不需要的东西

设计一个网络组件

- 需求：
  - 接口简单
  - 单线程应用
  - 多平台支持
  - 可靠性
  - 高性能

GNET组件设计



底层	平台	完成度
Epoll	Linux	高
IOCP	Windows	中
Asio	All	中
Kqueue	Mac	中
Libuv	All	中
Libev	Linux	中
Libevent	All	低

用户关心什么

- 直观问题
  - 创建连接对象
  - 释放连接对象
  - 事件：建立连接，收到数据，发送数据，连接关闭
- 潜在问题：
  - 事件回调是否同一个线程？
  - 发送是否线程安全？
  - 断包问题？
  - 性能？

多线程

- 多线程
  - CPU多核
  - 快 != 吞吐量
- 多进程（此组件不在此粒度）
  - 用户可以使用此组件做分布式系统

造芯片的家伙们正忙着生产那些大多数程序员不知道如何编程的多核CPU

发送数据

- 强制要求单线程？
- 内部句柄暴露给用户安全吗？Send(fd， buf,bufLen);

单线程逻辑层

- 应用层不需要复杂的线程同步
- 业务逻辑太多，同步数据过多，导致性能低下
- 容易出问题
- 组件集成还是应用实现？

多线程使用场景

- 状态无关业务：
  - 解码过程比较费CPU
  - 心跳数据
  - 转发数据包
- 有状态服务
  - 用户状态
  - 游戏交互逻辑等

N转1队列

- 并发技术
  - 互斥锁
  - 信号量
  - 条件变量
  - 原子操作
  - 内存序

Mutex 系列类(四种)

- std::mutex，最基本的 Mutex 类。
- std::recursive_mutex，递归 Mutex 类。
- std::time_mutex，定时 Mutex 类。
- std::recursive_timed_mutex，定时递归 Mutex 类。

Lock 类（两种)

- std::lock_guard，与 Mutex RAII 相关，方便线程对互斥量上锁。
- std::unique_lock，与 Mutex RAII 相关，方便线程对互斥量上锁，但提供了更好的上锁和解锁控制。

其他类型

- std::once_flag
- std::adopt_lock_t
- std::defer_lock_t
- std::try_to_lock_t

核心思想：减少冲突

简单实现

```cpp
#pragma once
#include <vector>
#include <functional>
#include <mutex>
template <class T>
class SimpleQueue
{
  public:
    //multi threads
    void push(const T &item){
      m_mutex.lock();
      m_data.emplace_back(std::move(item));
      m_mutex.unlock();
    }
    bool empty(){
        std::lock_guard<std::mutex> guard(m_mutex);
        return m_data.empty();
    }
    // single thread
    template <typename Processor>
      void process(Processor processor){
        {
         std::lock_guard<std::mutex> guard(m_mutex);
         if (m_cache.empty()) {
           m_data.swap(m_cache);
         }
        }
        for (m_cache : item )
        {
          processor(item);
        }
        m_cache.clear();
      }
  private:
    std::mutex  m_mutex;
    std::vector<T> m_data;
    std::vector<T> m_cache;
};
```

```cpp
// single thread
template <typename Processor>
  void process(Processor processor)
  {
    do{
      std::lock_guard<std::mutex> guard(m_mutex);
      if (m_cache.empty())
      {
        m_data.swap(m_cache);
      }
    }while(0);

    bool status = true;
    for (;m_index< m_cache.size(); m_index ++)
    {
      status = processor(m_cache[m_index]);
      if (!status)
      {
        break;
      }
    }

    if (status)
    {
      m_index = 0;
      m_cache.clear();
    }
  }
```

跨平台

- 平台差异
  - LINUX下EPOLL（LT/ET）
  - MAC下KQUEUE
  - WINDOWS下IOCP
  - LIBEV/LIBEVENT/LIBUV

接口设计-KISS原则

- KEEP IT SIMPLE & STUPID
- KISS 原则是指产品的设计越简单越好，任何没有必要的复杂都是需要避免的。
- **用户只关心他需要关心的**

Keep It Simple & Stupid；
Keep It Sweet & Simple；
Keep It Short & Simple；
Keep it Simple, Sweetheart；
Keep it Simple, Sherlock。

最容易的莫过于忙忙碌碌，最困难的莫过于卓有成效。

KISS ≠ 专业性

- 稳定性
  - 系统需要考虑极端情况下的处理
  - 负载平衡（LOAD BALANCING）是一种计算机技术，用来在多个计算机（计算机集群）、网络连接、CPU、磁盘驱动器或其他资源中分配负载，以达到最优化资源使用、最大化吞吐率、最小化响应时间、同时避免过载的目的。 使用带有负载平衡的多个服务器组件，取代单一的组件，可以通过冗余提高可靠性。负载平衡服务通常是由专用软件和硬件来完成。 主要作用是将大量作业合理地分摊到多个操作单元上进行执行，用于解决互联网架构中的高并发和高可用的问题。
- 高低水位：
  - 设置系统边界，避免局部崩溃，提升组件鲁棒性
- 木桶原则
  - 短板原则，即一个木桶装多少水，不是取决于最长的那块木板，而是取决于最短的那块木板。
  - BETTER IS BETTER

组件的劣势

- 能解决我的全部问题吗？
  - 区分不同消息类型？
  - 能实现N:M 线程模型吗？
  - 我不要发送缓存区可以吗？
- 能适合全部领域吗？
  - 我可以不用C++11吗？
  - 能跑在ANDROID/IOS平台？
  - 我的GO/PYTHON/NODEJS程序能使用吗？
- 我不会C++，能用吗？

组件只是尝试以合适的的方式解决特定领域的特定问题

组件只是一部分

- 网关需求：
  - 高并发网络处理
  - 数据转发
  - 会话管理
  - 广播处理
  - 多登陆处理
  - 。。。

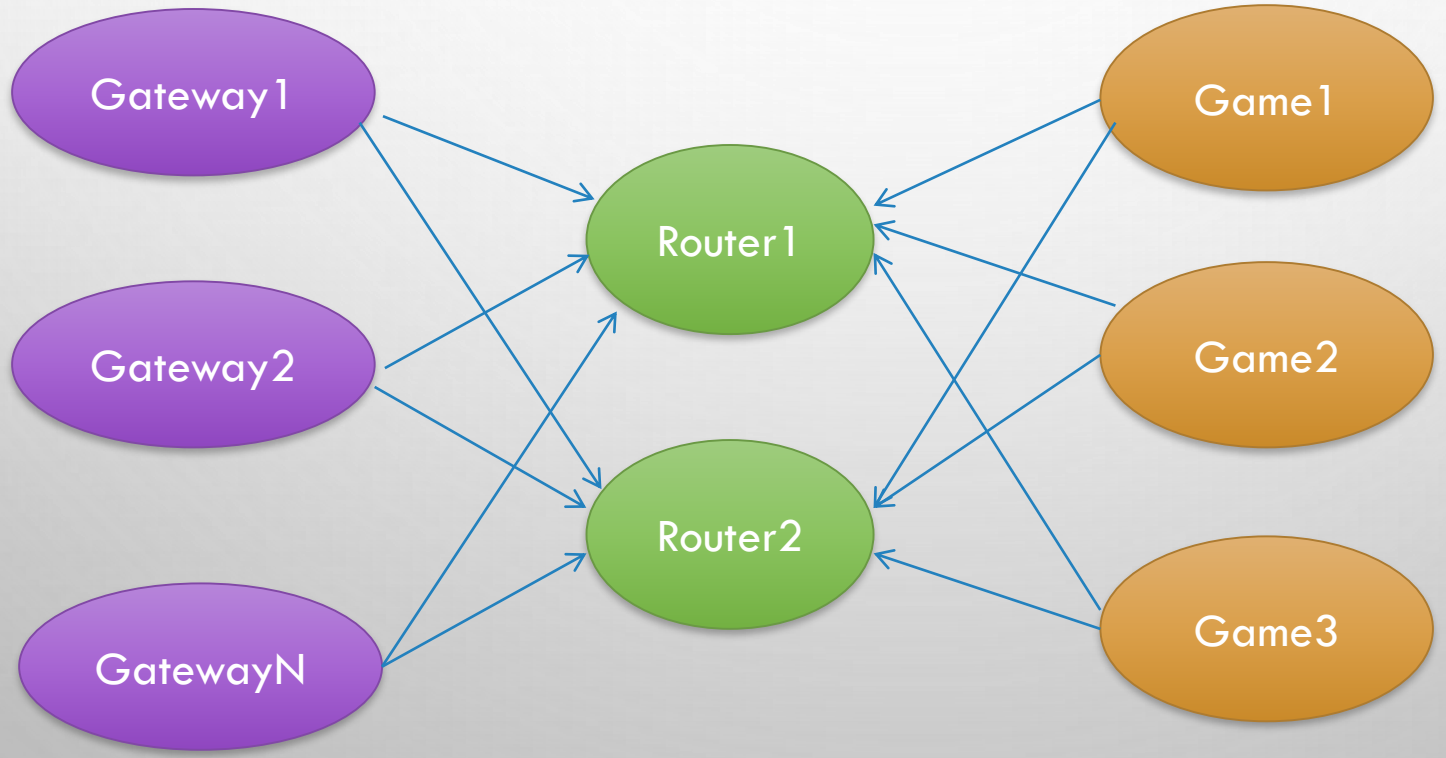
BUG无处不在

- 自身BUG
  - BUG无处不在
- 使用限制
  - 使用不当导致错误路径
- 风格不合
  - 萝卜白菜，各有所爱
- 就不想用
  - 。。。IT'S OK

