腾讯游戏学院
TENCENT INSTITUTE OF GAMES

成就游戏梦想！

2018

腾讯移动游戏
技术评审标准与实践案例

序言

关于手册

进入2018年，中国移动游戏市场增速放缓，竞争愈发激烈；同时现象级手游层出不穷，手游精品化趋势更加明显，对移动游戏研发质量也提出了更高的要求。

在此背景下，腾讯游戏学院汇集腾讯互娱技术团队多年积累的技术评审标准和项目实践经验，系统化整理成册，正式发布《2018腾讯移动游戏技术评审标准与实践案例》手册，开放分享给游戏从业者。

本手册分为技术评审标准和项目实践案例2部分，通过6大维度技术评审标准、15篇腾讯游戏项目实践案例，深度剖析腾讯互娱技术团队研发思路和经验，帮助游戏开发者了解腾讯移动游戏的技术标准要点和项目优化经验，助力提升游戏开发者的研发效率和游戏品质。

致谢

感谢百忙之中参与手册内容写作的所有作者，每篇文章都意义深刻。

感谢以下的技术专家，协助进行文章筛选审阅：

叶劲峰、高炼、薛鹏、安柏霖、梁本志、代宜君、李强、郭智、陈月璇、谢文、丁博、段军寅、邓鹏、孟昭俊、刘洋、陈智伟、邹楠、罗亮之

感谢腾讯互动娱乐事业群-研发效能部-效能协同中心的同事持续投入，协助制作手册内容。

关于腾讯游戏学院

腾讯游戏学院由腾讯互动娱乐发起，致力于打造游戏知识分享和交流平台，专注建设游戏职业培训和发展体系。学院通过提供游戏类专业培训课程、建设游戏职业发展通道、开展丰富的校园活动及行业活动，帮助在校生和游戏从业者提升职业竞争力，成就游戏创想梦。

更多游戏技术资讯与干货，请关注游戏学院公众号

CONTENTS 目录 Contents

技术标准

实践案例

腾讯移动游戏技术评审标准（2018版）

技术维度	评审内容
客户端性能	1、内存消耗：针对不同档次客户端配置，实际占用的物理内存消耗合理； 2、客户端帧率：在默认画质配置下，核心游戏场景FPS无明显波动，稳定在一定数值以上； 3、CPU占用：不同游戏场景下，CPU占用合理； 4、流量消耗：对于分局的游戏场景，以单局消耗流量作为判断标准；对于不分局游戏场景或流量与局时有关的场景，以一定时间内的流量消耗为判断标准； 5、APK大小：安装包过大建议使用CDN资源。
弱网络状况	正确处理弱网络状况，不能出现以下现象： 1、游戏中收支不等、客户端卡死/崩溃等异常情况； 2、游戏核心功能（如登录、单局、支付等）有导致游戏无法正常进行的UI、交互问题； 3、损害玩家利益或可被玩家额外获利的问题； 4、需要有合理的重连机制，避免每次重连都返回到登录界面。
适配兼容性	1、机型适配是否符合要求： TOP100基线以上的机器适配，适配不通过的机器其用户占比不超过2%； 2、crash率是否达标： 用户crash率必须低于3%，次数crash率必须低于5%。iOS crash无遗留问题； 用户crash率=crash影响设备数/联网设备数； 次数crash率=crash次数/登录次数。

服务器性能	1、针对服务器的技术规范要求，比如：容灾处理、过载保护要求、防雪崩机制等，全部测试通过； 2、单机的综合场景测试下，必须满足性能基线要求，并且各事务90%响应时间<1秒，各事务成功率>99.9%； 3、稳定性测试（服务进程无重启，无内存泄漏）运行10小时以上，各事务成功率>99.9%。
安全防护	1、进行代码混淆，加壳保护或服务器对安装包进行签名校验； 2、发布报在log文件或控制台log中无输出敏感信息； 3、通过重发和修改协议不能获得额外收益或对游戏造成攻击和负面影响； 4、核心游戏逻辑的用户行为与核心数据在服务器校验正确，有防止作弊的发现机制和处理机制； 5、拒绝服务器攻击扫描； 6、第三方组件风险检测，防止游戏运营数据泄露。
技术运营	1、具备弹性动态扩展能力（如前端、逻辑层、存储层在不影响用户使用的情况下支持线上动态扩容能力）； 2、高可用容灾能力（如核心模块和关键路径不允许有单点和无法扩展；大并发访问有防雪崩机制，例如排队、访问频率控制等；要求非关键核心模块故障不影响玩家主逻辑服务，可提供有损服务；服务进程间要求有自动重连机制）； 3、后台服务器无法负载新的请求，客户端需要实时检测并提供拒绝机制，保护后台服务器，比如频率限制； 4、业务数据和日志数据分离，如果不能分离的提供清理策略； 5、Cache的回写频率可配置； 6、DB访问必须不能有全表扫描； 7、应用程序必须支持通过重载配置工具实现动态重载相关的配置文件，而无需中断服务。

Unity 之 C#效率篇

腾讯互娱高级工程师-陶涛

一、简介

效率包括了代码的 GC 大小与内存大小，执行速度等等。其中执行速度不是关注的重点。

Mono

项目主页：http://www.mono-project.com/

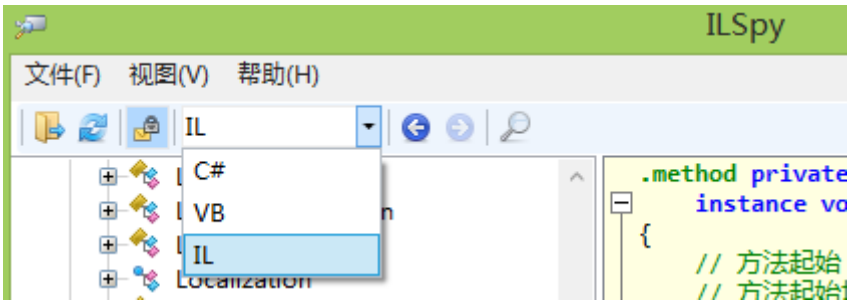
ILSPY

反编译工具，功能比 NET.Reflector 好用多了。

具体的使用方法可以百度。

简单的过程：

把编译的 dll（\Library\ScriptAssemblies 目录下的 Assembly-CSharp.dll）拖入左边窗口，选择需要查看的类以及函数，然后选择翻译的语言，具体如下



环境

IL 代码

举个简单的例子，来说明 IL 代码把，具体可查阅 msdn 文档。

C#代码：

```
object objValue = 4;
int value = (int)objValue;
```

生成的 IL 代码如下：（注释有详细的说明）

```
.locals init (
  [0] object objValue,
  [1] int32 'value'
) //上面IL声明两个局部变量object类型的objValue和int32类型的value变量
IL_0000: nop
IL_0001: ldc.i4.4 //将整型数字4压入栈
IL_0002: box [mscorlib]System.Int32  //执行IL box指令，在内存堆中申请System.Int32类型需要的堆空间
IL_0007: stloc.0 //弹出堆栈上的变量，将它存储到索引为0的局部变量中
IL_0008: ldloc.0//将索引为0的局部变量（即objValue变量）压入栈
IL_0009: unbox.any [mscorlib]System.Int32 //执行IL 拆箱指令unbox.any 将引用类型object转换成System.Int32类型
IL_000e: stloc.1 //将栈上的数据存储到索引为1的局部变量即value
```

二、GC 与内存篇

for 与 foreach 真相

目的

在 Mono 下，研究两种不同方式的 GC 情况。

环境

同简介中的环境

测试代码

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class TestCodPerfomanceScript : MonoBehaviour {

        // Use this for initialization
    List<int> mData = new List<int>();

        void Start () {
        mData.Add(1);  //可有 n 个元素，n 也可以为 0
        }

    void testForForeach_foreach()
    {
        foreach(var e in mData)
        {
            //Debug.Log(e);
```

```
        }
    }

    void testForForeach_for()
    {
int len = mData.Count;
        for (int i = 0; i < len; i++)
        {
        }
}

    void testForForeach_nonforeach()
    {
        var e = mData.GetEnumerator();
        while(e.MoveNext())
        {

        }
    }

         // Update is called once per frame
         void Update () {
        //testForForeach_foreach();
        //testForForeach_for();
        testForForeach_nonforeach();
         }
}
```

测试结果

testForForeach_foreach:

TestCodPerfomanceScript.Update()	0.0%	0.0%	1	40 B

testForForeach_for():

TestCodPerfomanceScript.Update()	0.0%	0.0%	1	0 B

testForForeach_nonforeach():

TestCodPerfomanceScript.Update()	0.0%	0.0%	1	0 B

数据分析

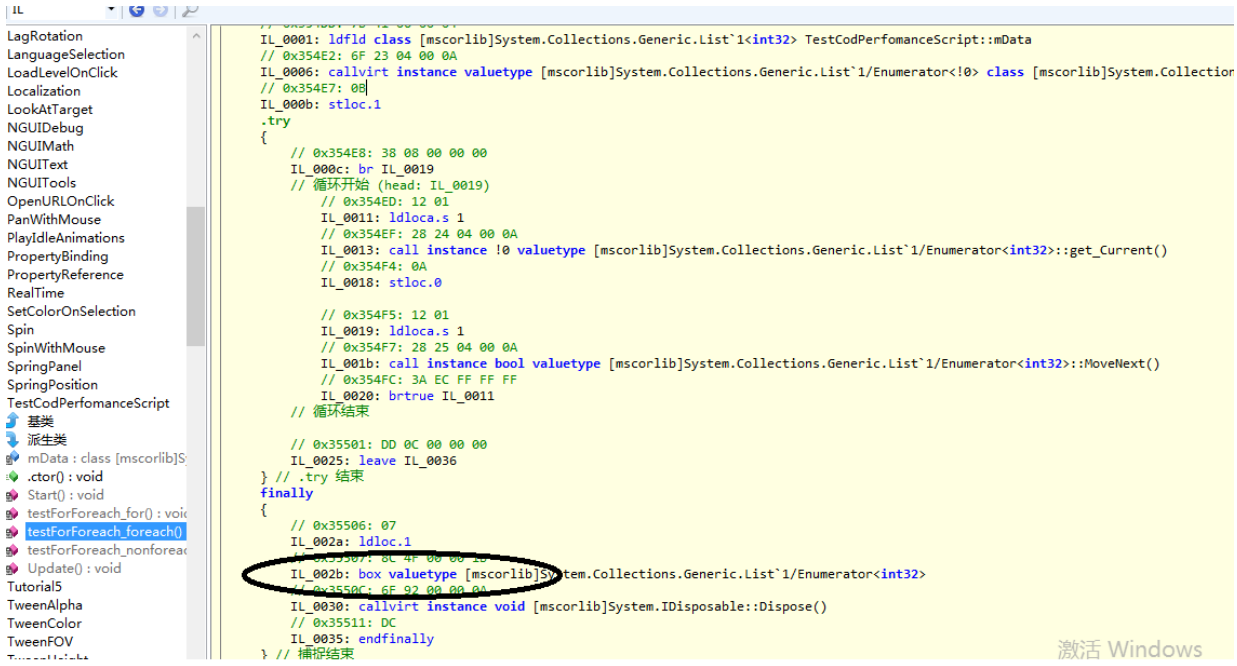
反编译->C#代码

```
private void testForForeach_foreach()
{
    using (List<int>.Enumerator enumerator = this.mData.GetEnumerator())
    {
        while (enumerator.MoveNext())
        {
            int current = enumerator.get_Current();
        }
    }
}

private void testForForeach_for()
{
    for (int i = 0; i < this.mData.get_Count(); i++)
    {
    }
}

private void testForForeach_nonforeach()
{
    while (this.mData.GetEnumerator().MoveNext())
    {
    }
}
```

从代码中，我们可知 foreach 中，编译器加了不少代码。从这里应该是有发生过装箱的操作。从对 testForForeach_nonforeach 函数的反编译 C#代码来看，应该是 using 那块产生装箱操作。继续查看反编译的 IL 代码。

反编译->IL 代码

testForForeach_foreach 的 IL 代码



从图中可以清楚中的看到有一次装箱过程。

数据结论

可以显然可知，foreach 每次有 40B 的 GC 产生 (这个是 Mono 的一个 bug) 。其余两种方式不产生 GC。

产生 GC 的根本原因是使用了 using 语句。（GetEnumerator()返回值类型，在 using 中装箱了）

那么，我们写一段测试代码，在去验证一下：

```
void testForForeach_nonforeachUsing()
    {
        using (var e = mData.GetEnumerator())
        {
            while (e.MoveNext())
            {

            }
        }
    }
```

得到的数据是

意义

所以，在目前的项目中，foreach 还是有 GC 的。（不管是 IList 还是 ArrayList 都会有 GC）

具体见测试数据

建议项目中采用 testForForeach_for()和 testForForeach_nonforeach()的写法。尤其是在 update 或 LateUpdate 中。

在处理 Dictionary 遍历的时候，我们可以这样：

```
 var enumerator = m_Dictionary.GetEnumerator();
while (enumerator.MoveNext())
{
var element = enumerator.Current;
element.Value.UpdateComponent(deltaTime);
}
```

字符串拼接真相

目的

在 Mono 下，研究两种常用的字符串拼接的内存与 GC 问题。

环境

同简介中的环境

测试代码

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.Text;

public class TestCodPerfomanceStringBuilderScript : MonoBehavi
our
{

          // Use this for initialization

          void Start () {
          }

     void testStringBuilder_plus()
     {
         string test = "abc" + "efg" + "h";
         int index = test.Length;
     }

     void testStringBuilder_StringBuilder()
     {
         StringBuilder testData = new StringBuilder();
         testData.Append("abc");
         testData.Append("efg");
         testData.Append("h");
         string test = testData.ToString();

     }

          // Update is called once per frame
          void Update () {
         testStringBuilder_plus();
         //testStringBuilder_StringBuilder();
          }
}
```

测试结果

testStringBuilder_plus ():

▼ TestCodPerfomanceStringBuilderScript.Update()	0.0%	0.0%	1	0 B
TestCodPerfomanceStringBuilderScript.testStringBuilder_plus()	0.0%	0.0%	1	0 B

testStringBuilder_StringBuilder ():

▼ TestCodPerfomanceStringBuilderScript.Update()	0.1%	0.0%	1	178 B
▼ TestCodPerfomanceStringBuilderScript.testStringBui	0.1%	0.0%	1	178 B
▶ StringBuilder.Append()	0.0%	0.0%	3	90 B
▶ StringBuilder.ToString()	0.0%	0.0%	1	40 B
▶ StringBuilder..ctor()	0.0%	0.0%	1	0 B

数据分析

反编译->C#代码

testStringBuilder_plus ():

```
// TestCodPerfomanceStringBuilderScript
private void testStringBuilder_plus()
{
    string text = "abcefgh";
    int length = text.get_Length();
}
```

testStringBuilder_StringBuilder ():

```
// TestCodPerfomanceStringBuilderScript
private void testStringBuilder_StringBuilder()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("abc");
    stringBuilder.Append("efg");
    stringBuilder.Append("h");
    string text = stringBuilder.ToString();
}
```