组件介绍

GROUTER V1

解决服务器分布式设计难问题，通过分布式算法保证数据的一致性，解决AKB服务器分布式设计需求。



解决akb服务器分布式需求开发的分布式组件，
接入此组件后项目组只需关注上层逻辑，通过组件分发消息，简化逻辑编程。目前已接入使用

GROUTER特性

一致性 1

通过分布式锁实现对象的定位
Router缓存对象位置(二级缓存)
目标服务器不存活重新定位

可用性 2

多个对称Router形成的集群，集群化部署

高性能 3

10Wqps的转发, 800到1000的定位

GROUTER发展

目标：将grouter发展为分布式服务器核心组件，使游戏服务器架构设计更简单，更清晰

03 优化算法

02 将组件加入到核心服务器上

01 与游戏网关合并，减少消息转发

01、将Router和Gateway合并，减少一次消息转发

02、将分布式定位算法移植到Game上，支持更灵活的分布式程序开发

03、用multi raft协议实现分布式锁，替代etcd，提高现在对象定位的速度

GDB V2

- 使用简单：
  - 简化的FORMAT语法：{FMT}
  - 一句话SQL执行：

  - 简化V1版本底层依赖，使用更容易
- 大幅提升项目组数据库服务器开发效率

```
sql.select("uid", "name").from("user_info").get([](ResultSetPtr res){

});
```

XML2CPP

- 一键生成配置代码
  - 1. 配置模板文件 GATEWAY.XML.EXAMPLE
  - 2. 执行命令：XML2CPP --FILE=./CONFIG/GATEWAY.XML.EXAMPLE --OUT=./CONFIG

和

```xml
<!-- gateway.xml.example -->
<?xml version="1.0" enconding="UTF-8"?>
<ip>0.0.0.0</ip>
<port>3000</port>
<log>
  <name>test</name>
  <level>trace</level>
  <thread>1</thread>
</log>
<queues>
  <server ip="192.168.1.1" port="1234" />
  <server ip="192.168.1.2" port="1234" />
  <server ip="192.168.1.3" port="1234" />
</queues>
```

```cpp
// main.cpp
#include <iostream>
#include "gateway.xml.h"

int main() {
  config::gateway().Init("./config/gateway.xml");

  std::cout << config::gateway().ip().as_string() << std::endl;
  std::cout << config::gateway().port().as_uint() << std::endl;

  std::cout << config::gateway().log().name().as_string() << std::endl;
  std::cout << config::gateway().log().level().as_string() << std::endl;
  std::cout << config::gateway().log().thread().as_uint() << std::endl;

  for(auto& server : config::gateway().queues()) {
    std::cout << server.ip.as_string() << std::endl;
    std::cout << server.ip.as_uint() << std::endl;
  }
}
```

共享内存组件

特性：

1. **逻辑和数据分离，运行时数据更安全**
2. **基于共享内存的数据守护进程，保证服务器异常情况下快速拉起**
3. **头文件库，丰富数据结构支持，简单易用**

目前业界比较流行的提升服务器稳定性主流方案

运行时守护

在服务器异常时保证数据不丢失，服务器快速重启，玩家不掉线。

跨平台组件

支持windows/linux平台使用

丰富数据结构支持

封装了标准库常用组件，头文件库无依赖，方面使用

为什么需要微服务

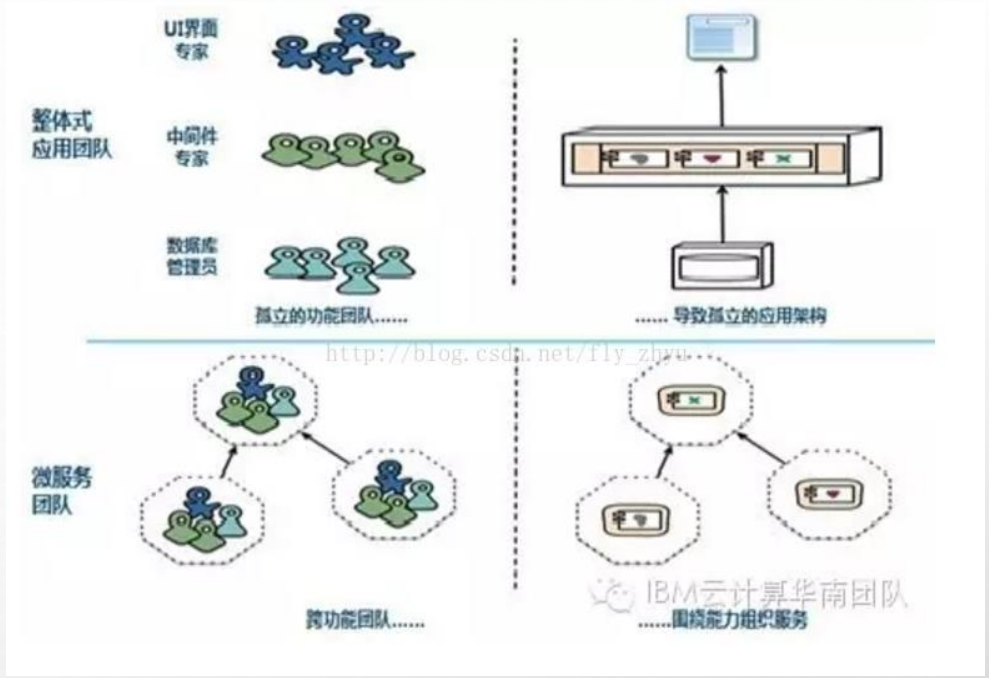
- 多语言架构：
  - 各类语言各有优劣，C/C++高性能，GO开发效率高
  - 硬件及软件发展推动服务器整体性能提升
  - 软件系统越来越大，代码越来越多，结构越来越复杂
  - 云化的发展，服务都通过云的方式提供
  - 人才的层次结构也反映了服务的分级实现
  - 更多的互联网技术，开源项目推动了专业化发展

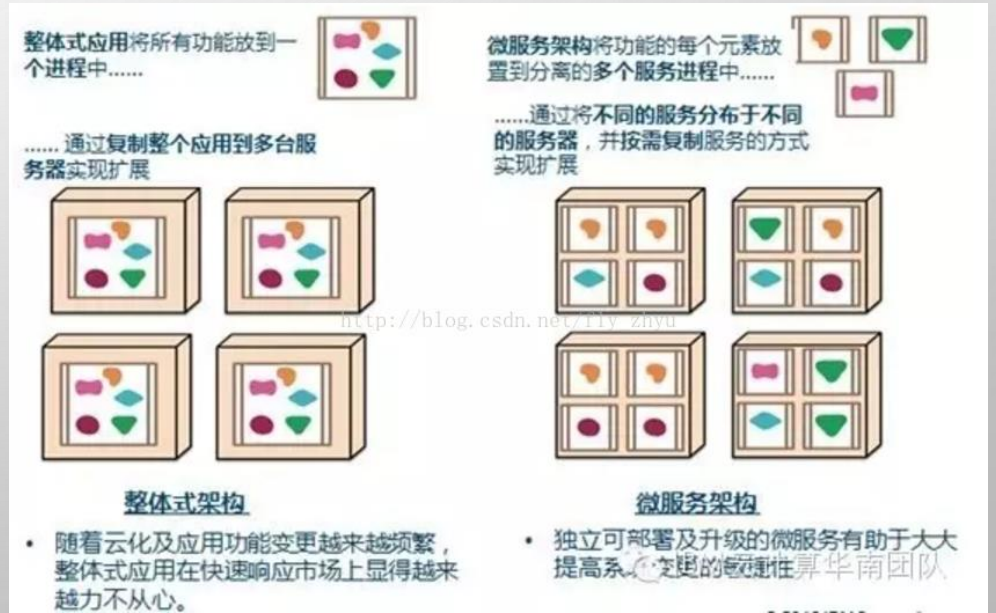
什么是微服务

- 微服务是一种**架构风格**，一个**大型复杂**软件应用由一个或多个微服务组成。系统中的各个微服务可被**独立部署**，各个微服务之间是**松耦合**的。每个微服务仅关注于完成一件任务并很好地完成该任务。在所有情况下，每个任务代表着一个小的**业务能力**。

微服务特点

- 通过服务实现应用的组件化(COMPONENTIZATIONVIA SERVICES)
  - 独立替换和升级的软件单元，在应用架构设计中通过将整体应用切分成可独立部署及升级的微服务方式进行组件化设计。
- 围绕业务能力组织服务(ORGANIZEDAROUND BUSINESS CAPABILITIES)
  - 微务架构采取以业务能力为出发点组织服务的策略，因此微服务团队的组织结构必须是跨功能的（如：既管应用，也管数据库）、强搭配的DEVOPS开发运维一体化团队，通常这些团队不会太大





微服务特点

- 产品而非项目模式(PRODUCTSNOT PROJECTS)
  - 传统的应用模式是一个团队以项目模式开发完整的应用，开发完成后就交付给运维团队负责维护；微服务架构则倡导一个团队应该如开发产品般负责一个“微服务”完整的生命周期，倡导“谁开发，谁运营”的开发运维一体化方法。
- 智能端点与管道扁平化(SMARTENDPOINTS AND DUMB PIPES)
  - 微服务架构主张将组件间通讯的相关业务逻辑/智能放在组件端点侧而非放在通讯组件中，通讯机制或组件应该尽量简单及松耦合。RESTFUL HTTP协议和仅提供消息路由功能的轻量级异步机制是微服务架构中最常用的通讯机制。

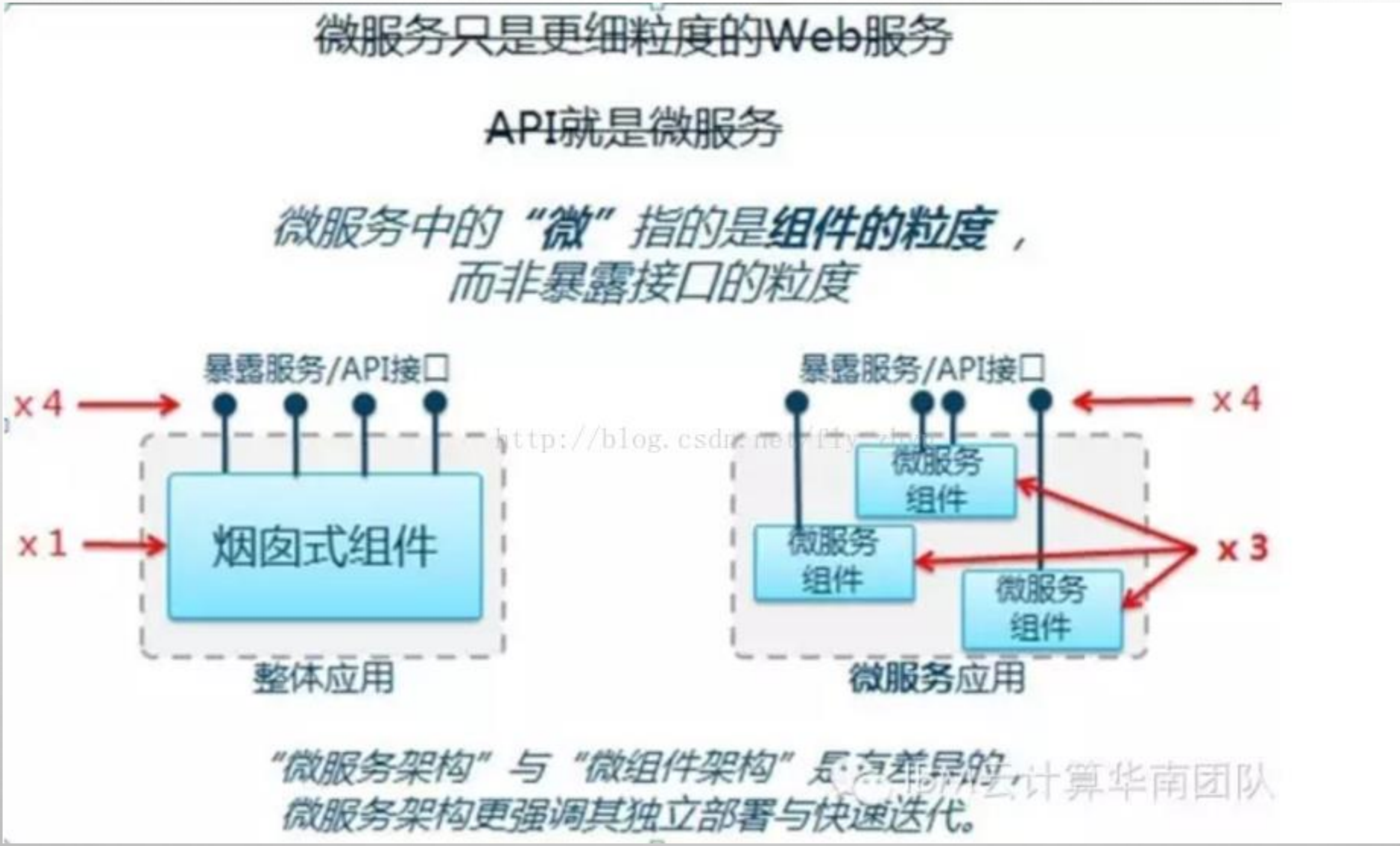
微服务特点

- “去中心化”治理(DECENTRALIZEDGOVERNANCE)
  - 整体式应用往往倾向于采用单一技术平台，微服务架构则鼓励使用合适的工具完成各自的任务，每个微服务可以考虑选用最佳工具完成(如不同的编程语言)。微服务的技术标准倾向于寻找其他开发者已成功验证解决类似问题的技术。
- “去中心化”数据管理(DECENTRALIZEDDATA MANAGEMENT)
  - 微服务架构倡导采用多样性持久化(POLYGLOTPERSISTENCE)的方法，让每个微服务管理其自有数据库，并允许不同微服务采用不同的数据持久化技术。
- 基础设施自动化(INFRASTRUCTUREAUTOMATION)
  - 云化及自动化部署等技术极大地降低了微服务构建、部署和运维的难度，通过应用持续集成和持续交付等方法有助于达到加速推出市场的目的。
- 故障处理设计(DESIGNFOR FAILURE)
  - 微服务架构所带来的一个后果是必须考虑每个服务的失败容错机制。因此，微服务非常重视建立架构及业务相关指标的实时监控和日志机制。

微服务特点

- 演进式的设计(EVOLUTIONARYDESIGN)
  - 服务应用更注重快速更新，因此系统的计会随时间不断变化及演进。微服务的设计受业务功能的生命周期等因素影响。如某应用是整体式应用，但逐渐朝微应用架构方向演进，整体式应用仍是核心，但新功能将使用应用所提供的API构建。再如在某微服务应用中，可替代性模块化设计的基本原则，在实施后发现某两个微服务经常必须同时更新，则这很可能意味着应将其合并为一个微服务。
-

微服务的一些常见误解



~~微服务只是更细粒度的Web服务~~
~~API就是微服务~~
微服务中的“微”指的是组件的粒度，
而非暴露接口的粒度
暴露服务/API接口
x 4
x 1
烟囱式组件
整体应用
暴露服务/API接口
x 4
微服务组件
微服务组件
微服务组件
x 3
微服务应用
“微服务架构”与“微组件架构”是有差异的，
微服务架构更强调其独立部署与快速迭代。

游戏微服务化

- 要求
  - 业务需求更具体
  - 协议更复杂（二进制/HTTP协议）
  - 可扩展性要求
  - 性能要求
  - 独立性
  - 数据交互频繁
  - 运维职责

微服务

敏感字服务

项目	状态	版本
辐射 Online 服务器	接入完成	3.2
辐射 Online 客户端	接入完成	2.1.1
传世端游 服务器	接入完成，已经上线	4.1
血族手游 服务器	接入完成，已经上线	4.1
血族手游 客户端	接入完成，已经上线	4.1
光明与勇士 服务器	接入完成，已经上线	4.3

敏感字组件是公司强制要接入的服务器组件，目前服务器引擎部自研了敏感字组件，支持服务器和客户端使用，已经上线多个项目组。敏感字支持utf8多语言环境。目前上线半年来，比较稳定，正在纳入公司服务器验收标准

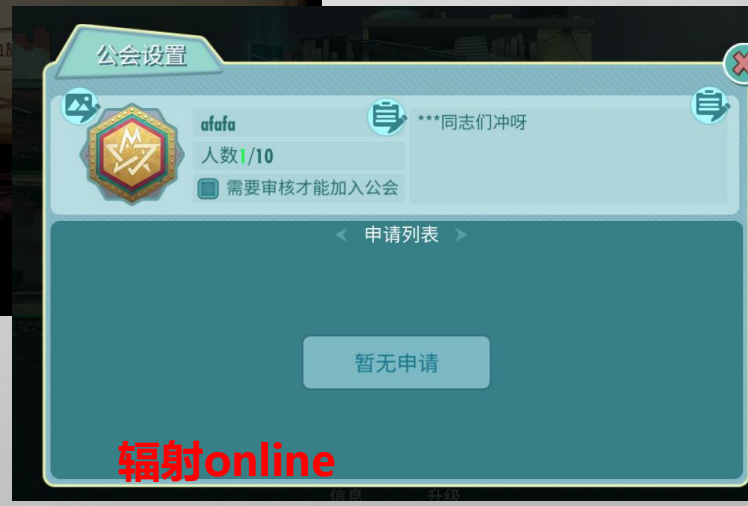
收入敏感字：12856

平台	版本	时间
Unity 5.6.2f1	2.1.1	2月12日2018年
VS2005 Win32/Win64	4.1	7月27日2018年
VS2010 Win32/Win64	4.1	7月27日2018年
VS2012 Win32/Win64	4.1	7月27日2018年
VS2013 Win32/Win64	4.1	7月27日2018年
VS2017 Win32/Win64	4.1	7月27日2018年
CentOS 7.4.1708 x86_64	4.1	7月27日2018年