反编译->IL 代码

testStringBuilder_plus ():

```
    .maxstack 2
    .locals init (
        [0] string,
        [1] int32
    )

    // 0x355D4: 72 73 1C 00 70
    IL_0000: ldstr "abcefgh"
    // 0x355D9: 0A
    IL_0005: stloc.0
    // 0x355DA: 06
    IL_0006: ldloc.0
    // 0x355DB: 6F B2 00 00 0A
    IL_0007: callvirt instance int32 [mscorlib]System.String::get_Length()
    // 0x355E0: 0B
    IL_000c: stloc.1
    // 0x355E1: 2A
    IL_000d: ret
} // 方法 TestCodPerfomanceStringBuilderScript::testStringBuilder_plus 结束
```

testStringBuilder_StringBuilder ():

```
  // 0x355F0: 73 BF 00 00 0A
  IL_0000: newobj instance void [mscorlib]System.Text.StringBuilder::.ctor()
  // 0x355F5: 0A
  IL_0005: stloc.0
  // 0x355F6: 06
  IL_0006: ldloc.0
  // 0x355F7: 72 83 1C 00 70
  IL_0007: ldstr "abc"
  // 0x355FC: 6F C0 00 00 0A
  IL_000c: callvirt instance class [mscorlib]System.Text.StringBuilder [mscorlib]System.Text.StringBuilder::Append(string)
  // 0x35601: 26
  IL_0011: pop
  // 0x35602: 06
  IL_0012: ldloc.0
  // 0x35603: 72 8B 1C 00 70
  IL_0013: ldstr "efg"
  // 0x35608: 6F C0 00 00 0A
  IL_0018: callvirt instance class [mscorlib]System.Text.StringBuilder [mscorlib]System.Text.StringBuilder::Append(string)
  // 0x3560D: 26
  IL_001d: pop
  // 0x3560E: 06
  IL_001e: ldloc.0
  // 0x3560F: 72 93 1C 00 70
  IL_001f: ldstr "h"
  // 0x35614: 6F C0 00 00 0A
  IL_0024: callvirt instance class [mscorlib]System.Text.StringBuilder [mscorlib]System.Text.StringBuilder::Append(string)
  // 0x35619: 26
  IL_0029: pop
  // 0x3561A: 06
  IL_002a: ldloc.0
  // 0x3561B: 6F C2 00 00 0A
  IL_002b: callvirt instance string [mscorlib]System.Text.StringBuilder::ToString()
  // 0x35620: 0B
  IL_0030: stloc.1
  // 0x35621: 2A
  IL_0031: ret
// 方法 TestCodPerfomanceStringBuilderScript::testStringBuilder_StringBuilder 结束
```


分析

从上面的测试用例得出，testStringBuilder_plus ()不产生 GC，而 testStringBuilder_StringBuilder()产生 GC，和我们之前的理解有些偏差。是不是我们的测试用例还不够全，我们把 3 次相加变成 300 次试试。

代码如下：

```
void testStringBuilder_plus_More()
    {
        string test = "abc" + "efg" + "h";
        for (int i = 0; i < 300; i++)
```

```
        {
            test += "testStringBuilder_plus_testStringBuilder_
plus_More";
        }
        int index = test.Length;
    }

    void testStringBuilder_StringBuilder_More()
    {
        StringBuilder testData = new StringBuilder();
        for (int i = 0; i < 300; i++)
        {
            testData.Append("testStringBuilder_plus_testString
Builder_plus_More");
        }
        string test = testData.ToString();
}
```

测试结果如下：

testStringBuilder_plus_More()

TestCodPerfomanceStringBuilderScript.testStringBuilder_plus_More()	92.9%	0.9%	1	4.3 MB
String.Concat()	92.0%	2.6%	300	4.3 MB
String.InternalAllocateStr()	48.2%	1.8%	300	4.3 MB
String.CharCopy()	41.1%	0.4%	600	0 B

testStringBuilder_StringBuilder_More()

TestCodPerfomanceStringBuilderScript.testStringBuilder_StringBuilder_M	8.0%	2.7%	1	100.1 KB
StringBuilder.Append()	4.1%	0.8%	300	100.0 KB
StringBuilder.ToString()	1.1%	0.0%	1	0 B
StringBuilder..ctor()	0.0%	0.0%	1	0 B

反编译代码分别如下：

testStringBuilder_plus_More()

```
private void testStringBuilder_plus_More()
{
    string text = "abcefgh";
    for (int i = 0; i < 300; i++)
    {
        text += "testStringBuilder_plus_testStringB
    }
    int length = text.get_Length();
}
```

```
// 0x35634: 72 73 1C 00 70
IL_0000: ldstr "abcefgh"
// 0x35639: 0A
IL_0005: stloc.0
// 0x3563A: 16
IL_0006: ldc.i4.0
// 0x3563B: 0B
IL_0007: stloc.1
// 0x3563C: 38 10 00 00 00
IL_0008: br IL_001d
// 循环开始 (head: IL_001d)
    // 0x35641: 06
    IL_000d: ldloc.0
    // 0x35642: 72 97 1C 00 70
    IL_000e: ldstr "testStringBuilder_plus_testStringBuilder_plus_More"
    // 0x35647: 28 23 00 00 0A
    IL_0013: call string [mscorlib]System.String::Concat(string, string)
    // 0x3564C: 0A
    IL_0018: stloc.0
    // 0x3564D: 07
    IL_0019: ldloc.1
    // 0x3564E: 17
    IL_001a: ldc.i4.1
    // 0x3564F: 58
    IL_001b: add
    // 0x35650: 0B
    IL_001c: stloc.1

    // 0x35651: 07
    IL_001d: ldloc.1
    // 0x35652: 20 2C 01 00 00
    IL_001e: ldc.i4 300
    // 0x35657: 3F E5 FF FF FF
    IL_0023: blt IL_000d
// 循环结束

// 0x3565C: 06
IL_0028: ldloc.0
// 0x3565D: 6F B2 00 00 0A
IL_0029: callvirt instance int32 [mscorlib]System.String::get_Length()
```

testStringBuilder_StringBuilder_More()

```
private void testStringBuilder_StringBuilder_More()
{
    StringBuilder stringBuilder = new StringBuilder(
    for (int i = 0; i < 300; i++)
    {
        stringBuilder.Append("testStringBuilder_plus
    }
    string text = stringBuilder.ToString();
}
```

```
// 0x35670: 73 BF 00 00 0A
IL_0000: newobj instance void [mscorlib]System.Text.StringBuilder::.ctor()
// 0x35675: 0A
IL_0005: stloc.0
// 0x35676: 16
IL_0006: ldc.i4.0
// 0x35677: 0B
IL_0007: stloc.1
// 0x35678: 38 10 00 00 00
IL_0008: br IL_001d
// 循环开始 (head: IL_001d)
    // 0x3567D: 06
    IL_000d: ldloc.0
    // 0x3567E: 72 97 1C 00 70
    IL_000e: ldstr "testStringBuilder_plus_testStringBuilder_plus_More"
    // 0x35683: 6F C0 00 00 0A
    IL_0013: callvirt instance class [mscorlib]System.Text.StringBuilder [mscorlib]System.
    // 0x35688: 26
    IL_0018: pop
    // 0x35689: 07
    IL_0019: ldloc.1
    // 0x3568A: 17
    IL_001a: ldc.i4.1
    // 0x3568B: 58
    IL_001b: add
    // 0x3568C: 0B
    IL_001c: stloc.1
```

在 300 次的数量下面，结果很明显。

数据结论

不同的使用场景结果是不同的，+ 号在几次连接中没有产生 GC。而在大量的连接过程中，产生大量的 GC。

意义

连接次数只有几次（10 以内），此时应该直接用 + 号连接，不产生 GC。实际上，编译器已经做了优化。

其余的使用 StringBuilder，StringBuilder 内部 Buffer 的缺省值为 16，按 StringBuilder 的使用场景，Buffer 肯定得重新分配。经验值一般用 256 作为 Buffer 的初值。当然，如果能计算出最终生成字符串长度的话，则应该按这个值来设定 Buffer 的初值。使用 new StringBuilder(256) 就将 Buffer 的初始长度设为了 256。

Struct 与 Class 真相之一

http://dev.yesky.com/msdn/359/3486359.shtml

目的

Struct 和 Class 的基本情况大致已经清楚，我们的真正目的是想知道如何架构我们的数据操作模块，才能充分地利用值类型和引用类型的，让效率最高（即装箱

或拆线少，堆内存少）。

环境

同简介中的环境

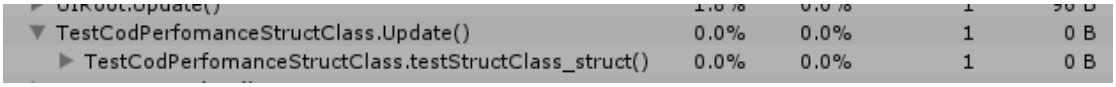
测试代码

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.Text;

public class TestCodPerfomanceStructClass : MonoBehaviour
{
    public struct PathStruct
    {
        public string path;
    }
    public class PathClass
    {
        public string path;
    }
    List<PathStruct> PathStructList = new List<PathStruct>();

    List<PathClass> PathClassList = new List<PathClass>();
    List<PathStruct> PathStructModifyList = new List<PathStruc
t>();

         // Use this for initialization

         void Start () {
        PathStruct mPathStruct = new PathStruct();
        mPathStruct.path = "def";
        PathStructModifyList.Add(mPathStruct);
         }

    void testStructClass_struct()
    {
        PathStruct mPathStruct = new PathStruct();
        mPathStruct.path = "abc";
        PathStructList.Add(mPathStruct);
    }

    void testStructClass_class()
    {
        PathClass mPathStruct = new PathClass();
        mPathStruct.path = "abcdef";
        PathClassList.Add(mPathStruct);
```

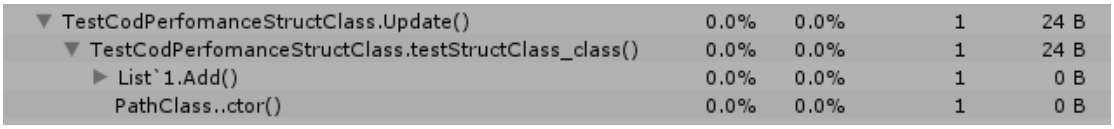
```
    }

    void testStructClass_struct_modifyData()
    {
        PathStruct mPathStruct = PathStructModifyList[0];
        mPathStruct.path = "new";
PathStructModifyList[0] = mPathStruct;
        //PathStructModifyList.RemoveAt(0);
        //PathStructModifyList.Add(mPathStruct);
    }

         // Update is called once per frame
         void Update () {
        //testStructClass_struct();
        //testStructClass_class();
        testStructClass_struct_modifyData();
         }

}
```

测试结果

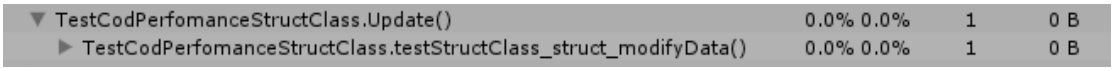
testStructClass_struct():



testStructClass_class():



testStructClass_struct_modifyData():



数据分析

这块内容比较简单,只介绍一下 testStructClass_struct_modifyData()函数的 IL 代码。

```
.maxstack 7
.locals init (
    [0] valuetype TestCodPerfomanceStructClass/PathStruct
)

// 0x3576C: 02
IL_0000: ldarg.0
// 0x3576D: 7B 44 06 00 04
IL_0001: ldfld class [mscorlib]System.Collections.Generic.List`1<valuetype TestCodPerfomanceStructClass/PathStr
// 0x35772: 16
IL_0006: ldc.i4.0
// 0x35773: 6F 2A 04 00 0A
IL_0007: callvirt instance !0 class [mscorlib]System.Collections.Generic.List`1<valuetype TestCodPerfomanceStru
// 0x35778: 0A
IL_000c: stloc.0
// 0x35779: 12 00
IL_000d: ldloca.s 0
// 0x3577B: 72 13 1D 00 70
IL_000f: ldstr "new"
// 0x35780: 7D 45 06 00 04
IL_0014: stfld string TestCodPerfomanceStructClass/PathStruct::path
// 0x35785: 02
IL_0019: ldarg.0
// 0x35786: 7B 44 06 00 04
IL_001a: ldfld class [mscorlib]System.Collections.Generic.List`1<valuetype TestCodPerfomanceStructClass/PathStr
// 0x3578B: 16
IL_001f: ldc.i4.0
// 0x3578C: 6F 2B 04 00 0A
IL_0020: callvirt instance void class [mscorlib]System.Collections.Generic.List`1<valuetype TestCodPerfomanceSt
// 0x35791: 02
IL_0025: ldarg.0
// 0x35792: 7B 44 06 00 04
IL_0026: ldfld class [mscorlib]System.Collections.Generic.List`1<valuetype TestCodPerfomanceStructClass/PathStr
// 0x35797: 06
IL_002b: ldloc.0
// 0x35798: 6F 28 04 00 0A
```

数据结论

Struct 在栈中不产生 GC，class 在堆中，会产生 GC。

对 Struct 的结点修改时，修改完以后记得重新赋值。因为 Struct 赋值是 copy 而不是引用，修改完以后，以前的不生效。

意义

堆栈的空间有限，对于大量的逻辑的对象，创建类要比创建结构好一些。

结构表示轻量对象，并且结构的成本较低，适合处理大量短暂的对象。

在表现抽象和多级别的对象层次时，类是最好的选择。

大多数情况下该类型只是一些数据时，结构是最佳的选择。

Struct 与 Class 真相之二

目的

在真相之一中，我们所有的集合结构是 List，在 C#中数组，ArrayList，List 都能够存储一组对象，那么这三者到底有什么样的区别呢？以及对上面提到的处理结果以及处理过程产生什么样的影响？

数组：内存中是连续存储的，索引速度非常快，赋值与修改元素也很简单。但不利于动态扩展以及移动。

因为数组的缺点，就产生了 ArrayList。

ArrayList：使用该类时必须进行引用，同时继承了 IList 接口，提供了数据存储和检索，ArrayList 对象的大小动态伸缩，支持不同类型的结点。
ArrayList 虽然很完美，但结点类型是 Object，故不是类型安全的，也可能发生装箱和拆箱操作，带来很大的性能耗损。
List 是泛型接口，规避了 ArrayList 的两个问题。

现在，我们试试集合用 ArrayList，看下有什么异同?

环境

同简介中的环境

测试代码

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.Text;

public class TestCodPerfomanceStructClassArrayList : MonoBehav
iour
{
    public struct PathStruct
    {
        public string path;
    }

    public interface IPathStructInterface
    {
        string path { get; set; }
    }

    public struct PathStructInterface : IPathStructInterface
    {
        private string _path;
        public string path
        {
            get { return _path; }
            set { _path = value; }
        }
    }

    ArrayList PathStructList = new ArrayList();
    ArrayList PathStructListInterface = new ArrayList();
    ArrayList PathStructModifyList = new ArrayList();
    ArrayList PathStructModifyInterfaceList = new ArrayList();
```

```
    // Use this for initialization

    void Start()
    {
        PathStruct mPathStruct = new PathStruct();
        mPathStruct.path = "def";
        PathStructModifyList.Add(mPathStruct);

        PathStructInterface mPathStructInterface = new PathStr
uctInterface();
        mPathStructInterface.path = "def";
        PathStructModifyInterfaceList.Add(mPathStructInterface
);
    }

    void testStructClass_struct()
    {
        PathStruct mPathStruct = new PathStruct();
        mPathStruct.path = "abc";
        PathStructList.Add(mPathStruct);
    }
    void testStructClass_struct_interface()
    {
        PathStructInterface mPathStructInterface = new PathStr
uctInterface();
        mPathStructInterface.path = "abc";
        PathStructListInterface.Add(mPathStructInterface);
    }

    void testStructClass_struct_modifyData()
    {
        PathStruct mPathStruct = (PathStruct)PathStructModifyL
ist[0];
        mPathStruct.path = "new";
PathStructModifyList[0] = mPathStruct;
        //PathStructModifyList.RemoveAt(0);
        //PathStructModifyList.Add(mPathStruct);
    }

    void testStructClass_struct_modifyData_interface()
    {
        IPathStructInterface mPathStruct = (IPathStructInterfa
ce)PathStructModifyInterfaceList[0];
        mPathStruct.path = "new";

    }
```

```
    // Update is called once per frame
    void Update()
    {
        testStructClass_struct();
        testStructClass_struct_interface();
        testStructClass_struct_modifyData();
        testStructClass_struct_modifyData_interface();
    }

}
```

测试结果

TestCodPerfomanceStructClassArrayList.Update()	0.2%	0.0%	1	72 B
TestCodPerfomanceStructClassArrayList.testStructClass_struct()	0.0%	0.0%	1	24 B
TestCodPerfomanceStructClassArrayList.testStructClass_struct_interface()	0.0%	0.0%	1	24 B
TestCodPerfomanceStructClassArrayList.testStructClass_struct_modifyData()	0.1%	0.0%	1	24 B
TestCodPerfomanceStructClassArrayList.testStructClass_struct_modifyData_interface()	0.0%	0.0%	1	0 B

数据分析

testStructClass_struct_interface();

```
    // 方法起始地址（相对于文件绝对值：0x35880）
    // 代码长度 39 (0x27)
    .maxstack 5
    .locals init (
        [0] valuetype TestCodPerfomanceStructClassArrayList/PathStructInterface
    )

    // 0x3588C: 12 00
    IL_0000: ldloca.s 0
    // 0x3588E: FE 15 D4 00 00 02
    IL_0002: initobj TestCodPerfomanceStructClassArrayList/PathStructInterface
    // 0x35894: 12 00
    IL_0008: ldloca.s 0
    // 0x35896: 72 83 1C 00 70
    IL_000a: ldstr "abc"
    // 0x3589B: 28 43 07 00 06
    IL_000f: call instance void TestCodPerfomanceStructClassArrayList/PathStructInterface::set_path(string)
    // 0x358A0: 02
    IL_0014: ldarg.0
    // 0x358A1: 7B 48 06 00 04
    IL_0015: ldfld class [mscorlib]System.Collections.ArrayList TestCodPerfomanceStructClassArrayList::PathSt
    // 0x358A6: 06
    IL_001a: ldloc.0
    // 0x358A7: 8C D4 00 00 02
    IL_001b: box TestCodPerfomanceStructClassArrayList/PathStructInterface
    // 0x358AC: 6F 2D 04 00 0A
    IL_0020: callvirt instance int32 [mscorlib]System.Collections.ArrayList::Add(object)
    // 0x358B1: 26
    IL_0025: pop
    // 0x358B2: 2A
    IL_0026: ret
} // 方法 TestCodPerfomanceStructClassArrayList::testStructClass_struct_interface 结束
```

testStructClass_struct_modifyData_interface();

```
    .maxstack 5
    .locals init (
        [0] class TestCodPerfomanceStructClassArrayList/IPathStructInterface
    )

    // 0x3590C: 02
    IL_0000: ldarg.0
    // 0x3590D: 7B 4A 06 00 04
    IL_0001: ldfld class [mscorlib]System.Collections.ArrayList TestCodPerfomanceStructClassArrayList::PathStruct
    // 0x35912: 16
    IL_0006: ldc.i4.0
    // 0x35913: 6F 2E 04 00 0A
    IL_0007: callvirt instance object [mscorlib]System.Collections.ArrayList::get_Item(int32)
    // 0x35918: 74 D3 00 00 02
    IL_000c: castclass TestCodPerfomanceStructClassArrayList/IPathStructInterface
    // 0x3591D: 0A
    IL_0011: stloc.0
    // 0x3591E: 06
    IL_0012: ldloc.0
    // 0x3591F: 72 13 1D 00 70
    IL_0013: ldstr "new"
    // 0x35924: 6F 41 07 00 06
    IL_0018: callvirt instance void TestCodPerfomanceStructClassArrayList/IPathStructInterface::set_path(string)
    // 0x35929: 2A
    IL_001d: ret
} // 方法 TestCodPerfomanceStructClassArrayList::testStructClass_struct_modifyData_interface 结束
```

数据结论

第二个函数有装箱的原因产生的对象是值类型。

第四个函数实现无 GC 的原因是：由于产生的临时变量的类型不同，前者已经在前面进行说明了，后者由于产生的临时变量的类型为 IPathStructInterface，属于引用类型，那么相当于临时变量就是原对象的引用，那么对于对于它的修改会直接修改到原对象，因此是可以的。可以说这里的不同本身在于产生临时对象的类型不同，从而造成本质的区别。

在使用 Arraylist 和结构体搭配的时候，修改结点的时候，建议按照第四个函数的方式去实现，较少一次 GC 的产生。

意义

减少 GC 的产生。

善用 ref。

Enum 真相

目的

在 Mono 下，研究 Enum 使用不当情况下的 GC。

环境

同简介中的环境

测试代码

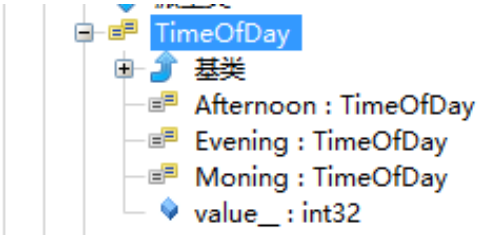
```
public class TestCodEnum : MonoBehaviour
{
```

```
    public enum TimeOfDay
    {
        Moning = 0,
        Afternoon = 1,
        Evening = 2,
    };
         // Use this for initialization
         void Start () {

         }

    private void TestDic()
    {
        Dictionary<TimeOfDay, int> dicData = new Dictionary<Ti
meOfDay, int>();
        dicData.Add(TimeOfDay.Evening, 1);
    }

    private void TestStr()
    {
        string mm = TimeOfDay.Evening.ToString();
    }

         // Update is called once per frame
         void Update () {
        TestDic();
        TestStr();
         }
}
```

测试结果

TestCodEnum.Update()	1.4%	0.0%	1	0.6 KB
TestCodEnum.TestStr()	1.1%	0.0%	1	140 B
Enum.ToString()	1.1%	0.0%	1	120 B
TestCodEnum.TestDic()	0.2%	0.0%	1	508 B
Dictionary`2..ctor()	0.0%	0.0%	1	400 B
Dictionary`2.Add()	0.0%	0.0%	1	20 B

数据分析



```
.class nested public auto ansi sealed TimeOfDay
    extends [mscorlib]System.Enum
{
    // 成员
    .field public specialname rtspecialname int32 value__
    .field public static literal valuetype TestCodEnum/TimeOfDay Moning = int32(0)
    .field public static literal valuetype TestCodEnum/TimeOfDay Afternoon = int32(1)
    .field public static literal valuetype TestCodEnum/TimeOfDay Evening = int32(2)

} // 类 TimeOfDay 结束
```

```
.method private hidebysig
    instance void TestStr () cil managed
{
    // 方法起始 RVA 地址 0x372ac
    // 方法起始地址（相对于文件绝对值：0x354ac）
    // 代码长度 13 (0xd)
    .maxstack 2
    .locals init (
        [0] string
    )

    // 0x354B8: 18
    IL_0000: ldc.i4.2
    // 0x354B9: 8C CD 00 00 02
    IL_0001: box TestCodEnum/TimeOfDay
    // 0x354BE: 6F 45 00 00 0A
    IL_0006: callvirt instance string [mscorlib]System.Enum::ToString()
    // 0x354C3: 0A
    IL_000b: stloc.0
    // 0x354C4: 2A
    IL_000c: ret
} // 方法 TestCodEnum::TestStr 结束
```

```
thod private hidebysig
 instance void TestDic () cil managed

 // 方法起始 RVA 地址 0x37290
 // 方法起始地址（相对于文件绝对值：0x35490）
 // 代码长度 15 (0xf)
 .maxstack 4
 .locals init (
     [0] class [mscorlib]System.Collections.Generic.Dictionary`2<valuetype TestCodEnum/TimeOfDay, int32>
 )

 // 0x3549C: 73 23 04 00 0A
 IL_0000: newobj instance void class [mscorlib]System.Collections.Generic.Dictionary`2<valuetype TestCodEnum/Ti
 // 0x354A1: 0A
 IL_0005: stloc.0
 // 0x354A2: 06
 IL_0006: ldloc.0
 // 0x354A3: 18
 IL_0007: ldc.i4.2
 // 0x354A4: 17
 IL_0008: ldc.i4.1
 // 0x354A5: 6F 24 04 00 0A
 IL_0009: callvirt instance void class [mscorlib]System.Collections.Generic.Dictionary`2<valuetype TestCodEnum/
 // 0x354AA: 2A
 IL_000e: ret
/ 方法 TestCodEnum::TestDic 结束
```

数据结论

TestStr() 有装箱发生。

TestDic()，在构造集合的时候，也会有 GC 的产生。

意义

不要把枚举当 Tkey 使用，不要把枚举转成 string 使用。

闭包真相

目的

了解闭包的方式带来的 GC。

先了解一下变量、委托的基本概念：

先要简单描述下变量的作用域以及生命周期。

作用域：成员变量作用于类、局部变量作用于函数、次局部变量作用于函数局部片段。

生命周期：变量随着其寄存对象生而生和消亡(不包括非实例化的 static 和 const 对象）。

委托：

委托概念：是一个类型安全的对象，它指向程序中另一个以后会被调用的方法(或多个方法）。通俗的说，委托是一个可以引用方法的对象，当创建一个委托，也就创建一个引用方法的对象，进而就可以调用那个方法，即委托可以调用它所指的方法。

写法：

public delegate void xxxxx();

委托 委托名=new 委托（会调用的方法名); 委托名（参数）；

委托 委托名 =会调用的方法名; 委托名（参数）；

委托 委托名=delegate(参数）{会调用的方法体};委托名（参数）；

委托 委托名=（（参数 1，。。参数 n）=>{会调用的方法体}）；委托名（参数）；

用 Action<T>和 Func<T>,第一个无返回值

Func<参数 1, 参数 2, 返回值> 委托名= ((参数 1，参数 2) => {带返回值的方法体 });返回值=委托名（参数 1，参数 2）；

目的：适合多人开发，模块解耦；提高设计接口的水平。

闭包：

闭包概念：函数和与其相关的引用环境组合而成的实体。

本质 1：代码块依然维护着它第一个被创建时环境（执行上下文）- 即它仍可以使用创建它的方法中局部变量，即使那个方法已经执行完了。

本质 2：Closures close over variables, not over values。

目的：从用户角度考虑的一种设计概念，它基于对上下文的分析，把龌龊的事情、复杂的事情和外部环境交互的事情都自己做了，留给用户一个很自然的接口。

环境

同简介中的环境

测试代码

```
using UnityEngine;
using System.Collections;
using System;

public class TestCodPerfomanceClosure : MonoBehaviour {
         void Start () {

         }

         // Update is called once per frame
         void Update () {
        int value = 3;
        //TestDelegate testFunc = ((int index) =>
        //{
        //    return (++value) * index;
        //});
        //int res = testFunc(4); //16
        //int res1 = testFunc(4); //20

        Action action = () =>
        {
            int y = 2;
            int result = value + y;
        };

        action();
         }
}
```

测试结果

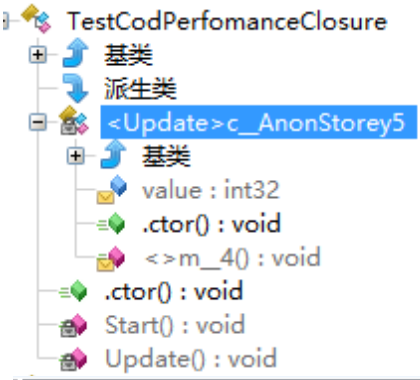
TestCodPerfomanceClosure.Update()	0.0%	0.0%	1	124 B
<Update>c__AnonStorey5..ctor()	0.0%	0.0%	1	0 B
<Update>c__AnonStorey5.<>m__4()	0.0%	0.0%	1	0 B

数据分析

首先看 update 中的 IL 代码：

```
// 0x3549C: 73 06 00 00 0A
IL_0000: newobj instance void TestCodPerfomanceClosure/'<Update>c__AnonStorey5'::.ctor()
// 0x354A1: 0B
IL_0005: stloc.1
// 0x354A2: 07
IL_0006: ldloc.1
// 0x354A3: 19
IL_0007: ldc.i4.3
// 0x354A4: 7D 62 06 00 04
IL_0008: stfld int32 TestCodPerfomanceClosure/'<Update>c__AnonStorey5'::'value'
// 0x354A9: 07
IL_000d: ldloc.1
// 0x354AA: FE 06 07 08 00 06
IL_000e: ldftn instance void TestCodPerfomanceClosure/'<Update>c__AnonStorey5'::'<>m__4'()
// 0x354B0: 73 23 04 00 0A
IL_0014: newobj instance void [System.Core]System.Action::.ctor(object, native int)
// 0x354B5: 0A
IL_0019: stloc.0
// 0x354B6: 06
IL_001a: ldloc.0
// 0x354B7: 6F 24 04 00 0A
IL_001b: callvirt instance void [System.Core]System.Action::Invoke()
// 0x354BC: 2A
IL_0020: ret
// 方法 TestCodPerfomanceClosure::Update 结束
```

在 IL 代码中，出现了一个新的类，如下：




```
.class nested private auto ansi sealed beforefieldinit '<Update>c__AnonStorey5'
    extends [mscorlib]System.Object
{
    .custom instance void [mscorlib]System.Runtime.CompilerServices.CompilerGeneratedAttri
        01 00 00 00
    )
    // 成员
    .field assembly int32 'value'

    // 方法
    .method public hidebysig specialname rtspecialname
        instance void .ctor () cil managed
    {
        // 方法起始 RVA 地址 0x37c08
        // 方法起始地址（相对于文件绝对值）：0x35c08
```

如何没有涉及到闭包的话，委托代码只生产一个函数而不是一个类。

数据结论

使用闭包的时候，涉及到大约 120B 的 GC，随着闭包引用内容的增加而增大。

意义

看自己的需求，如果函数调用频繁的话，要考虑是否不使用闭包来实现。

三、其它篇

U3D 对象

MonoBehaviour

如果没有相应的事件处理，删除对应的空函数

Update 优化

如果没必要每帧的逻辑，可以降低频率，方法如下：

Void Update（）{if（Time.frameCount%6==0）{DoSomething();}}

如果没必要每帧的逻辑，可以使用周期性的协程

如果没必要每帧的逻辑，可以使用

InvokeRepeating("DoSomeThing",0.5f,1.0f);

Gameobject 不可见时，设置 enabled = false 时，update 就会停止调用。

在 update 中尽量不要调用查找对象或组件方法如 FindByTag 或 Find 等等。可以在 start 中先缓存下来，然后使用。

协程 WaitForSeconds 优化

我想要指出的是使用 yield return new WaitForSeconds() 将会每帧导致垃圾分配GC，21个字节，由于new 部分（相对于标准的协程 yield return null只产生 9 个字节）。

若要避免此问题，只是提前设置你的wait 等待的时间.....

```
WaitForSeconds shortWait = new WaitForSeconds(0.1f);
WaitForSeconds longWait = new WaitForSeconds(5.0f);
IEnumerator myEvenAwesomerCoroutine()
{
    while (true)
    {
        if (iNeedToDoStuffFast)
        {
            doAwesomeStuffReallyFast();
            yield return shortWait;

        }
        else{
            dontDoMuch();
            yield return longWait;
        }
    }
}
```

现在你coroutine 协程每次调用只会引起最低 的9 字节 GC 分配 （不包括其他分配allocations，当然你可能通过您其他的代码会导致 ！）。

Component 相关优化

Transform

使用内建的数组，比如用 Vector3.zero 而不是 new Vector(0, 0, 0);

transform.localRotation = Quaternion.Euler(Vector3.zero);

transform.localScale =Vector3.one;

transform.localPosition =Vector3.one;

GameObject 相关优化

（脚本和本地引擎 C++代码之间的通信开销）

Gameobject 缓存：类似组件的缓存策略。

查找对象标签：if (go.CompareTag（“xxx”)来代替 if (go.tag == “xxx”)，因为内部循环调用对象分配的标签属性以及拷贝额外内存。

SendMessageUpwards、SendMessage：少用这两个函数，使用委托替代。

缓存组件：调用 GetComponent 函数耗性能，用变量先缓存到内存在使用, 有必要时记得更新缓存组件。

NGUI 优化

优化数学运算，尽量采用整型代替浮点型，除法改乘法。

《御龙在天移动版》UI 性能优化总结

腾讯互娱高级工程师-刘绵光

导语：《御龙在天移动版》是一款 MMORPG 国战手游，全 3D 视角、同屏人数多和复杂的 UI 逻辑玩法是我们的主要特点。本文将介绍御龙手游在前期 UI 框架选型的思路和使用 UGUI 开发过程中，遇到的常见性能问题以及解决方案。

NGUI vs UGUI 选择

我们的项目在 demo 预研阶段使用的是当时最新的 Unity4.5.x 版本，没得选，UI 用的是 NGUI。demo 阶段的 UI 其实只有一个摇杆和四个技能按钮，UI 系统框架的代码更是一篇空白。在 demo 预研后期，Unity4.6 版本发布了！带来了全新的 UGUI，还是挖的 NGUI 的作者去做的，应该靠谱，于是我这边开始预研引入 UGUI 的阶段。

各种测试、源码分析、官方数据、论坛数据对比之后，我们最终决定选择 UGUI，具体的原因主要有以下几点：

1.UI 的渲染顺序和 Hierarchy 节点布局相关，非常直观，这点与端游的 UI 编辑器 tenio 是一样的，对于整个端游开发团队转移手游开发可以降低学习成本。重点是 NGUI 依赖 depth 来设置渲染顺序，这个实在无法认同，对于 MMORPG 复杂的 UI 界面来说后期维护起来会很麻烦；