敏感字接入游戏

光明勇士

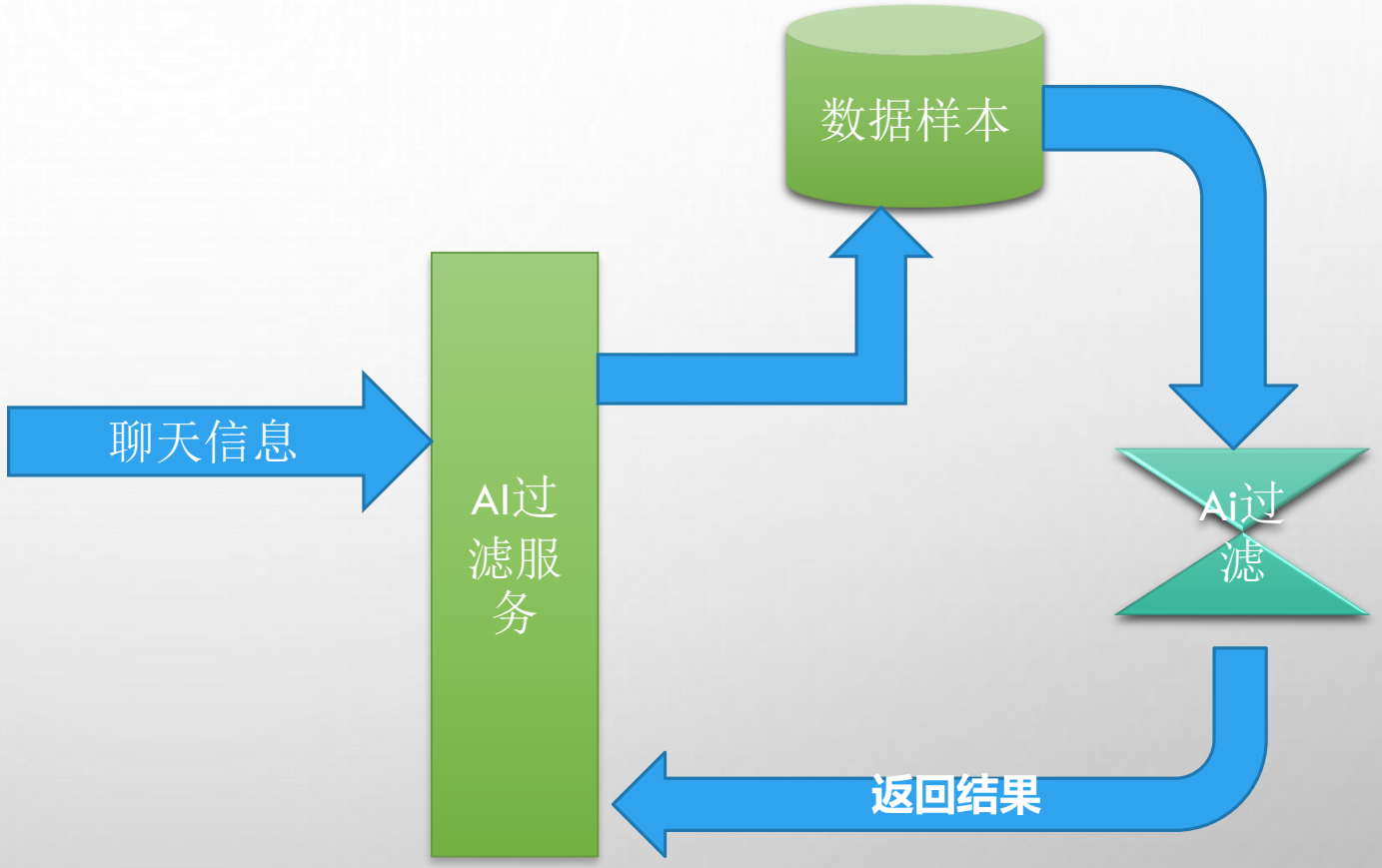
血族



辐射online

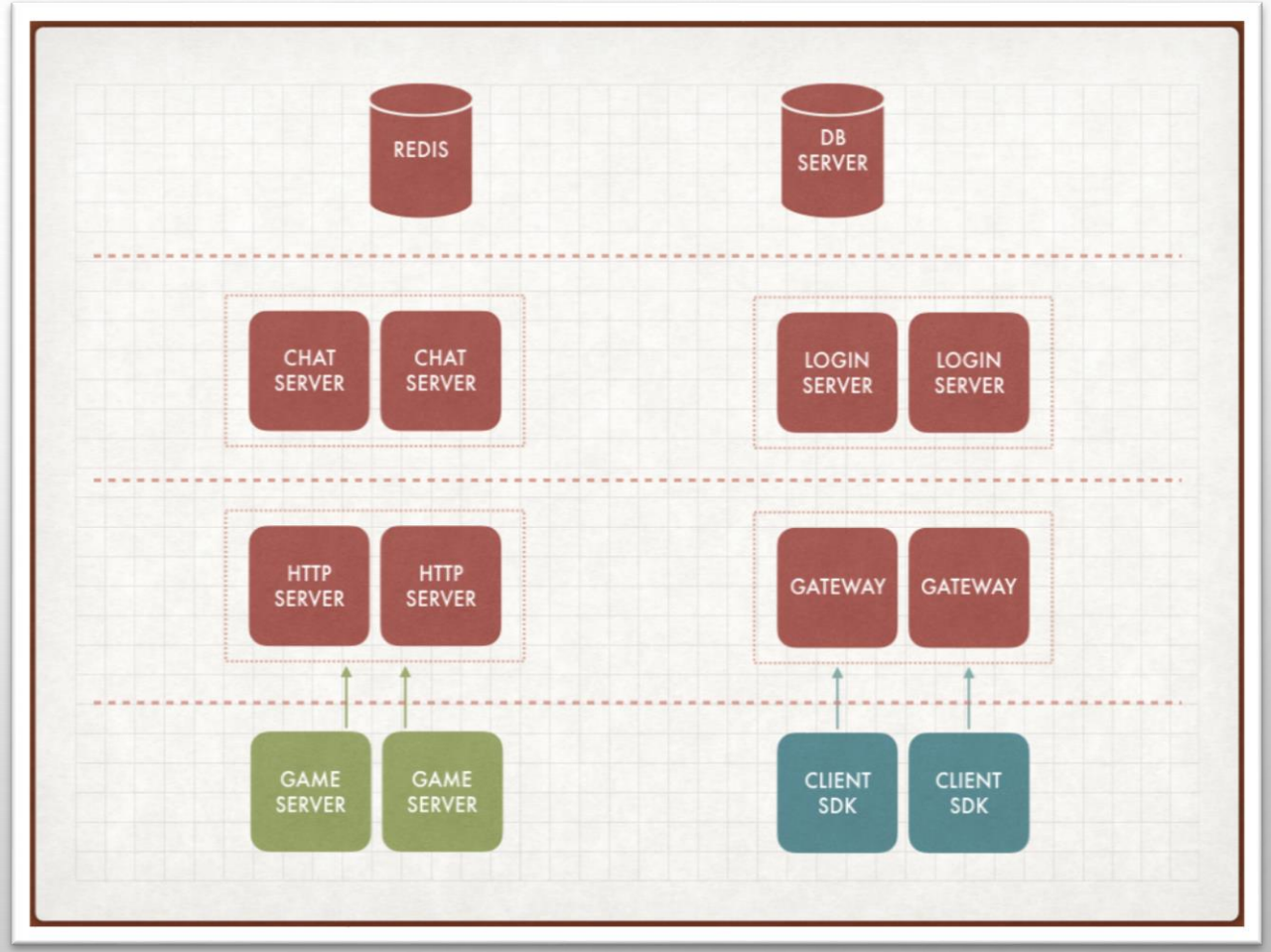
AI过滤

- 基于机器学习的智能过滤
- 屏蔽乱发广告



聊天微服务

- ✓敏感词过滤
- ✓万人以上的大群
- ✓离线消息
- ✓跨服聊天
- ✓消息广播
- ✓消息可靠性保证
- ✓数据加密



聊天服务

Performance

服务性能经过大量测试优化工作，
单台服务能支撑6万人群广播

Versatility

通用型服务器微服务框架，完成游戏内不同频道的聊天需求，业务上比较通用，适合公司游戏项目使用

Stability

服务器经过神无月上线半年多来，已经非常稳定，目前未出现宕机情况。

Distributed

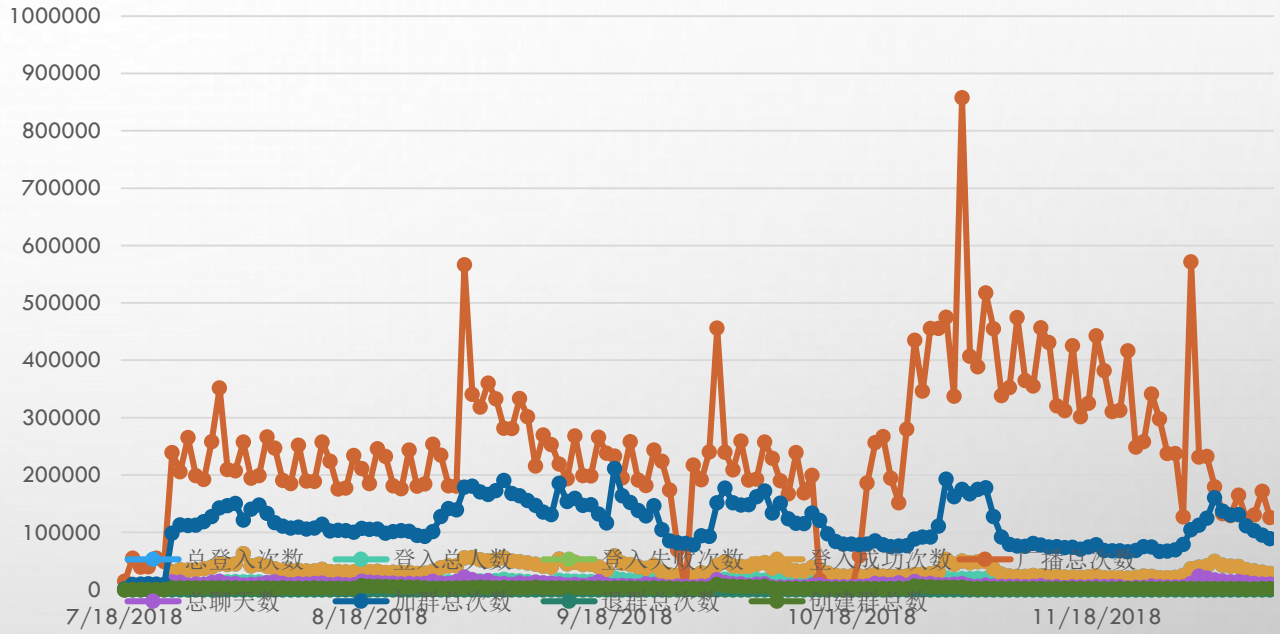
聊天服务也采用了服务器设计中常用的分层方案，支持横向扩展，并支持超大群或游戏大世界内的广播需求