2.强大的 Rectranform，对于编辑 UI 来说比 NGUI 方便很多；

3. Unity 官方的 UI 系统，在兼容性和后续官方可优化空间上来说，原生的 UI 系统具有一定优势。在后续的 5.2 版本证明了这一点，例如：Canvas.BuildBatch()，合批 Canvas 下所有网格，这个性能热点在 5.2 版本后挪到了子线程去做减轻了主线程的压力，而 NGUI 作为一个插件没法做到这一点，网格合批的性能热点还是耗在主线程的 UIPanel.LateUpdate()；

4.性能上有一个略微的优势点：UGUI 的 UIMesh 生成是通过底层 C++代码实现的，而 NGUI 只能通过上层的不断创建 Vertex List，这样在堆内存的管理上，UGUI 确实要好很多，带来的隐形收益就是 GC 触发次数会少很多。

当然，我们选择了 UGUI，并不是说 UGUI 一定比 NGUI 好，衡量一套 UI 系统的好坏应该是全方位的：性能、配套工具、易用性、稳定性等，UGUI 在配套工具上还有待加强，我们也相信官方会不断完善改进（5.2 版本确实给力改进了很多）。

关于 NGUI 和 UGUI 的性能对比，我没有去做一个全方位的真机测试比较，看了网上不少帖子讨论过这个问题，大家的结论是 UGUI 有略微的优势但差别不大。现在回过头来看咱们部门上线的两款手游，九龙战（NGUI）和御龙在天移动版（UGUI）都是性能表现优异的游戏，当然两款游戏的类型不一样，UI 复杂度也不一样，没有直接的可比性。我个人觉得 **NGUI 和 UGUI 都是很好的 UI 框架，只要把它们的原理特性掌握好，都可以做出性能很好的 UI 界面。**

补充一点：我们项目组目前用的 Unity 版本还是 4.6.9，出于稳定性和风险等综合因素考虑没有贸然升级 5.x 版本，所以 5.2 版本 UGUI 升级带来的各种优化改进和我们也没啥关系了，只是对新版本做了一些预研的工作，所以接下来讨论的性能优化都是基于 Unity4.6.9 版本。

UI 资源规范（内存优化）

客户端做任何的性能优化首先想到的都是规范美术资源，前期不给美术资源定制一定的规范，后期做优化性能会非常的被动。对于 UI 资源的规范，主要是考虑的内存优化。

1.任何的 UI 图集最大 size 1024*1024（内存优化）；
2.同一个界面出现的 UI 资源尽量放到一个图集，重复利用的公用资源放 common（DrawCall 优化）；
3.能用九宫格的尽量用九宫格来减小原图大小（内存优化）；
4.美术给过来的 UI 原图 size 尽量小，对于一些全屏的 loading 原画图，原画大小是 1136*640（我们 UI 的标准分辨率），让美术按照比例高度缩小到 500，这样一张 1024*1024 的图集就可以放两张原图了，提升图集利用率。对于一些

600*400 类似大小的原图，就尽量按比例把最长边压小到 500，这样出来的图集就是 512*512 而不是 1024*1024（内存优化）；
5.对于特别长条的 UI 原图，例如 1000*100，如果由于加入这个长条的原图导致图集大小变大而且利用率很低的话，要把 1000*100 的原图拆分成两张图 500*100，在制作界面的时候用两个 Image 拼接即可，这样可以把 1024 的图集缩小到 512（内存优化）；
6.图集利用率低于 1/3 的时候，要考虑和其他同一个 size 的图集合并以提升利用率。合并的原则是不改变任何一个图集的大小，这样即可完全省掉一张图集（内存优化、安装包量优化）；
7.尽量复用 UI 资源，减少不必要的原图，例如一个卡牌分了五种品质原画底图，白蓝黄绿紫，就不要使用五张大底图了，让美术同事画一个灰色原图，Image 在使用的时候直接按需求修改顶点色即可（内存优化）；
8.关闭 mipmaps（内存优化）。

GPU 优化

谈及 GPU，我们知道它负责整个渲染流水线，从处理 CPU 传递过来的模型数据开始，进行 Vertex Shader、Fragment Shader 等一系列工作，最后输出屏幕上的每个像素。因此我们可以从两个方面着手，优化 Shader、减少 OverDraw。

Shader 优化

我们的 UI Shader 是自己编写的，没有使用默认的 UI-Default.shader，因此 Shader 优化非常有必要。

1.Fog { Mode Off }，最早有一个版本我们没有关闭 Fog，导致打开全屏 UI 的时候帧率明显往下掉（我们的全屏 UI 开启后会关闭主相机，理论上渲染压力和 CPU 消耗会降低），后来 Adreno Profiler 真机调式 Shader 代码后发现了此问题；
2.Fragment 剔除掉 Alpha 为 0 的像素点，减少 OverDraw；
3.尽量简化 Shader 代码，最早我们有一些 UI 效果的需求全都写在了一个 Shader 里，其实 99%的 UI 不要这些东西，单独把效果需求代码拎出来独立一个 Shader。

OverDraw 优化

在每帧绘制中，如果一个像素被反复绘制的次数越多，那么它占用的资源也必然更多。目前在移动设备上，OverDraw 的压力主要来自半透明物体。因为多数情况下，半透明物体需要开启 Alpha Blend 且关闭 ZWrite，同时如果我们绘制像 alpha=0 这种实际上不会产生效果的颜色上去，也同样有 Blend 操作，这是一种极大的浪费。

我们的 UI 绘制是 Alpha Blend 且关闭 ZWrite，因此 UI OverDraw 的优化主要是在制作界面的时候减少 UI 重叠层级（和策划、美术 pk）。除此之外还是有一些我们程序可以控制的优化点：

1.对于九宫格的 Image，如果去掉 fillcenter 不影响最后出来的效果就要把 fillcenter 去掉，可以减少中间一片的像素绘制；
2.看不见的元素且没有逻辑功能要 disable 或者挪出裁剪区域，而不要通过设置 Alpha=0 来隐藏；
3.不要使用一张 Alpha=0 的 Image 来实现放大响应区域的功能；
4.UI 底层系统来控制隐藏看不见的元素，例如打开全屏 UI 的时候把下面看不见的 UI 挪出裁减区域、关闭主相机渲染。

CPU 优化

在我们项目开发的过程中，遇到的大部分 UI 相关性能热点都是和 CPU 消耗相关的。大致可以分为以下几类：DrawCall、Canvas.SendWillRenderCanvases()、Canvas.BuildBatch()和其他。

DrawCall 优化

DrawCall 是 CPU 调用底层图形接口，频繁的调用对 CPU 性能的影响是很明显的。优化思路很简单，合批绘制。UGUI 本身的动态合批机制会帮我们尽量的去优化合批，我们要做的就是弄清楚它的合批机制然后让 UI 元素尽量合批绘制。以下是我们项目组在制作 UI 过程中总结出来了的一些优化 Tips：

1.合理分配图集，同一个界面上的图尽量打到一个图集，多个界面复用的图，放到 common；
2.制作界面的时候，相邻节点尽量使用同一个图集的图片；

3.Text 本身也是用的 Font Texture，不同字体的 Text 也是来自不同的图集，所以在布局界面的时候也要尽量避免穿插打断绘制流程；
4.DrawCall 的数量不是完全由 Hierarchy 的布局决定，和 UI 的位置也有关系，这个位置不是指的 Rectranform 上面的 size 位置重叠就一定打断绘制，而是真实的三角面的位置是否重叠。这个可以在 Scene 视图下用线框模式(Texture Wire)去观察；
5.少用 Mask 组件，Mask 实现的原理是 Stencil Buffer，往模版缓存里绘制，模版缓存里的东西才是可见的。模板缓存会打断所有的合批，Mask 的子节点和外面的节点无法合批，模板缓存自己占一个 DrawCall。Unity5.2 之后的版本建议使用 2D Rect Mask 替代。

Canvas.SendWillRenderCanvases()

真机 Profiler 会经常看到这个函数的消耗飙高，研究 UGUI 源码得知该 API 为 UI 元素（Graphic 子类）自身发生变化（比如被 Enable、顶点色变化、Text 组件的文本变化等）时所产生的调用，CPU 飙高大部分情况下主要是因为 UnityEngine.UI.Graphic.UpdateGeometry()重新生成了所有的顶点数据，具体有哪些操作会导致 Graphic 重新生成顶点数据，可以搜索 UGUI 的源码(4.6.9) Graphic.SetVerticesDirty()。刷新顶点数据的消耗和当前 Graphic 的顶点数正相关，所以接下来研究一下 UI 顶点数。对于 UI 来说，顶点数量主要是由 Image 控件和 Text 控件产生的。

对于 Image 控件，顶点的数量取决于 ImageType。
1.Simple 模式只有 4 个顶点；
2.Sliced 模式，就是我们常用的九宫格，要分两种情况：勾选了 fillcenter 的顶点数是 36 个，没勾选 fillcenter 的顶点数是 32 个；
3.Tiled 模式，就是平铺模式，这个定点数量就完全取决 Image 控件的 Rectranform 设置的大小和原图大小，简单来说就是铺开了 N 张图就是 4*N 个顶点；
4.Filled 模式，一般用来做进度条、技能 cd 转圈等效果，这个顶点数就取决于 Fill Method 和进度值，组合的情况比较多，我就不展开算了，具体数值大家感兴趣的自行 demo 看下，总之不少于 4 个。
由此可以得出结论，**Image 能用 Simple 模式的尽量用 Simple 模式。**在项目组中经常遇到的一种情况是，以前的需求是这个 Image 用九宫格，后来需求变

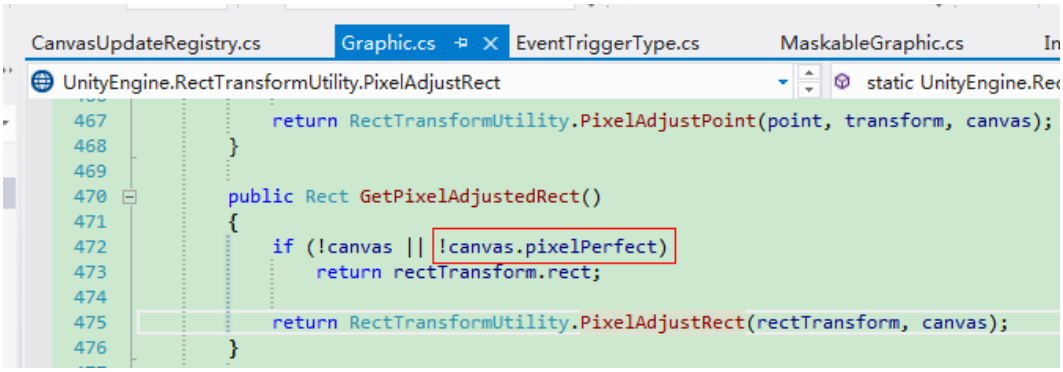
动不需要九宫格了，但是没有把 Image Type 切换会 Simple 从而导致多余的顶点数。

其实 UI 顶点数的重灾区主要是在 Text 控件，Text 在不添加任何效果的情况下是一个字符串是 4 个顶点，单独添加 Shadow 后一个字符串顶点数是 8 个，单独添加 Outline 后一个字符串顶点数是 20 个，Shadow 和 Outline 都加一个字符串顶点数是 40 个。如果界面上有 N 个字符串，那顶点数就是 N*上面的系数，这个量级是很可怕的。

因此对于 Text 控件尽量不要去添加 Shadow、Outline 效果，特别是 Outline！

这对 CPU 和 GPU 都很伤。策划和美术想要的文字凸显的效果可以用背景和文字配色来实现，效果实在无法接受的可以少量使用 Shadow。**对于频繁变动的 Text 文本（例如聊天），更要慎重考虑是否使用 Shadow、Outline。**

除了优化 UI 顶点数以外还有一些设置是会影响此函数消耗的，查看源码得知如果勾选了 Canvas 下的 pixelPerfect，在刷新顶点的时候会做一些额外的操作 RectTransformUtility.PixelAdjustRect，像素对齐调整，此操作很费，**所以不要勾选 Canvas 下的 pixelPerfect**

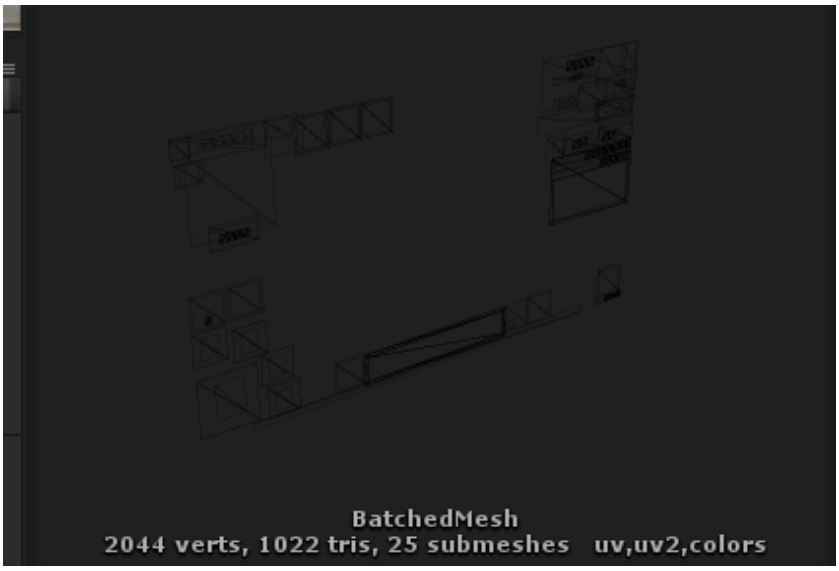


Canvas.BuildBatch()

理解该函数的消耗之前先来说一下 UGUI 的动态合批机制(减 DrawCall)，UGUI 是以 Canvas 为合批的根节点，首先 Canvas 与 Canvas 之间是没法合批的，Canvas 下所有节点的顶点数据会生成一张大的 BatchedMesh，再根据材质纹理、渲染顺序等把这个 Canvas 下能合批的 Mesh 生成 SubMesh，一个 SubMesh 就是一个绘制批次。每当这个大的 BatchedMesh 中任意一个顶点数据发生变化的时候，UGUI 目前的实现方式很粗暴，就会直接重新生成 BatchedMesh，Canvas.BuildBatch() 干的就是这个事情。由此可见 Canvas.BuildBatch()消耗飙高的因素有两点：**Canvas 下的顶点数量多和 Canvas 下的顶点发生变化。**

通常之前所提到的 Canvas.SendWillRenderCanvases()的调用都会引起 Canvas.BuildBatch()的调用。

理解了该函数的原理之后，优化的思路就清晰了：减少一些频繁变动的 UI 元素的 Canvas 下的顶点数量。关于优化 UI 顶点数量上面说的比较多了，接下来重点说下顶点变化。先来看下上面提到的 BatchedMesh



该 Mesh 包含了顶点色，所以修改顶点色、缩放、位移等导致 Mesh 数据变化的操作，都会引起 Canvas.BuildBatch()。游戏中难免有 UI 频繁移动、缩放、变颜色的需求，既然一定要变动，那我们就让这些频繁变动的 UI 元素单独放在一个 Canvas 下，且尽量减少这个频繁变动的 Canvas 下的顶点数量。简单的概括一句优化思路就是：**动静分离。**

当然如果 Canvas 分离得太细也有另外一个问题，DrawCall 数量提升，所以要做一个权衡。

其他 UI 卡顿优化建议

除了以上三点，UI 方面造成 CPU 消耗飙高的原因还有很多，就不一一展开详细解释了。

1.精简拆分 UI Prefab 文件，把打开界面瞬间看不见的 UI 元素尽量拆分，功能逻辑触发其他 UI 元素时再动态加载，例如：背包、练鼎根据子页签内容拆分，这样可以减少打开 UI 时卡顿的感觉；

2.对于 ScrollRect 里存在非常多元素的界面，避免一次全部 Initiation，回收重复里节点元素，刷新数据即可（可以实现一个公用组件）；

3.避免频繁 Initiation、Destroy 某个 UI 子元素，类似小地图的 SceneActor 圆点、寻路路径、YLIconBox 的子节点，使用对象池维护起来重复利用；
4.序列化保存Prefab的时候,尽量让active的状态和大部分情况下出现的active状态一致，可以避免切换 active 状态带来的性能消耗；
5.在 UI 布局位置不会动态调整的情况下，不要偷懒用 Layout 组件实现自动布局，直接 Rectranform 写死坐标位置即可，Layout 动态计算有开销；
6.UGUI 对比 NGUI 有个不好的地方是所有的 Graphic 组件默认都是参与接受射线检测，响应 EventSystem 事件的，这样在在长时间按屏幕输入的时候参与遍历检测的 UI 节点会很多，界面复杂到一定程度会导致 EventSystem.Update 消耗飙升。建议用 5.2 之后版本的可以让所有没有响应需求的 Graphic 组件 Raycast Target 默认不勾选，4.x 也是可以添加 Canvas Group 然后去掉 Block Raycasts 和 Interactable；
7.不要使用 Text 的 Best Fit 选项，有额外开销；
8.对部分 UI 有频繁显示和隐藏需求的，建议不要直接调用 SetActive，上面有提到这样会导致顶点数据重新生成，过于频繁调用有性能问题。有两个建议：挪出裁减区域或者禁用 CanvasRender，具体采用什么方案要看需求，因为禁用 CanvasRender 还是会响应事件检测。

《穿越火线：枪战王者》客户端技术方案

腾讯互娱专家工程师-郭智

一、项目背景

CF 手游的团队有着相当丰富的 FPS 游戏制作经验，但是移动端开发经验相对匮乏。团队面对的挑战很大，我们需要在手机端完美还原 CF 十多个游戏模式，上百把枪械手感。

虽然我们有实时对战 FPS 游戏开发经验，但是手游网络质量很差，我们需要保证在高 Ping 值，高丢包率的弱网络环境下流畅同步，并且避免外挂的出现。CF 端游用户量很大，手游也会是一个高在线的游戏。对于后台我们希望能大幅度地提高负载，节省成本。

二、解决方案

1. 移动端实时同步技术

1.1 CF 手游的 C/S 架构

CF 手游是竞技向的手游，为了杜绝外挂，弱网络下有良好表现，注重游戏的时效性，最终选择基于 C/S 架构使用可靠 UDP 作为网络层的同步方案：

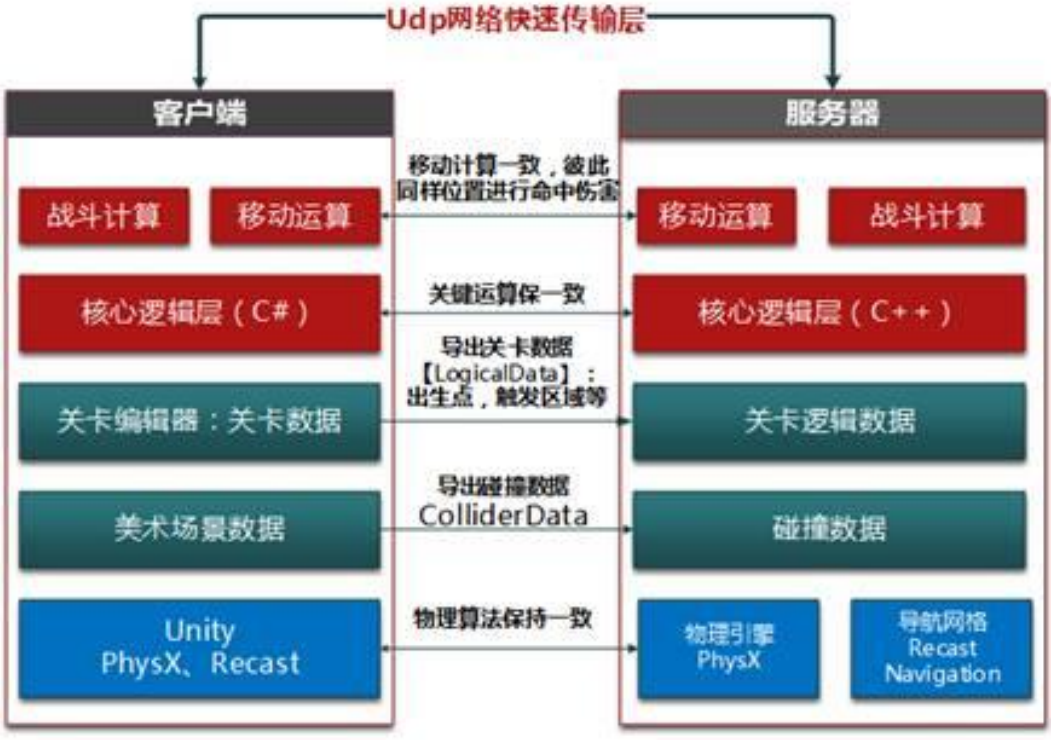


图 CF 手游网络架构

引擎层：Unity 的物理引擎是 Physx。为了保证两边物理算法一致，我们搭建了基于 Physx 的物理服务器，并对其作出优化，同时集成了 Recast Navigation 等高性能引擎组件为服务器提供基础功能。

数据层：基于我们的关卡工具导出碰撞模型和关卡数据提供给服务器，保证数据一致。

游戏逻辑层：客户端使用 C#编写，服务器为了高性能使用 C++，保证核心运算一致。客户端和服务器都会维护一致的逻辑实体，两边都做移动计算，服务器会在同样的位置进行命中伤害计算，并且通知客户端做表现。服务器永远是权威，客户端是哑终端，借此达到高强度的反外挂。

整体 C/S 架构基于我们实现的 UDP 网络层进行快速传输。

1.2 高性能的 Dedicated Server 与服务器负载优化

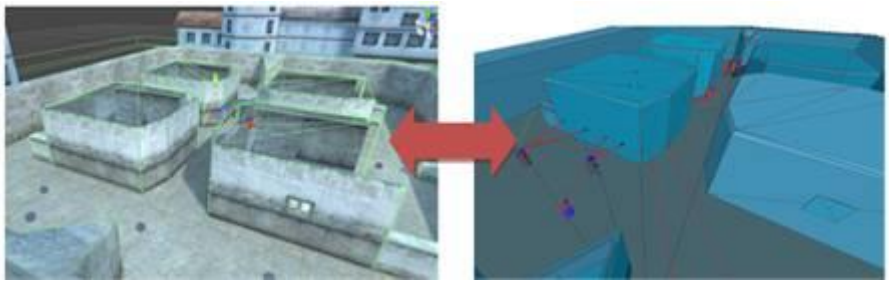


图 CF 手游服务器物理世界还原

CFM 的即时对战服务端（DS）是依赖虚幻 3 引擎开发过程中积累的经验，完全为手游重写的。DS 基于 PhysX 和 Recast 开源组件开发，对程序性能有良好的控制。在腾讯标准 M1 机型配置下，单机综合承载可以达到 8000 人。DS 反外挂强度接近端游，单机性能比同类端游提高 7-8 倍。

1.3 高时效的 UDP 网络层

游戏对网络有实时性要求高，带宽要求小的特点。TCP 虽然提供了可靠传输，但是内置的复杂拥塞控制算法并不是专为实时性优化的，也没有提供较好的方法让业务定制化。CF 手游实现了自定义的 RUDP 协议方案，使用 UDP 保证协议实时性，同时通过自定义重传策略兼顾可靠性，取得了很好效果，主要技术要点包括：

数据传输分类

CF 手游中并非所有数据都要求可靠，按游戏逻辑需要，只有不到 50%的协议有

可靠性需求，RUDP 中只对这部分协议提供可靠性保证。而非可靠包则保证尽快到达，以满足游戏的实时性需求。对玩家移动状态等信息，由业务层定时冗余重传。

图 CF 手游网络数据传输分类

客户端与服务器的交互时序如下：

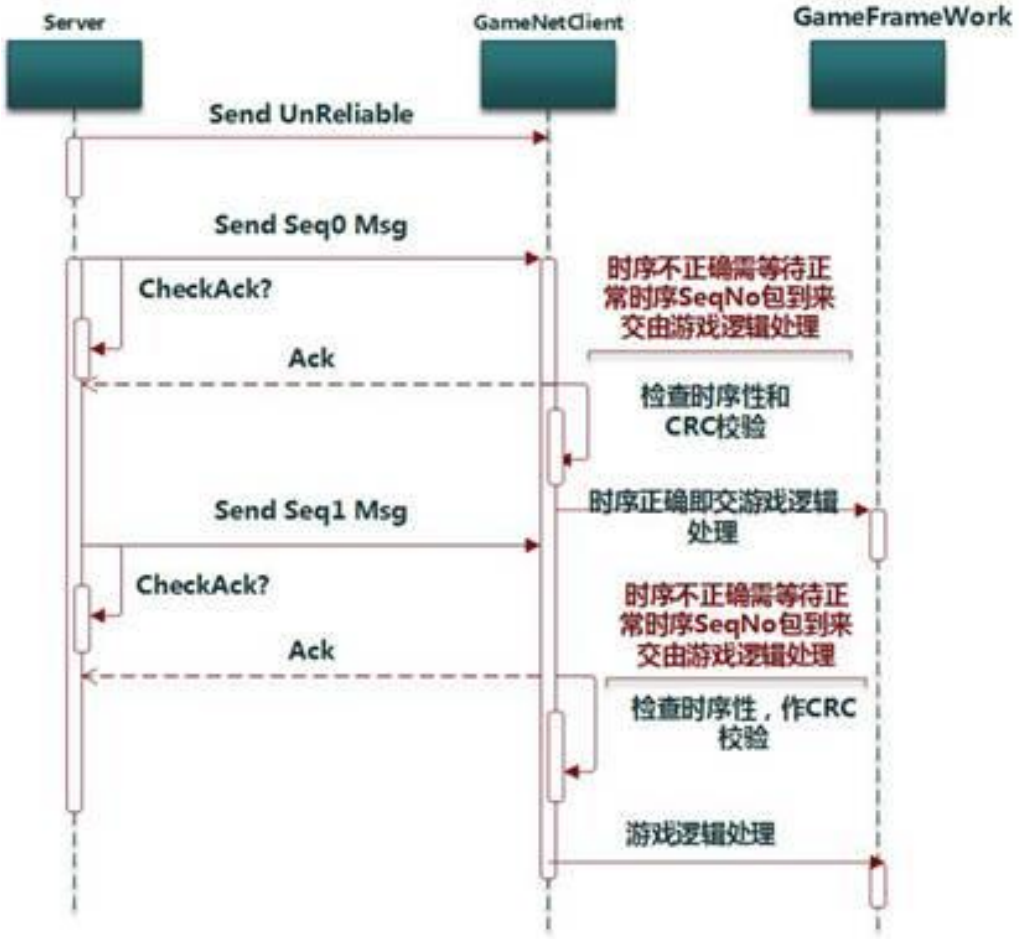


图 CF 手游网络数据交互时序

快速发送

发送方不维护发送窗口，不等待前面包是否 ack，有数据需要发送时立即发送。

快速重传

CF 手游使用比 TCP 更高精度，响应速度更快的重传策略以保证实时性。

	TCP实现	CF实现	优势
重传定时精度	0.25s*	1ms	可以根据网络波动，灵活调整超时重传策略
RTO	防拥堵，慢重传	平方间隔 根据历史值调整	快速重传
超时重传策略	只会重传最早的未ACK的包	所有发过的包都独立重传	提高实时性，避免前面的包丢失导致后面的数据包被延迟传输

表 RUDP 快速重传策略

带宽优化

主要思路是有损服务和降低不必要开销。CFM 持续进行了多轮流量优化，包括：

1.MTU 设计为 500+字节，应用逻辑保证数据包大小不超过 MTU，避免拆包。

2.减小包头，8 字节。

3.小包合并。同一帧发往同一个目标的多个小数据包合并为大包，减少包数量。

4.降低服务端帧率和位置精度，但不影响玩家体验。

1.4 弱网络移动同步方案

PVP 实时玩家同步

CFM 采用 C/S 架构，服务器上会执行完整的游戏逻辑。在移动同步上，玩家的位置也是以服务器的位置为准，客户端 1P 视角和 3P 视角分别进行同步和模拟。

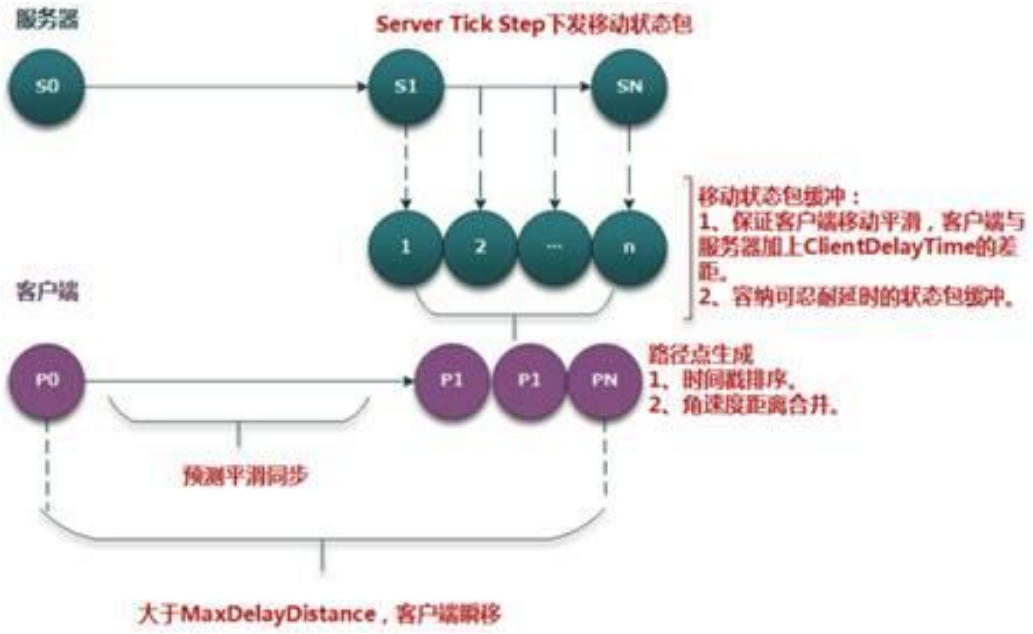


图 CF 手游玩家实时同步方案

如上图所示，客户端 1P 根据玩家的输入调整移动方向和速度，并把相关信息上报到服务器。服务器对上报的信息进行合法性检验之后更新玩家的移动状态，并在每个逻辑帧下发移动信息到所有客户端。客户端在收到服务器下发的移动信息，使用适当的同步算法对玩家的移动进行模拟，达到实时流畅的同步效果。其技术要点如下：

客户端 1P 即时预表现，避免网络延迟带来的操作手感迟滞的问题。

客户端在接受玩家输入后，会即时更新本地的移动状态，并上报服务器。在收到服务器回包之前，客户端 1P 就会按照新的状态进行移动，保证玩家的操作在第一时间得到响应。客户端与服务器物理世界的一致性则保证了客户端 1P 预表现的结果和服务器的移动结果是一致的。

客户端 3P 进行延迟补偿和预测同步，消除网络因素引入的位置误差。

在 3P 的同步模拟上，CFM 参考了 DR 算法（Dead Reckoning）和影子跟随算法等方案，并针对移动弱网络的特点，加入了延迟补偿和预测同步的特性，解决了在网络抖动的情况下同步位置差距大和移动过程卡顿等关键问题。客户端收到服务器的移动信息后，会对服务器的移动路点进行过滤和合并，然后构造出 3P 同步模拟的移动路径。同时，客户端会根据当前与服务器的位置差距以及当前的网络延迟对模拟移动时的运动速度进行补偿，从而保证客户端 3P 经过

SimulateTime 时间的模拟移动后与服务器的位置达到一致。

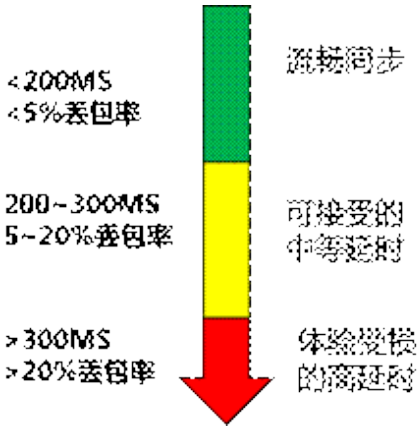
另外，在移动弱网环境下，网络抖动会导致游戏过程中网络瞬时延迟变大，客户端在 3P 同步模拟过程中可能出现服务器下发的移动信息丢失或者延迟到达的问题。在这种情况下，3P 模拟的移动过程就会被打断，出现走走停停的卡顿现象。为此，CFM 在同步模拟的过程中加入了预测同步，在没有收到服务器信息的情况下，会根据玩家之前的移动状态继续预测移动，预测时间为 2 倍的网络延迟。在预测移动过程中，如果收到来自服务器的移动信息，则会自动恢复为正常的同步模拟过程，从而保证了 3P 同步模拟过程中移动的流畅性和连续性。