项目	状态	版本
神无月，国服	接入完成，已经上线	1.0
神无月，港澳台	正式环境已经部署，还没上线	1.0
神无月，韩国	正式环境已经部署，还没上线	1.0
神无月，日本	正式环境已经部署，还没上线	1.0
神无月，东南亚	正式环境已经部署，还没上线	1.0
AKB	核心功能接入完成，部分定制需求还在调试	2.0

聊天服务统计

单台服务器：

最高日广播100万
日均登陆用户1万
日均登陆次数3.4万，登陆成功率100%

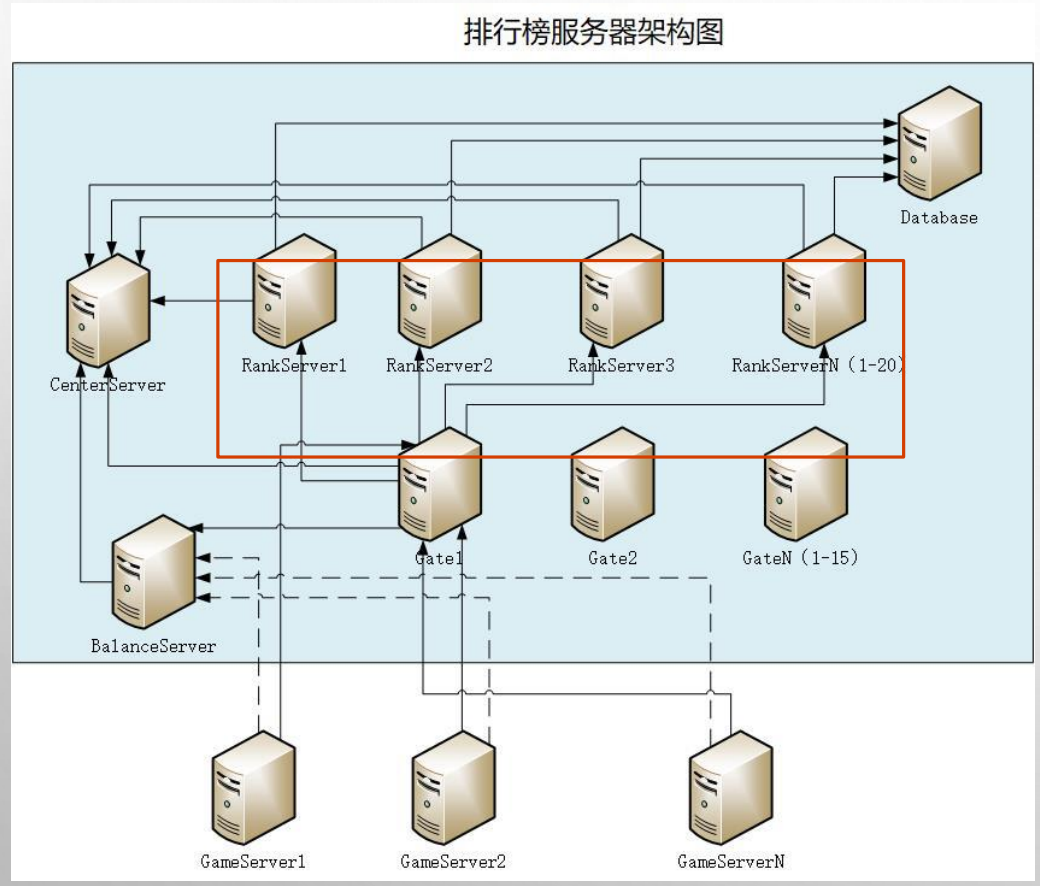


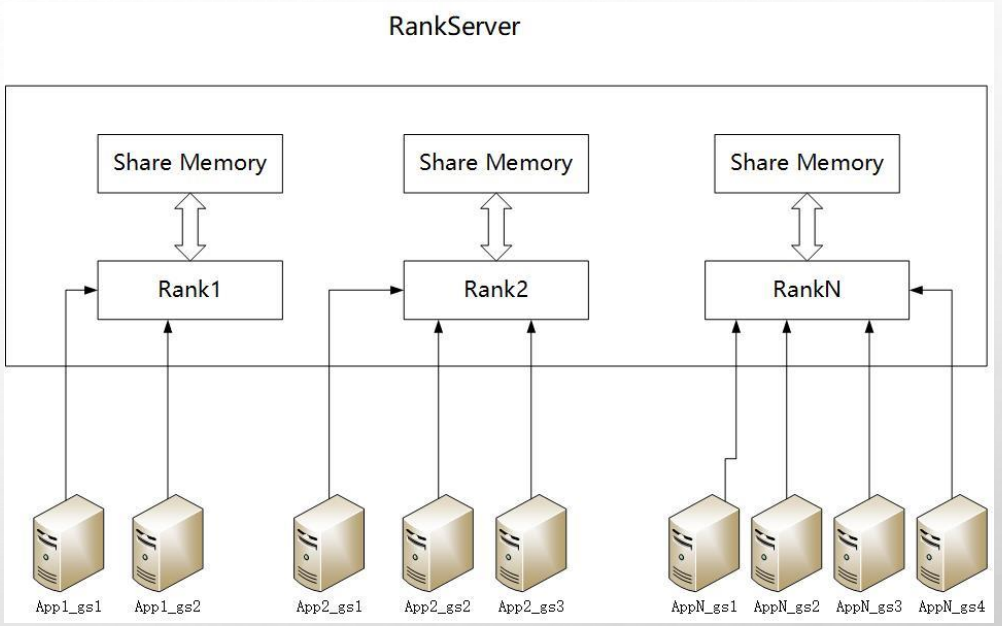
排行榜服务

设计目标

1. 支持海量级的排行数量，百万级数据排行
2. 支持大并发数据访问
3. 支持弹性动态扩容，支持横向扩展
4. 高可用，采用共享内存等先进技术提升服务器可用性

排行榜服务器架构





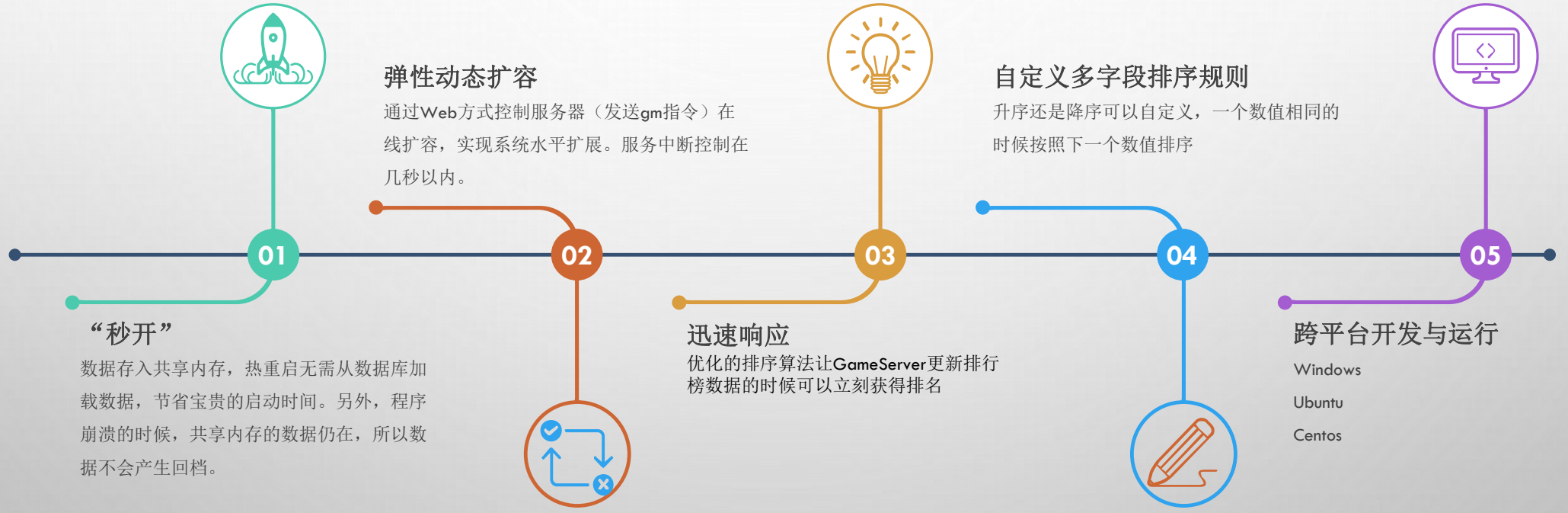
共享内存技术保障

1. 零宕机风险，

2. 毫秒级快速恢复

3. 服务器恢复数据不丢失

排行榜服务特性



01

“秒开”

数据存入共享内存，热重启无需从数据库加载数据，节省宝贵的启动时间。另外，程序崩溃的时候，共享内存的数据仍在，所以数据不会产生回档。

02

弹性动态扩容

通过Web方式控制服务器（发送gm指令）在线扩容，实现系统水平扩展。服务中断控制在几秒以内。

03

迅速响应

优化的排序算法让GameServer更新排行榜数据的时候可以立刻获得排名

04

自定义多字段排序规则

升序还是降序可以自定义，一个数值相同的时候按照下一个数值排序

05

跨平台开发与运行

Windows

Ubuntu

Centos

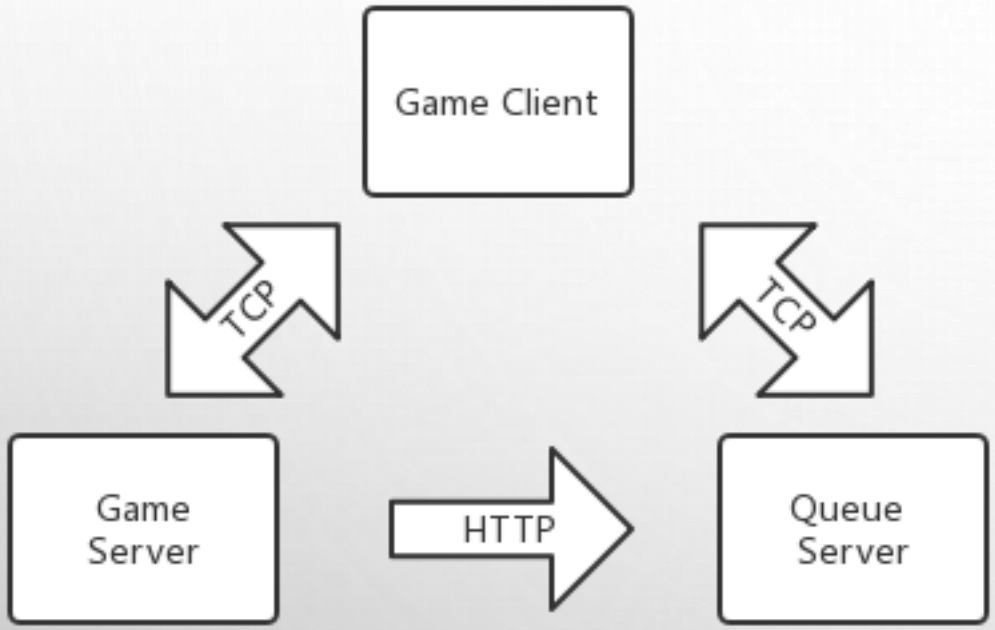
排行榜测试

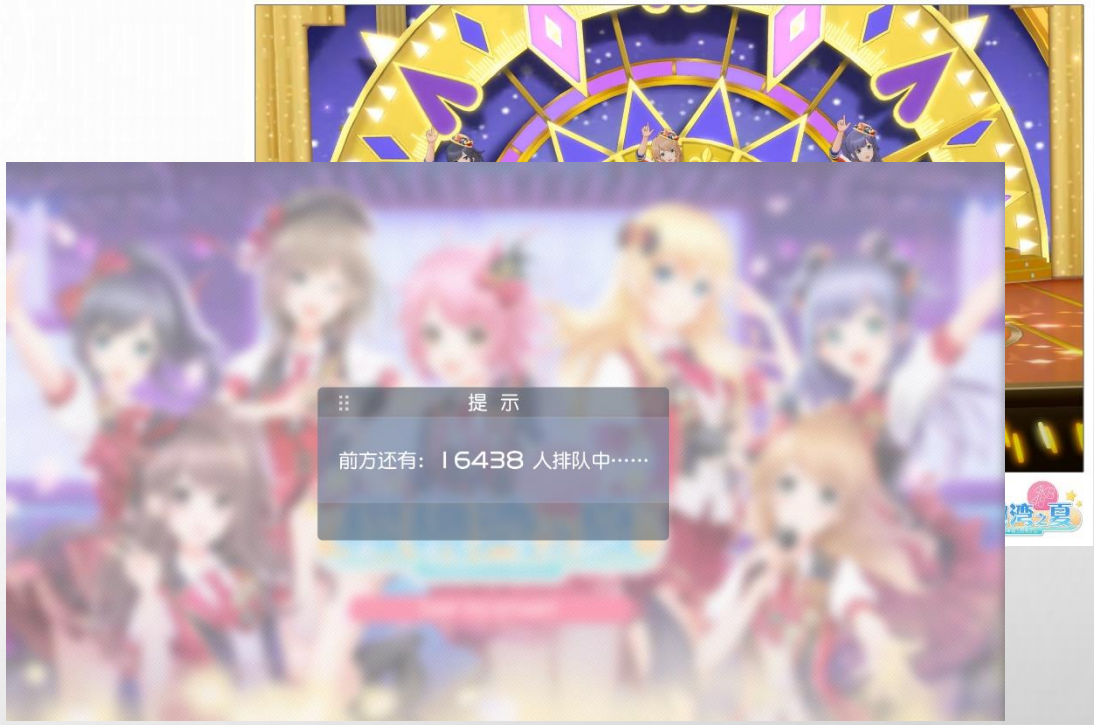
榜单大小 / 前列大小	500	1000	3000	5000	10000	30000	50000	100000
50	5500	5450	4900	4500	4300	1600	1000	620
100	4800	4650	4200	4000	3600	1500	950	560
300	2750	2700	2650	2500	2250	1400	900	540
500	2150	2200	2100	1950	1850	1200	750	480
1000		1450	1350	1200	1100	900	600	420
2000			850	800	750	520	490	400

- 这部分测试模拟真实使用环境。我们设计80%的操作为查询操作，20%的操作为更新操作。服务器同时处理两种请求，并记录测试结果。在本测试中，设定了每组不同数量的条目有不同的前列大小。测试结果如下：

- 参考查询和更新操作的测试结果，初步验证混合测试的结果是正确的。我们认为单次响应在1MS左右时，判定服务器可以正常工作。由测试数据表可以看出，在榜单3万人，前列大小1000的情况下，响应性能为900次/S。如果前列大小缩小为300，那么榜单可以支撑到5万人。

排队服务





游戏服务器 → 排队服务器

游戏服务器策略性的汇报服务器当前负载情况，用来给排队服务作为排队算法的依据。

游戏客户端 → 排队服务器

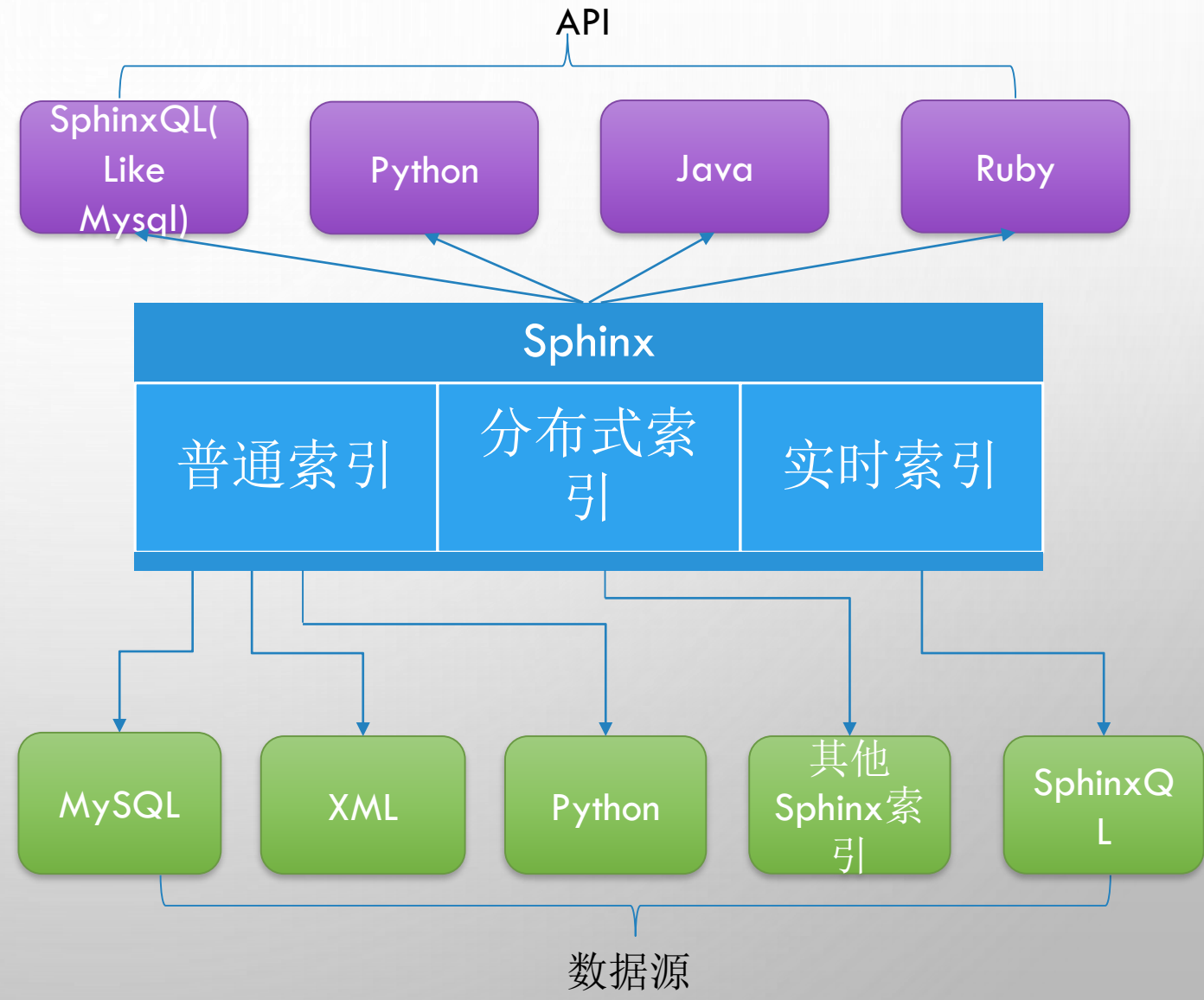
游戏客户端完成登陆身份验证后，先连接排队服务器，如果需要排队，则等待，待拿到票据后连接游戏服务器正常登陆。