服务器延时、丢包补偿

丢包时，服务端可能由于移动路径不连贯而卡在拐角处，需要有补偿机制。因为传输延迟，从客户端向服务器发起移动请求，到服务器开始模拟有时间差。CF 手游服务端需要回溯到移动包的历史时间，再追到当前时间。这样允许最多减少 100-200ms 内的 C/S 不同步现象。另外当客户端移动路径部分丢失后，服务端根据历史数据，尝试推算可能的移动路径。通过尝试绕路策略避开障碍，在较小范围内近似模拟客户端路径。

应用平滑插值和物理力学。

在移动弱网环境出现抖动时，服务器的移动信息可能会延迟到达或者扎堆到达，导致客户端 3P 同步模拟过程中速度陡变，移动过程忽快忽慢等问题。因此，CFM 对同步模拟过程中速度的变化进行了平滑插值，保证速度的变化符合玩家对现实世界变速的认知，提高游戏的真实感。另外，在进行大角度变向过程中，根据物理力学规律，玩家角色应该遵循先减速再逆向加速的物理过程。CFM 在同步模拟过程中对玩家的此类行为进行了分析判定，并实现了相应的物理模型，避免玩家在变向移动时速度过快难以命中的问题，增强了游戏体验。



PVE 大规模怪物同步

CFM 的多人 PVE 模式以末日逃生，僵尸围城为设定，为了更好地表现尸潮围堵的感觉，提高玩家闯关过程中的爽快感，必须支持大规模的怪物行为同步。面对大量怪物带来的流量消耗及客户端性能压力，CFM 针对 PVE 怪物行为的特点重新设计了低误差，低流量消耗，支持复杂行为的同步方案。

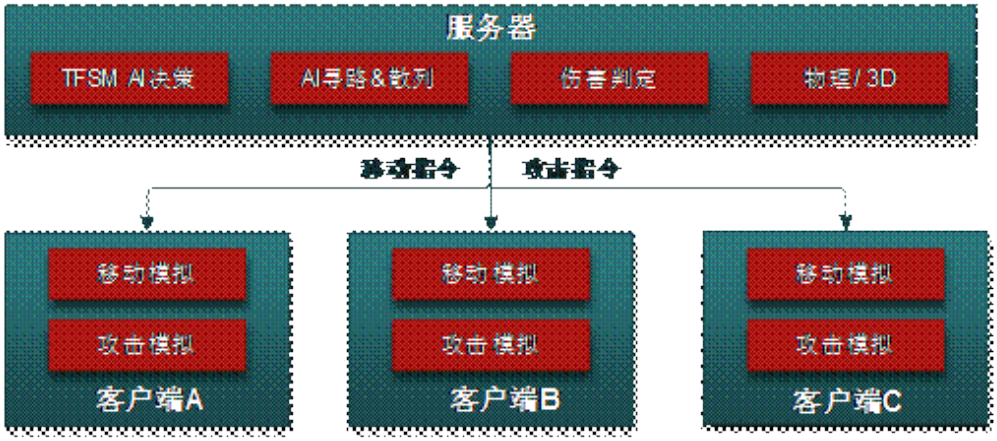


图 CFM 指令式怪物 AI 同步方案

支持上百 AI 单位的实时同步，核心问题：

服务器伤害逻辑与客户端行为表现一致性；

怪物群体行为与移动同步；

大规模怪物同步流量控制；

AI 命令按需发送，降低流量消耗。

在 CFM 的多人 PVE 模式中，怪物的移动信息只会在怪物移动路径变更时进行同步，而不是传统方案中的每帧同步。怪物路径更新的频率约为 1~2 秒/次，其同步流量消耗只有传统方案的 1/8 左右。

通用怪物行为建模，对复杂怪物行为进行抽象。

CFM 中怪物种类多样，怪物行为特别是 BOSS 技能流程复杂，包括各种连续技和组合技。CFM 怪物同步方案中对怪物的行为进行了抽象，把怪物的行为规范到统一的流程中，通过对服务器下发的技能指令进行解释，生成客户端技能状态机，驱动怪物的复杂行为表现。

PVE 大量怪物同步：群体行为

难点：

Ø 常规寻路导致怪物位置重叠;

Ø 纯客户端方案导致位置差异大;

解决：

Ø 服务器计算怪物群体力，构造移动路径;

Ø 客户端根据下发路径进行移动模拟;

效果：

Ø 保证服务器与客户端路径一致;

Ø 路径合包下发，节省流量;

Ø 避免客户端寻路与碰撞计算，提高性能;

2. 移动 FPS 游戏手感

2.1 移动端操作手感

FPS 竞技类游戏，对玩家操作的反应和准确性要求极高，如何在移动端实现良好的操作手感是个很大的挑战。我们从移动、瞄准和射击三个维度进行了深度探索和优化，不断尝试了几十种手感方案，上百套参数调整，最终得到了目前的操作手感解决方案。

适配多样屏幕的操作数学模型

我们获取玩家的触屏输入 Touch，通过计算 Touch 的位置和 DeltaPosition，转换为角色的移动加速度和视角转向角度等。

由于屏幕的密度和手机分辨率各不一样，相同的滑屏物理距离会得到不一样的 DeltaPos，因此 CFM 在操作模型计算中加入了移动设备的 dpi 和分辨率因子来解决一致性问题。

抖动手感数学模型

抖动的主要因素有两个：一是触摸的不精确性，另一个是游戏帧率的抖动。触摸的不精确性，是由于我们手指与屏幕的接触是一块区域，但程序中获取的却是一个像素点，并且各个系统的实现也各不一致。

3D Touch

在苹果推出的新一代多点触控技术 3D Touch 后，我们立刻进行了相关的技术预研，最终实现了一种能完美解决操作体验的手感方案，也成为了首款支持 3D Touch 的 FPS 手游，并申请技术专利。我们在跟随开火的基础上，加入了按压力度来触发开火（即 3D Touch）。在 iOS 上，我们通过 Hook UIKit 的事件分发，从而把按压力度转发到 C#层来实现。

图 CF 手游 3D Touch

2.2 枪械射击手感

作为经典第一人称射击游戏 CF 在移动平台上的延续，CFM 力求为玩家还原最真实的枪械手感。CFM 结合移动客户端上玩家的操作方式进行优化，创新性的设计自身的枪械弹道数学模型，此模型成为射击游戏的标杆：

T 字弹道模型

枪械射击时，子弹射出的最终方向主要由三个因素决定，一是枪射当前瞄准的方向，即玩家希望射击的位置；二是枪械后座力引起的枪口上抬和左右偏向，形成 T 字形的主弹道；三是枪管震动和环境因素对子弹弹道形成的轻微扰动，即子弹弹道的散发。

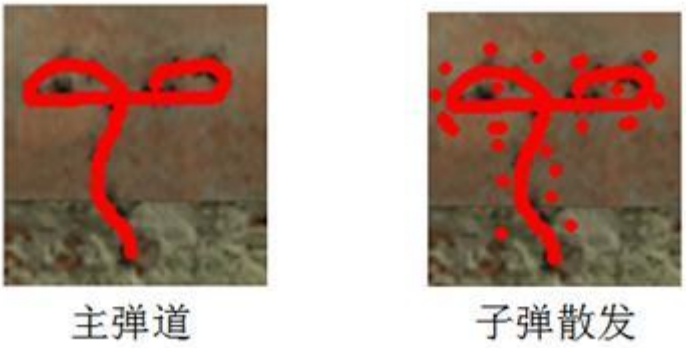


图 CF 手游枪械弹道模型

枪械后座力模型

枪械每次射出的子弹的反作用力都会导致枪口上抬和左右偏向，上抬和摇晃的幅度与枪械当前连发数成正比，枪口随着连发数的增加逐步上抬，到达最大上抬幅度后转为随机方向的左右摆动。

在枪械停止射击后，游戏模拟枪口位置在玩家控制下从上抬位置恢复到初始位置，恢复的过程使用弹簧模型，遵循胡克定律。

子弹弹道散发模型

现实中子弹射出的时候受各种环境因素影响，会出现小幅度的随机偏向，子弹的最终射出方向即是在后座力计算后的前向方向叠加上散发的偏差，弹道散发的幅度和方向符合高斯正态分布。因此 CFM 在实现枪械散发模型时也使用高斯随机数实现。

创新辅助瞄准机制

CFM 在射击上创新性地设计了一套辅助瞄准机制，帮助玩家自动瞄准静止和低速运动的目标，降低新手玩家的操作门槛，提高新手玩家的乐趣。同时，辅助瞄准在面对快速移动和转向目标时将失去效用，以此保证高手玩家的技术优势，提高玩家对技术提升的追求。

弱联网优化之道

腾讯互娱专家工程师-樊华恒

一、移动网络的特点

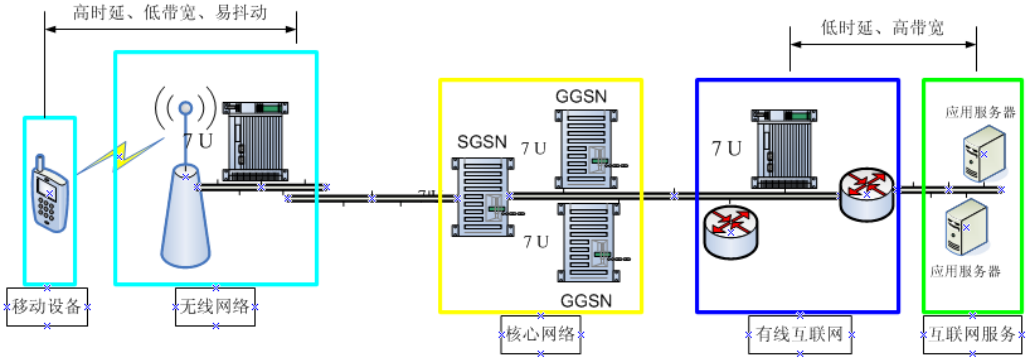
我们看到移动网络和移动互联网时代用户的行为有如下三个典型特点：

1）移动状态网络信号不稳定，高时延、易抖动丢包、通道狭窄；

2）移动状态网络接入类型和接入点变化频繁；

3）移动状态用户使用高频化、碎片化、非 WIFI 流量敏感；

为什么？参考【图一 无线网络链路示意】，我们尝试从物理上追根溯源：



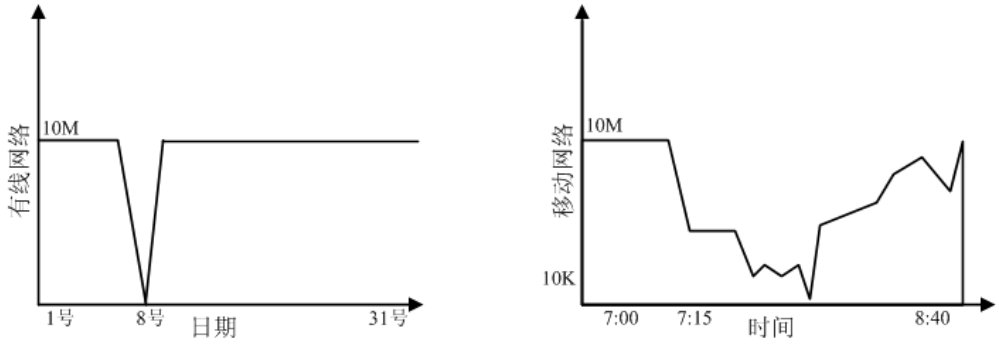
【图一 无线网络链路示意】

第一、直观印象是通讯链路长而复杂，从（移动）终端设备到应用服务器之间，相较有线互联网，要多经过基站、核心网、WAP 网关（好消息是 WAP 网关正在被依法取缔）等环节，这就像送快递，中间环节越多就越慢，每个中转站的服务质量和服务效率不一，每次传递都要重新交接入库和分派调度，一不小心还能把包裹给弄丢了；

第二、这是个资源受限网络，移动设备接入基站空中信道数量非常有限，信道调度更是相当复杂，如何复杂就不展开了，莫文蔚那首歌词用在这里正合适：“我讲又讲不清，你听又听不懂……”，最最重要的是分配的业务信道单元如果 1 秒钟不传数据就会立马被释放回收，六亲不认童叟无欺；

第三、这个链条前端（无线端）是高时延（除某些 WIFI 场景外）、低带宽（除某些 WIFI 场景外）、易抖动的网络，无线各种制式网络带宽上限都比较低而传输时延比较大（参见【表一 运营商移动信号制式带宽标准】），并且，没事就能丢个

包裹玩玩，最最重要的是，距离基站的远近，把玩手机的角度、地下室的深度等等都能影响无线信号的质量，让包裹在空中飞一会，再飞一会。这些因素也造成了移动互联网网络质量稳定性差、接入变化频繁，与有线互联网对比更是天上人间的差别，从【图二 有线互联网和移动互联网网络质量差异】中可以有更直观的感受；



【图二 有线互联网和移动互联网网络质量差异】

【表一 运营商移动信号制式带宽标准】

数据来自互联网各种百科，定性不定量，仅供参考；

移动运营商	通信标准	信号制式	备注
中国移动	2.5G	GPRS	属 2G 网络，基于 GSM，理论峰值速率 172 Kbps（实际最高 115 Kbps），图标“G”，时延 300 ~ 500ms；
中国移动	2.75G	EDGE	属 2G 网络，基于 GSM，理论下行峰值速率 236 Kbps、上行峰值速率 118 Kbps，图标“E”，时延 200 ~ 400ms；
中国移动	3G	TD-SCDMA	属 3G 网络，理论下行峰值速率 1.6 Mbps、上行峰值速率 1.6 Mbps，图标“T”，时延 100ms；
中国移动	3.5G	HSPA	属 3G 网络，基于 TD-SCDMA，包括 HSDPA 和 HSUPA，理论下行峰值速率 2.8 Mbps、上行峰值速率 2.2 Mbps，图标“H”，时延 100ms；
中国移动	4G	TD-LTE	属 4G 网络，理论下行峰值 100 Mbps，上行峰值 50Mbps，时延 10 ~ 50ms；
中国电信	2G	CDMA 1X	属 2G 网络，基于 CDMA，理论下行峰值速率 153Kbps、上行峰值速率 153 Kbps，图标“1X”，时延 500 ~ 600ms；

移动运营商	通信标准	信号制式	备注
中国电信	3G	EDVO	属 3G 网络，基于 CDMA 2000，理论峰值下行 3.1 Mbps、上行 1.8 Mbps，图标“3G”，时延 100ms；
中国电信	4G	TD-LTE FDD-LTE	均属 4G 网络，TD-LTE 理论下行峰值速率 100 Mbps，上行峰值 速率 50Mbps，FDD-LTE 理论下行峰值速率 150 Mbps，上行峰值速率 40 Mbps，时延 10 ~ 50ms；
中国联通	2.5G	GPRS	属 2G 网络，基于 GSM，理论峰值 114 Kbps，图标“G”，时延 300 ~ 500ms；
中国联通	2.75G	EDGE	属 2G 网络，基于 GSM，理论下行峰值速率 236 Kbps、上行峰值速率 118 Kbps，图标“E”，时延 200 ~ 400ms；
中国联通	3G	HSPA	属 3G 网络，基于 WCDMA，包括 HSDPA 和 HSUPA，理论下行峰值速率 14.4 Mbps、上行峰值速率 5.76 Mbps，图标“H”，时延 100ms；
中国联通	4G	TD-LTE FDD-LTE	均属 4G 网络，TD-LTE 理论下行峰值速率 100 Mbps，上行峰值 速率 50Mbps，FDD-LTE 理论下行峰值速率 150 Mbps，上行峰值速率 40 Mbps，时延 10 ~ 50ms；
N/A	WIFI	WLAN IEEE 802.11a/b/g	论峰值速率 54 Mbps，时延 10 ~ 50ms；
N/A	WIFI	WLAN IEEE 802.11n	理论峰值速率<600 Mbps，时延 10 ~ 50ms；

第四、这是个局部封闭网络，空中信道接入后要做鉴权、计费等预处理，WAP 网络甚至还要做数据过滤后再转发，在业务数据有效流动前太多中间代理人求参与，效率可想而知。产品研发为什么又慢又乱，广大程序猿心里明镜似的；最最重要的是，不同运营商之间跨网传输既贵且慢又有诸多限制，聪明的运营商便也用上了缓存技术，催生了所谓网络“劫持”的现象。

如果我们再结合用户在移动状态下 2G/3G/4G/WIFI 的基站/AP 之间，或者不同网络制式之间频繁的切换，情况就更加复杂了。

二、移动网络为什么“慢”

我们在移动网络的特点介绍中，很容易的得到了三个关键字：“高时延”、“易抖动”、“通道窄”，这些物理上的约束确实限制了我们移动冲浪时的速度体验，那么，还有别的因素吗。

当然有，汗牛充栋、罄竹难书：

2.1 DNS 解析，这个在有线互联网上司空见惯的服务，在移动互联网上变成了一种负担，一个往复最少 1s，还别提遇到移动运营商 DNS 故障时的尴尬；

2.2 链路建立成本暨 TCP 三次握手，在一个高时延易抖动的网络环境，并且大部分业务数据交互限于一个 HTTP 的往返，建链成本尤其显著；

2.3 TCP 协议层慢启动、拥塞控制、超时重传等机制在移动网络下参数设定的不适宜；

2.4 不好的产品需求规定或粗放的技术方案实现，使得不受控的大数据包、频繁的数据网络交互等，在移动网络侧 TCP 链路上传输引起的负荷；

2.5 不好的协议格式和数据结构设计，使得协议封装和解析计算耗时、耗内存、耗带宽，甚至协议格式臃肿冗余，使得网络传输效能低下；

2.6 不好的缓存设计，使得数据的加载和渲染计算耗时、耗内存、耗带宽；

现在终于知道时间都去哪了，太浪费太奢侈，还让不让人愉快的玩手机了。天下武功，唯快不破，我们一起踏上“快”的探索之路吧。

三、移动联网快的四个方法

在移动互联网时代，对我们的产品和技术追求提出了更高的挑战，如何从容和优雅的面对，需要先从精神上做好充分的准备，用一套统一的思考和行动准则武装到牙齿：

1) 不要我等，一秒响应；

2) 可用胜于完美；

3) 水到渠成，润物无声；

听起来很抽象，也不着急解释，耐心看完整篇文章再来回味，定有醍醐灌顶，昏昏欲睡之功效。

从来就没有什么救世主，只有程序员征服一切技术问题的梦想在空中飘荡。屡败屡战，把过往实践中的经验教训总结出来，共同研讨。针对移动网络的特点，我们提出了四个方法来追求极致的“爽快”：快链路、轻往复、强监控、多异步。下面逐一展开研讨。

3.1. 快链路

我们需要有一条（相对）快速、（相对）顺畅、（相对）稳定的网络通道承载业务数据的传输，这条路的最好是传输快、不拥堵、带宽大、收费少。生活中做个类比，我们计划驱车从深圳到广州，如果想当然走广深高速十之八九要杯具，首先这个高速略显破败更像省道，路况不佳不敢提速；其次这条路上的车时常如过江之鲫，如果身材不好操控不便，根本就快不起来；最后双向六车道虽然勉强可以接受，但收费居然比广深沿江高速双向八车道还贵；正确的选路方案目前看是走沿江高速，虽然可能要多跑一段里程，但是通行更畅快。实际上，真实情况要更复杂，就如同【图二 有线互联网和移动互联网网络质量差异】所示，漫漫征途中常常会在高速、国道、省道、田间小道上切换。

3.1.1. TCP/IP 协议栈参数调优

纯技术活，直接上建议得了，每个子项争取能大致有个背景交待，如果没说清楚，可以找 Google。

3.1.1.1 控制传输包大小

控制传输包的大小在 1400 字节以下。暂时不讲为什么这样建议，先举个例子来类比一下，比如一辆大卡车满载肥猪正在高速上赶路，猪笼高高层叠好不壮观，这时前方突然出现一个隧道限高标识，司机发现卡车超限了，这下咋整。方案一，停车调头重新找路，而且十之八九找不到，最后只能哪来回哪；方案二，把其中一群猪卸下来放本地找人代养，到达目的地卸完货回来再取，你别说，这个机制在 TCP/IP 协议栈中也有，学名“IP 分片”，后面会专门介绍。美国计算机科学家也曾经蹲在高速路边观察生猪超载运输的过程，并饱受启发。且慢，每次遇到问题，想到一些方案后我们都应该再扪心自问：“还有没有更好的办法呢？”。当然有，参照最近流行的说法，找个台风眼，把猪都赶过去，飞一会就到了，此情此景想想也是醉了。

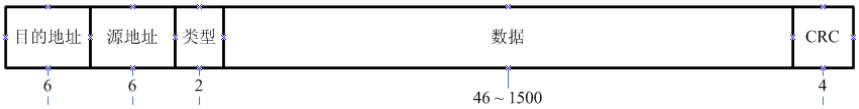
回归正题，概括地说，我们设定 1400 这个阈值，目的是减少往复，提高效能。因为 TCP/IP 网络中也有类似高速限高的规定，如果在超限时想要继续顺畅传输，要么做 IP 分片要么把应用数据拆分为多个数据报文（意指因为应用层客户端或服务器向对端发送的请求或响应数据太大时，TCP/IP 协议栈控制机制自动将其拆分为若干独立数据报文发送的情况，后面为简化讨论，都以 IP 分片这个分支为代表，相关过程分析和结论归纳对二者均适用）。而一旦一个数据报文发生了

IP 分片，便会在数据链路层引入多次的传输和确认，加上报文的拆分和拼接开销，令得整个数据包的发送时延大大增加，并且，IP 分片机制中，任何一个分片出现丢失时还会带来整个 IP 数据报文从最初的发起端重传的消耗。有点枯燥了，我们从一些基础概念开始逐步深入理解：

a. 【以太网】

这个术语一般是指数字设备公司（Digital Equipment Corp.）、英特尔公司（Intel Corp .）和 Xerox 公司在 1982 年联合公布的一个标准，它是当今 TCP/IP 采用的主要网络技术。以太网采用一种称作 CSMA/CD 的媒体接入方法，其意思是带冲突检测的载波侦听多路接入（Carrier Sense, Multiple Access with Collision Detection）。随着以太网技术的不断演进，传输速率已由最初的 10 Mb/s 发展到如今 100Mb/s、1000Mb/s、10000Mb/s 等。

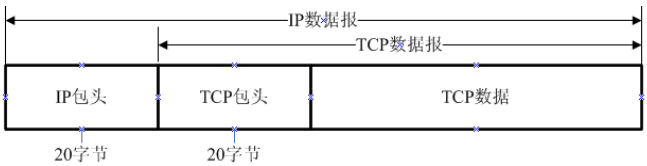
我们现在使用的 TCP/IP 网络协议，基本上都在以太网上传输，数据被封装在一个个以太网包中传递，这些以太网包就是那一辆辆运猪的大卡车。以太网包的封装格式可以参考【图三 以太网的封装格式（RFC 894)】，很容易看出以太网包能传输的有效“数据”大小在 46 ~ 1500 字节之间。如果我们把以太网看做是运猪的高速公路，能承载有效数据的最大值看作是高速路上隧道的限高，那么这个限高在 TCP/IP 协议中学名是 MTU（Maximum Transmission Unit，最大传输单元)。MTU 属于链路层制定的逻辑（并非物理）特性限制，所谓无规矩不成方圆。



【图三 以太网的封装格式（RFC 894)】

b. 【TCP/IP 数据报】

TCP/IP 数据报被封装在以太网包的“数据”中，通过【图四 TCP 数据在 IP 数据报中的封装】可以看到，一个 IP 数据报包括 IP 包头、TCP 包头和 TCP 数据三个部分，其中两个包头分别用于 IP 层和 TCP 层的报文传输控制，可以理解为运猪的大卡车和猪笼。TCP 数据则是有效载荷，可以理解为那群肥猪。

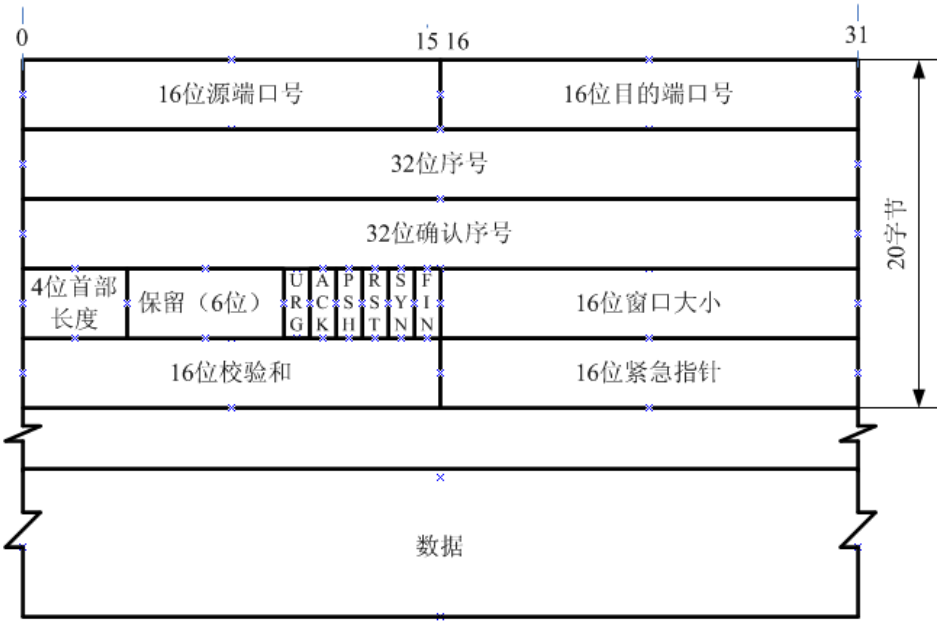


【图四 TCP 数据在 IP 数据报中的封装】

我们再来详细看看 IP 数据报，如【图五 IP 数据报格式及首部中的各字段】所示，一个标准 IP 数据报中，IP 包头大小为 20 字节，如果加上可选项，则 IP 包头最大可以达到 60 字节。

【图五 IP 数据报格式及首部中的各字段】

TCP 数据报如【图六 TCP 数据报格式及首部中的各字段】所示，一个标准 TCP 包头大小为 20 字节，如果加上可选项，则最大也可以达到 60 字节。



【图六 TCP 数据报格式及首部中的各字段】

c. 【TCP MSS】

TCP MSS（TCP Maximum Segment Size，TCP 最大报文段长度，后面均简称 MSS）表示 TCP/IP 协议栈一次可以传往另一端的最大 TCP 数据长度，注意这个长度是指 TCP 报文中的有效“数据”（即应用层发出的业务数据）部分，它不包括 TCP 报文包头部分，我们可以把它理解为卡车能装运生猪的最大数量或重量。它是 TCP 选项中最经常出现，也是最早出现的选项，占 4 字节空间。

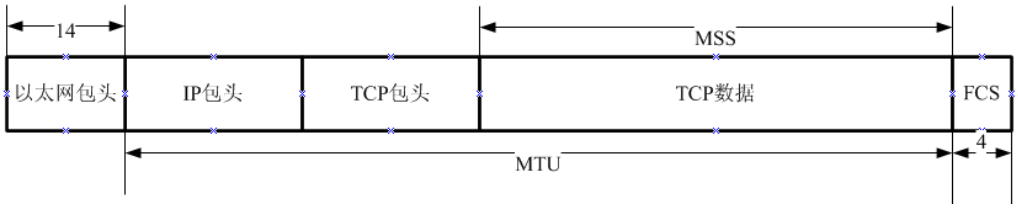
MSS 是在建立 TCP 链接的三次握手过程中协商的，每一方都会在 SYN 或 SYN/ACK 数据报文中通告其期望接收数据报文的 MSS（MSS 也只能出现在 SYN 或 SYN/ACK 数据报中），说是协商，其实也没太多回旋的余地，原因一会讲。如果协商过程中一方不接受另一方的 MSS 值，则 TCP/IP 协议栈会选择使用默认值：536 字节。

有了以上的基础知识，我们就能比较清晰的描述出以太网、MTU、TCP/IP 数据报文和 MSS 之间的关系了，如【图七 TCP/IP 数据报、MTU/MSS 在以太网格式中的关系】所示，MTU 和 MSS 关系用公式表达就是：

MTU = IP 包头 + TCP 包头 + MSS；

对照到我们肥猪装运的例子，自然得出公式如下：

限高 = 卡车高度 + 笼子高度 + 生猪数量或重量；



【图七 TCP/IP 数据报、MTU/MSS 在以太网格式中的关系】

注：FCS（Frame Check Sequence）是指帧校验值；

实际上 MSS 值太小或太大都不合适。

太小比如设为 1 字节，那么为了传输 1 个字节的数据，得搭上 IP 包头的 20 字节和 TCP 包头的 20 字节，如果再加上链路层、物理层的其它开销，显然效率低下不够环保，这就如同卡车跑一趟只拉一头肥猪一样，相当坑。

MSS 是不是越大越好呢，这也符合我们的正常思维逻辑，就好比养猪场和买家都希望卡车一趟能多运几头肥猪，可以加快资源周转效率。但实际情况是 MSS 如果设得太大，封装的数据过多，不但传输时延会增加，还很可能因为超过 MTU

的限制，使得在 IP 层传输过程中发生分片（又是它，忍着，马上就会展开了），接受方在处理 IP 分片包所消耗的资源和处理时间都会增大，前面也提到过，如果 IP 分片在传输中出现分片丢失，哪怕只是丢失一个分片，都会引起整个 IP 数据报的重传，这是因为 IP 层本身没有设计超时重传机制，有兴趣可以研读《TCP/IP 详解 卷一：协议》了解详细细节。由此可以想见网络开销会因此大大增加。
TCP/IP 协议设计者是不希望分片出现的，现在有点明白前面说 MSS 协商回旋余地不大的含义了吧。另外，MSS 同滑动窗口和拥塞控制也有关联，后续谈到相关话题时我们再细聊。

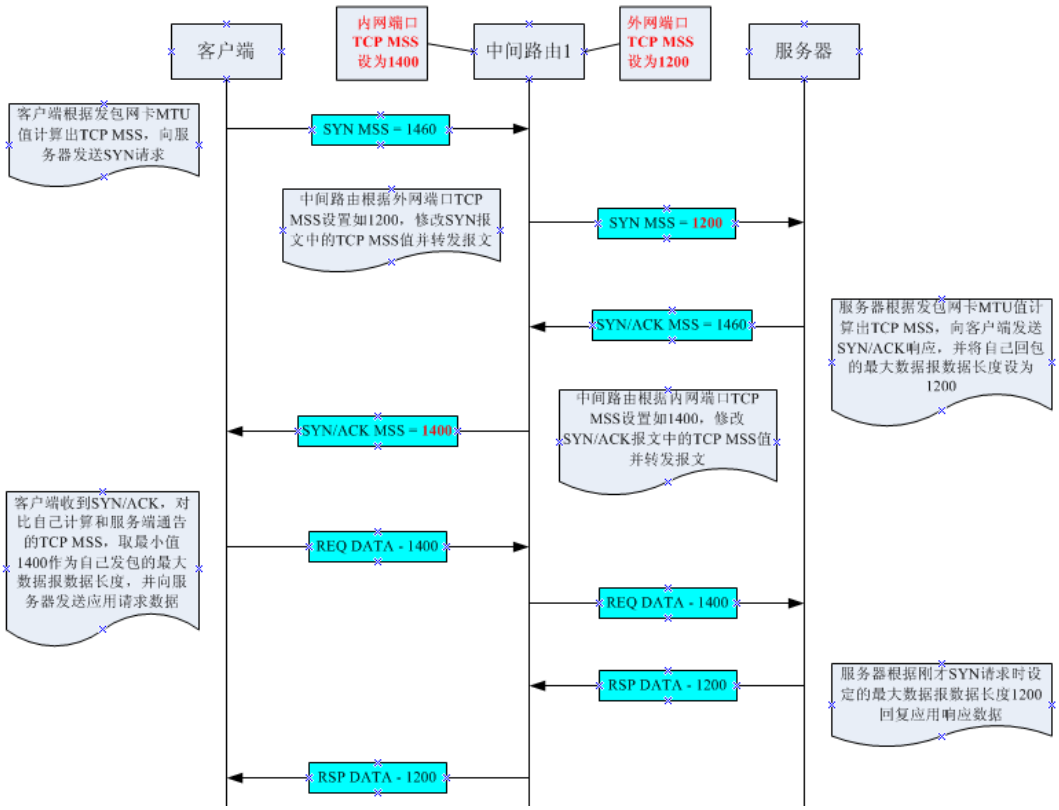
d. 【IP 分片】
快乐运猪路遇限高紧急应对方案二闪亮登场，IP 数据报文传输过程中，任何传输路径上节点的 IP 层在接收到一份要发送的 IP 数据报文时，首先会通过路由选路判断应从本地哪个网络接口把 IP 数据报转发出去，随后查询获取该网络接口的 MTU，如果 IP 数据报文长度超过了这个 MTU，且该数据报文没有设置 DF（Don't Frament，不要分片，非缺省值）标志位，就得做 IP 分片，即把接收到的 IP 数据报文拆分成多个更小（不超过该接口 MTU）的 IP 数据报文继续传输，并且，分片的数据可能在路上会被再次分片，分片到达最终目的地后会按顺序重新组装还原，【图五 IP 数据报格式及首部中的各字段】中 3 位标志和 13 位片偏移就是用来干这个的。