队列类型：

1. 不用排队的队列
2. 优先级队列
3. 普通队列

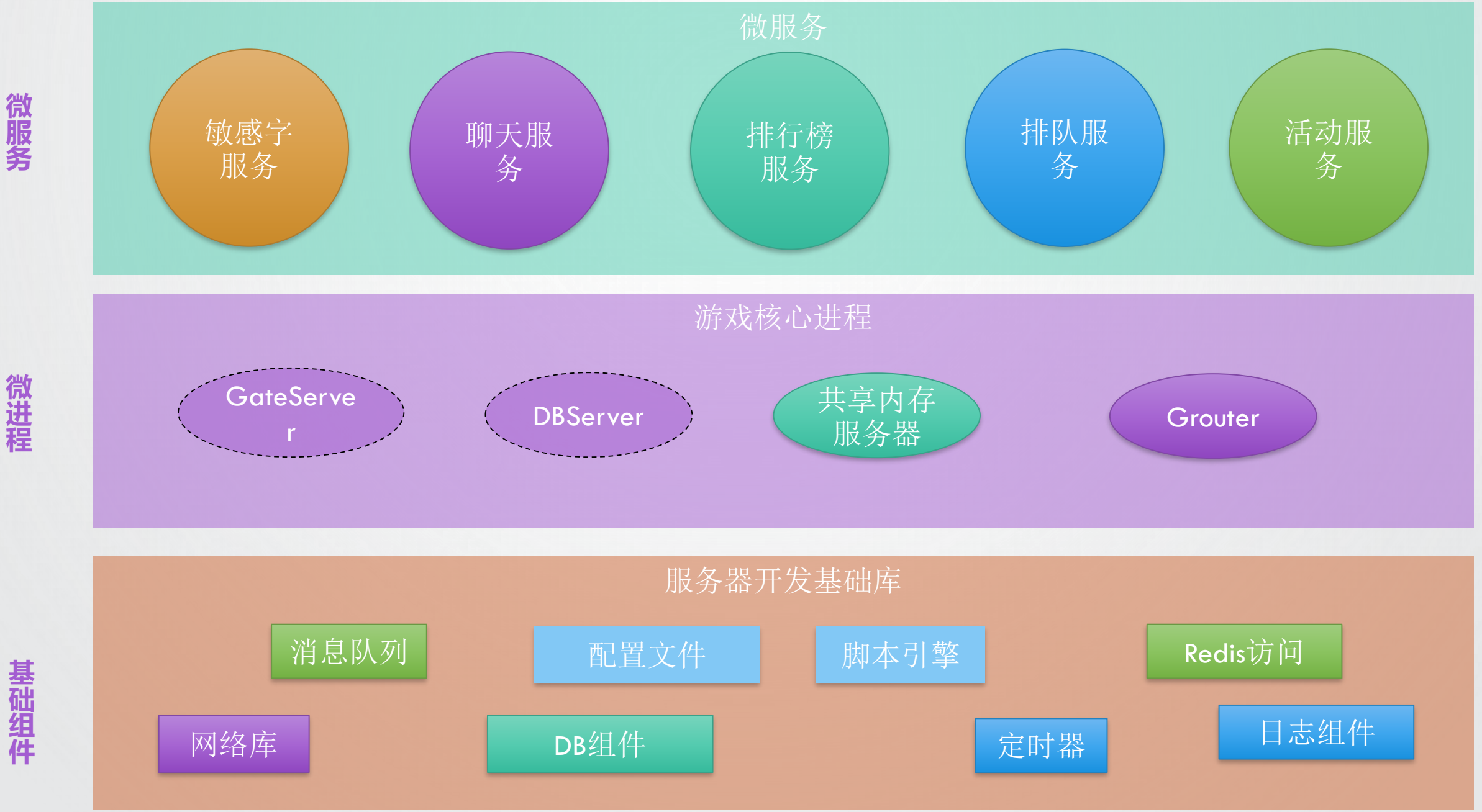
搜索服务

- 模糊查询
- 基于MYSQL数据库
- 实时索引
- 中文分词



文件服务器

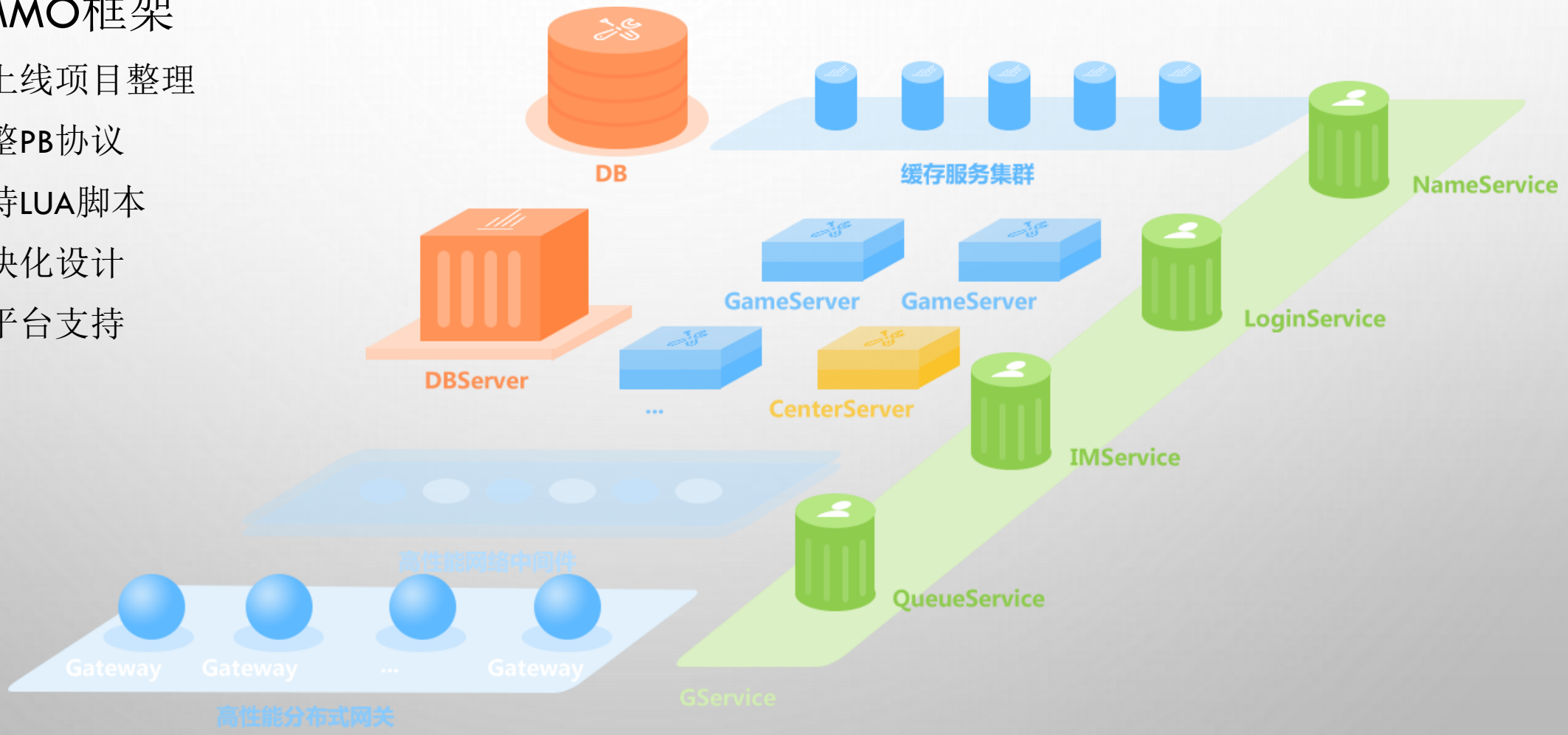
文件服务器是提供给akb的供客户端上传文件的服务器，支持单文件或多文件上传，同时支持上传到国内主流的腾讯云/阿里云对象存储服务。
而且增加了管理后台，方面对文件的增删查等操作，并提供了简单的用户管理


微服务
敏感字服务
聊天服务
排行榜服务
排队服务
活动服务
游戏核心进程
GateServer
DBServer
共享内存服务器
Grouter
服务器开发基础库
消息队列
配置文件
脚本引擎
Redis访问
网络库
DB组件
定时器
日志组件
微服务
微进程
基础组件

框架

MMO框架

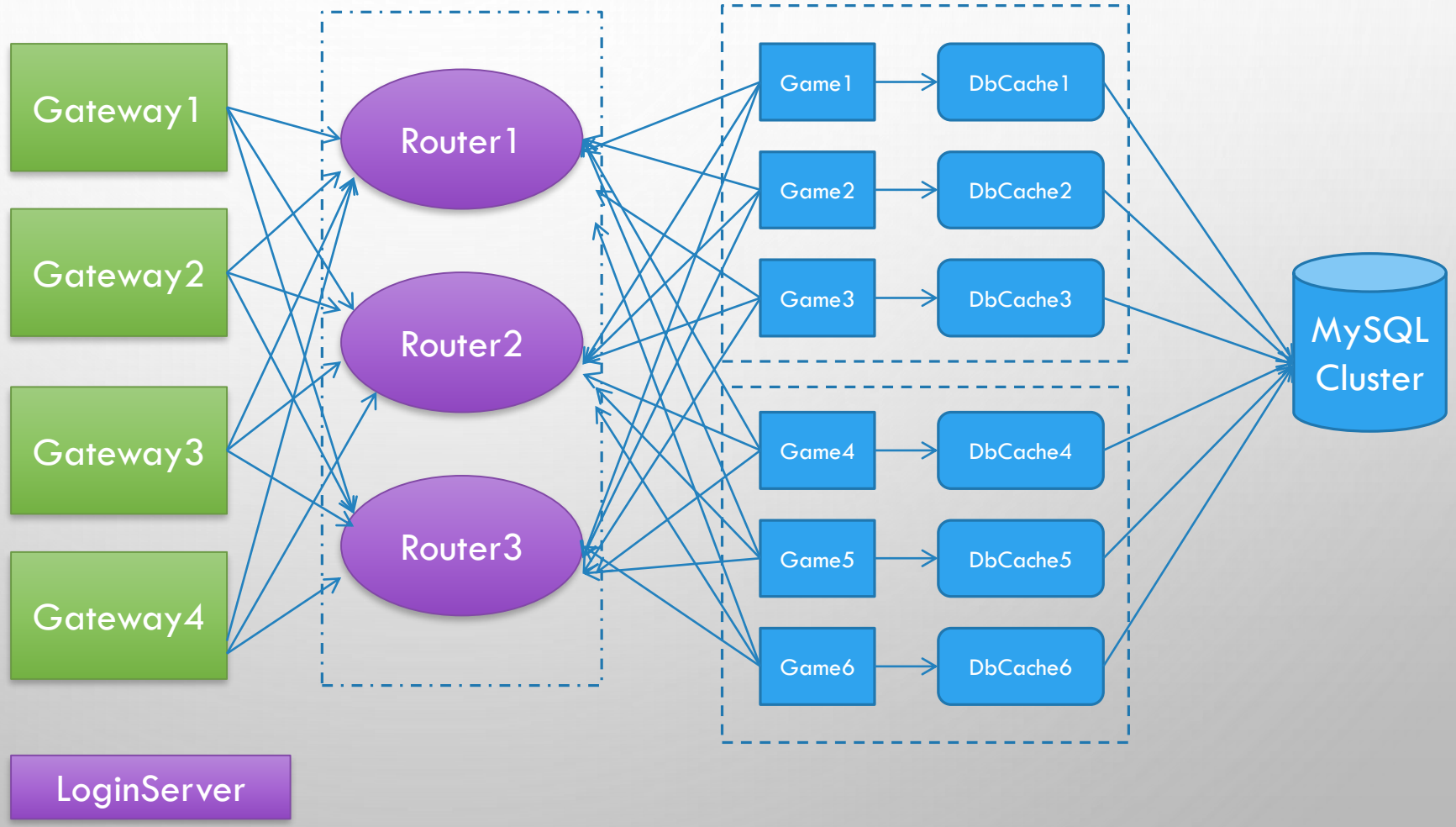
- 完整MMO框架
  - 从上线项目整理
  - 完整PB协议
  - 支持LUA脚本
  - 模块化设计
  - 跨平台支持



MMO功能

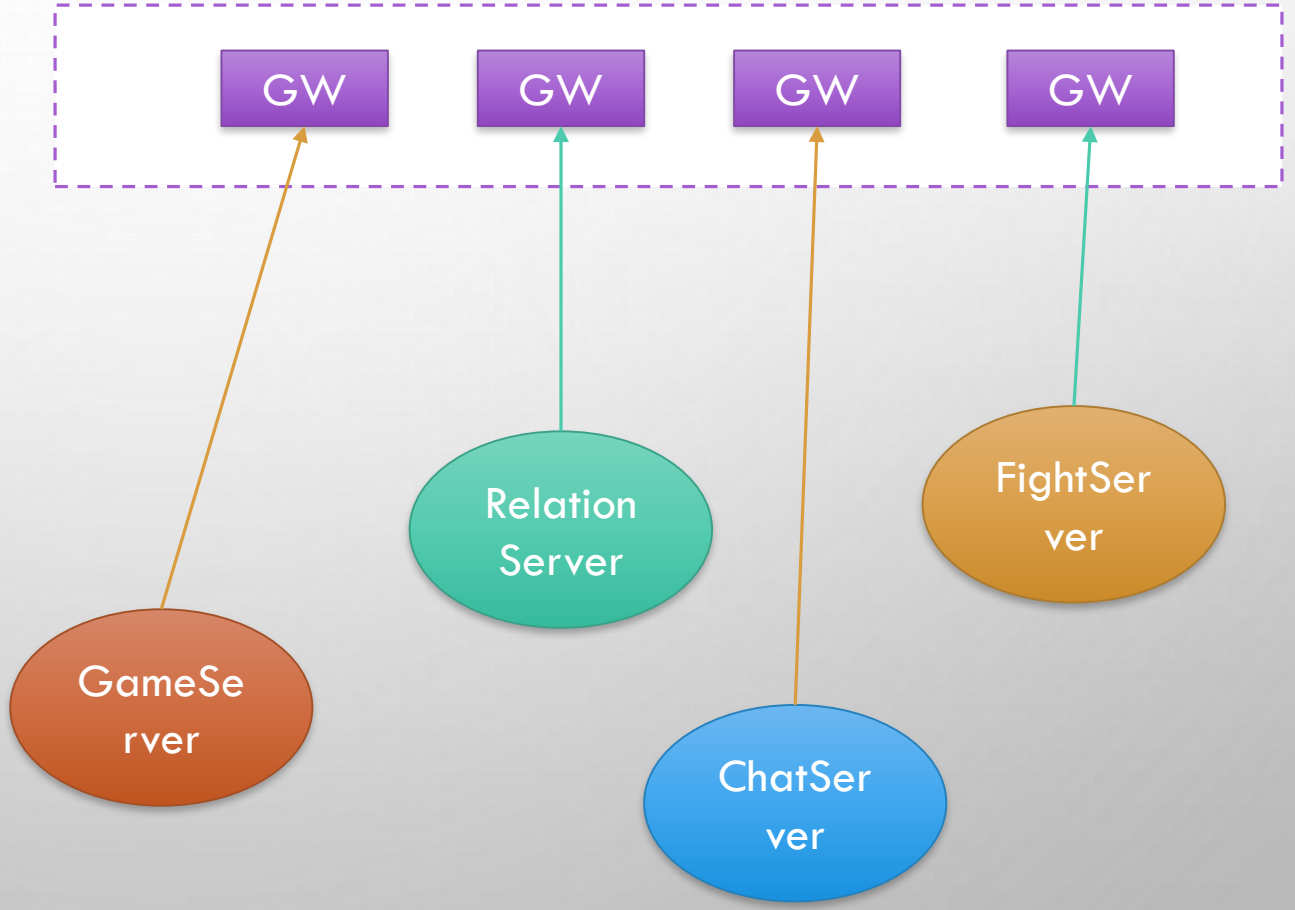
通服框架

全球通服
横向可扩展
无单点



统一游戏网关

- 支持大量网络连接
- 支持分布式
- 有状态会话管理
- 多协议（TCP/UDP/WEBSOCKET/HTTP）
- 安全加密
- 大群广播
- 自动路由
- 横向可扩展
- 分布式事务
- 负载均衡
- 业务无关性
- TOKEN认证



分布式编译特点

加速编译速度（根据机器数成本加速）

集中部署分布式机器，资源最大化利用

支持多平台
vs2015/2017/linux等

多主流平台支持，一键操作，方便使用

开发了一键安装脚本简单易用，开发机一键配置

开源项目，无需额外费用，达到和商业软件的易用性

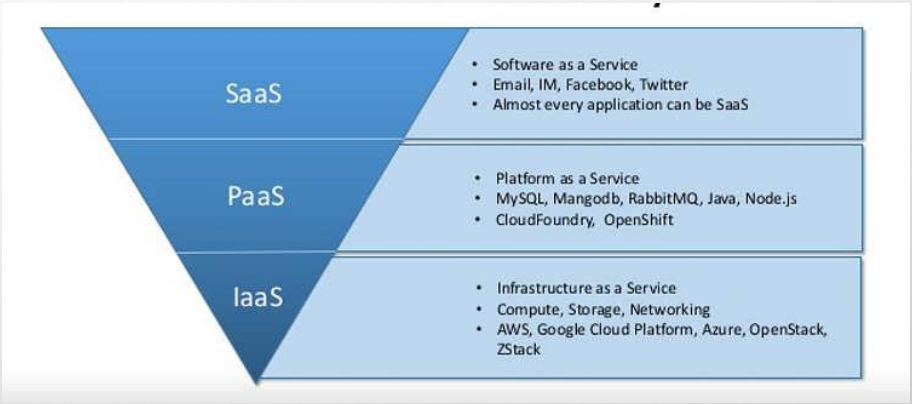
一键脚本安装

基于FastBuild开源实现（无法律风险）

谢谢！

PAAS /IAAS /SAAS

- IAAS：基础设施服务，INFRASTRUCTURE-AS-A-SERVICE
  - **SAAS 是软件的开发、管理、部署都交给第三方，不需要关心技术问题，可以拿来即用。**
- PAAS：平台服务，PLATFORM-AS-A-SERVICE
  - **抽象掉了硬件和操作系统细节，可以无缝地扩展（SCALING）。开发者只需要关注自己的业务逻辑，不需要关注底层。**
- SAAS：软件服务，SOFTWARE-AS-A-SERVICE
  - **是云服务的最底层，主要提供一些基础资源。**



ECS VS ACTOR

- ECS模型
  - 通过**组合**而不是继承的方法来进行实体的构建，一个实体，就是一群组件的聚合，一个实体指的是存在于你的游戏世界中的物体。实体在代码上就是一个组件的列表
- ACTORS
  - 一个ACTOR指的是一个最基本的**计算单元**。它能接收一个消息并且基于其执行计算。
  - ACTORS一大重要特征在于ACTORS之间相互隔离，它们并不互相共享内存。这点区别于上述的对象