为了避免 IP 分片，TCP/IP 协议设计者在 TCP 层实现了 MSS 协商机制，设想如果最终确定的 MSS 小于路由路径中最小的那个 MTU，那么就能避免 IP 分片的发生。

在 TCP 链接三次握手过程中，网络通讯的两个端点在 SYN 和 SYN/ACK 数据报文中分别把自己出口 MSS 发给对端，以便对方了解自己的“限高”水平，最终控制发出的应用数据报文大小，达到避免 IP 分片的目的。

如果运气好，路由路径上的路由设备会积极参与三次握手过程中 MSS 协商机制，一旦发现自己出口的 MSS 比数据报文中的那个小，就会主动修改数据报文中的 MSS，这样整个路由链路端到端这条“高速路”的整体“限高”水平就准确清晰了。

通过【图八 TCP MSS 协商过程】，可以了解上述 TCP MSS 的协商过程。注意，这个“完美”方案需要运气好才行。因为中间路由设备五花八门，不能支持或者不愿支持 MSS 协商的情况时有发生。想让大伙都积极支持协商的美好愿望，就如同满怀对全世界各国政府官员实施财产公示的期许，结果是一样一样的。



【图八 TCP MSS 协商过程】

快乐运猪路遇限高紧急应对方案一有没有发挥空间呢？很好的问题，聪明的 TCP/IP 协议设计者当然不甘心，于是利用前面提到的 DF 标志位设计了一个叫做路径 MTU 发现的机制就用到了方案一的原理，如果 IP 数据报文的 3 位标志字段中的 DF 位置为 1，则 IP 层遇到需要 IP 分片时，就会选择直接丢弃报文，并返回一个相应的 ICMP 出错报文，看到了吧，此路不通，请带领群猪原路返回。这个方案运作成本颇高。不继续深入描述了，有兴趣可以研读《TCP/IP 详解 卷一：协议》。

至此，我们可以得出如下结论，TCP/IP 数据报文大小超过物理网络层的限制时，会引发 IP 分片，从而增加时空开销。

因此，设定合理的 MSS 至关重要，对于以太网 MSS 值建议是 1400 字节。什

么，你的数学是体育老师教的吗？前面说以太网最大的传输数据大小是 1500 字节，IP 数据报文包头是 20 字节，TCP 报文包头是 20 字节，算出来 MSS 怎么也得是 1460 字节呀。如果回答是因为很多路由设备比如 CISCO 路由器把 MSS 设定为 1400 字节，大伙肯定不干，回忆一下 IP 和 TCP 的数据报包头都各有 40 字节的可选项，MTU 中还需要为这些可选项留出空间，也就压缩了 MSS 的空间。要是再追问为啥这个值不是 1380 字节，那就有点过分了。

那么问题来了，控制“限高”哪种方案才最强。我们尝试探讨一下。
首先，可以在我们自己 IDC 内将各种路由交换设备的 MSS 设定小于或等于 1400 字节，并积极参与 TCP 三次握手时的 MSS 协商过程，期望达到自动控制服务器收发数据报文大小不超过路径最小 MTU 从而避免 IP 分片。这个方案的问题是如果路由路径上其它设备不积极参与协商活动，而它的 MTU（或 MSS 设置值）又比较 low，那就白干了。这就好比国家制定了一个高速沿途隧道限高公示通告标准，但是某些地方政府就是不告诉你，没辙。

其次，可以在业务服务中控制应用数据请求/响应的大小在 1400 字节以下（注：也无法根本避免前述方案中间路由 MTU/MSS low 的问题），在应用层数据写入时就避免往返数据包大小超过协商确定的 MSS。但是，归根到底，在出发前就把数据拆分为多个数据报文，同 IP 分片机制本质是相同的，交互响应开销增加是必然的。考虑到人在江湖，安全第一，本方案从源头上控制，显得更实际一些。

当然，最靠谱的还是做简法，控制传输数据的欲望，用曼妙的身姿腾挪有致，相关的内容放到轻往复章节探讨。
对应到前面的快乐运猪案例，就是要么在生猪装车之前咱们按照这条路上的最低限高来装车（问题是怎么能知道整个路上的最低限高是多少），要么按照国家标准规定允许的最小限高来装车，到这里，肥猪们终于可以愉快的上路了，风和日丽，通行无阻，嗯，真的吗？

3.1.1.2 放大 TCP 拥塞窗口

把 TCP 拥塞窗口（cwnd）初始值设为 10，这也是目前 Linux Kernel 中 TCP/IP 协议栈的缺省值。放大 TCP 拥塞窗口是一项有理有据的重要优化措施，对移动网络尤其重要，我们同样从一些基本理论开始逐步深入理解它。

TCP 是个传输控制协议，体现控制的两个关键机制分别是基于滑动窗口的端到端之间的流量控制和基于 RTT/RTO 测算的端到网络之间的拥塞控制。
流量控制目标是为了避免数据发送太快对端应用层处理不过来造成 SOCKET 缓存溢出，就像一次发了 N 车肥猪，买家那边来不及处理，然后临时囤货的猪圈又已客满，只好拒收/抛弃，相关概念和细节我们不展开了，有兴趣可以研读《TCP/IP 详解 卷一：协议》。

拥塞控制目标是在拥塞发生时能及时发现并通过减少数据报文进入网络的速率和数量，达到防止网络拥塞的目的，这种机制可以确保网络大部分时间是可用的。拥塞控制的前提在于能发现有网络拥塞的迹象，TCP/IP 协议栈的算法是通过分组丢失来判断网络上某处可能有拥塞情况发生，评判的具体指标为分组发送超时和收到对端对某个分组的重复 ACK。在有线网络时代，丢包发生确实能比较确定的表明网络中某个交换设备故障或因为网络端口流量过大，路由设备转发处理不及时造成本地缓存溢出而丢弃数据报文，但在移动网络中，丢包的情况就变得非常复杂，其它因素影响和干扰造成丢包的概率远远大于中间路由交换设备的故障或过载。比如短时间的信号干扰、进入一个信号屏蔽的区域、从空闲基站切换到繁忙基站或者移动网络类型切换等等。网络中增加了这么多不确定的影响因素，这在 TCP 拥塞控制算法最初设计时，是无法预见的，同时，我们也确信未来会有更完善的解决方案。这是题外话，如有兴趣可以找些资料深入研究。

拥塞控制是 TCP/IP 协议栈最经典的和最复杂的设计之一，互联网自我牺牲的利他精神表露无遗，设计者认为，在拥塞发生时，我们应该减少数据报文进入网络的速率和数量，主动让出道路，令网络能尽快调整恢复至正常水平。
拥塞控制机制包括四个部分：
a. 慢启动；
b. 拥塞避免；
c. 拥塞发生时的快速重传；
d. 快速恢复；

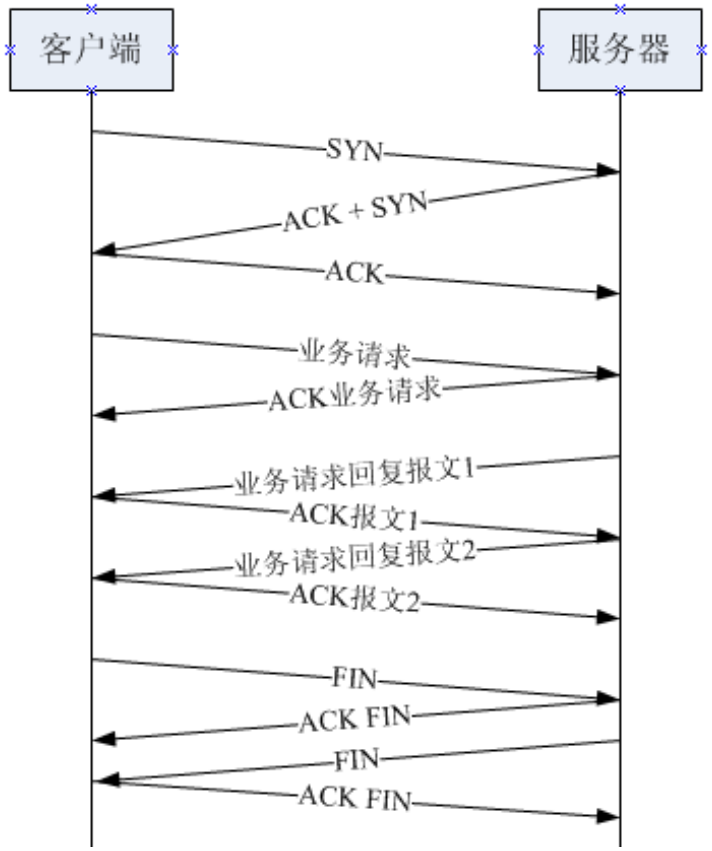
话题太大，我们聚焦到与本主题相关的【慢启动】上。
慢启动这项措施的缘起是，当新链接上的数据报文进入一个拥塞状况不可预知的网络时，贸然过快的数据发送可能会加重网络负担，就像养猪场每天都会向很多买家发车送肥猪，但是出发前并不了解各条高速路上的拥堵情况，如果按照订单

一口气全部发出去，会遇到两种情况，一是高速很顺畅，很快到达（此时流量控制可能要干预了）；二是高速本身就有些拥堵，大批卡车上路加剧了拥堵，并且肥猪们堵在路上，缺衣少食饿瘦了买家不干，风餐露宿冻死了卖家吃亏，重新发货还耽误时间，并且，用于重新发货的货车加入高速则进一步加重了拥堵的情况。作为一个充满社（wei）会（li）良（shi）知（tu）精神的养猪场，我们肯定不愿意贸然增加高（zi）速（ji）的负担。

下面进入简单的理论知识介绍部分，如觉枯燥，敬请谅解。

TCP 是一个可靠传输协议，基础是发送-应答（ACK）式确认机制，就好比肥猪运到目的地买家签收以后，要给卡车司机一个回执带回去交差，猪场老板一看回执，大喜过望，马上继续装车发运，如此往复。如【图九 TCP 链接建立、传输和关闭示意】，可以了解这种发送-应答式工作的基本流程，如果再结合流量控制、拥塞控制和超时重传等机制，会有很多变种 case，整个协议栈因而显得比较复杂。

但，万变不离其宗，老子说“是以圣人抱一为天下式”，真经典。



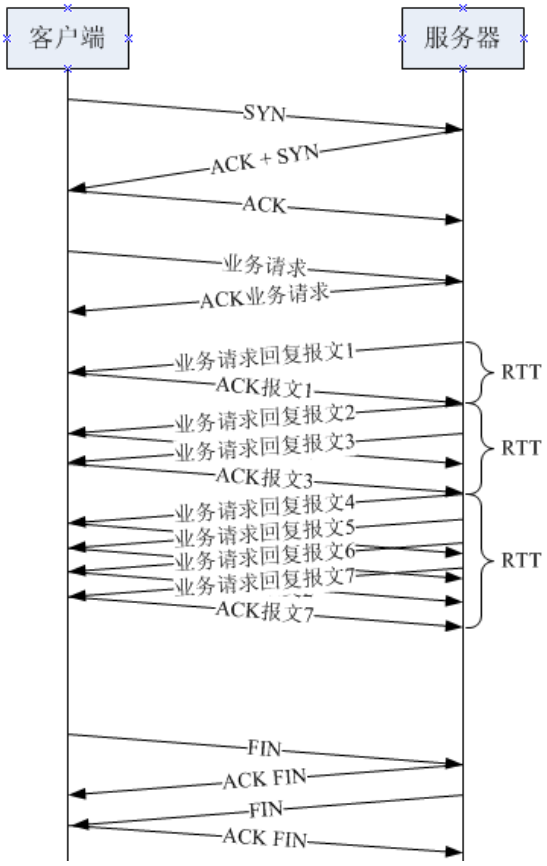
【图九 TCP 链接建立、传输和关闭示意】

慢启动顾名思义，就是把（网络链路数据报文传输）启动的速度放慢一些。方法其实也挺简单，TCP 发送方维护了两个参数用于控制这个过程，它们分别是拥塞窗口（cwnd，Congestion Window)和慢启动门限（ssthresh，Slow Start Threashold），具体算法如下：

1) TCP 链接建立好以后，cwnd 初始化 1，单位是链接建立过程中协商好的对端 MSS，1 代表一次可以发送 1 * MSS 个字节。ssthresh 初始化为 65535，单位是字节；
2) 每当收到一个 ACK，cwnd ++，cwnd 呈线性上升，发送方此时输出数据量不能超过 cwnd 和接收方通告的 TCP 窗口（这个概念我们在后面的章节中会介绍）大小；
3) 每当经过一个 RTT（Round Trip Time，网络往返时间），cwnd = cwnd * 2，cwnd 呈指数让升，同样发送方此时输出数据量不能超过 cwnd 和接收方通告的 TCP 窗口大小；
4) ssthresh（slow start threshold）是一个上限，当 cwnd >= ssthresh 时，就进入“拥塞避免”算法；

广告时间，插播简单介绍一下 RTT，它是 Round Trip Time（网络往返时间）的简写，简单的理解就是一个数据报文从发送出去到接收到对端 ACK 确认的时间（这样描述其实不够严谨，因为我们没有展开数据报文发送和对端 ACK 确认的各种复杂 case）。RTT 是 TCP 超时重传机制的基础，也是拥塞控制的关键参数，准确的估算出 RTT 具有伟大的现实意义，同时也是一项相当艰巨复杂的任务。计算机科学先辈们在持续完善 RTT 的计算方法，从最初 RFC793 中描述的经典算法，到 Karn / Partridge 算法，最后发展到今天在使用的 Jacobson / Karels 算法，如有兴趣可自行以深入研究。

通过【图十 慢启动过程示意】，可以更直观的理解慢启动的过程，经过两个 RTT，cwnd 已经由初始值 1 演化为 4：即在接收方通告窗口大小允许的情况下，可以连续发送 4 个数据报文，然后继续指数增长，这么看来，慢启动一点都不慢。



【图十 慢启动过程示意】

注：示意图中三个 RTT 大括弧逐渐变大不是因为 RTT 数值变大，而是要 示意包含的数据报文变多；

猪场老板来解读一下这个算法，我们对一个买家同时维护两个账单数字，一是起运数量设为 n，单位是卡车，二是最大同时发货数量设为 m，以肥猪头数为单位，描述如下：

1) 同买家订单协商确定后，n 初始化 1，把符合通往买家的高速路上限高要求的一辆卡车最大装载肥猪头数设为 h，1 代表一次可以发送 1 * h 头肥猪。m 初始化为 65535，单位是头；

2) 每当收到一个买家回执，n ++，n 呈线性上升，猪场老板此时发货数量不能超过 n 和买家通告的临时囤货的猪圈大小；

3) 每当经过一个送货往返，n = n * 2，n 呈指数让升，同样猪场老板此时发货数量不能超过 n 和买家通告的临时囤货的猪圈大小；

4) m 是一个上限，当 n >= m 时，为了避免可能带来的高速拥堵，就要进入“拥塞避免”算法；

Linux Kernel 从 3.0 开始采用了把 cwnd 初始化为 10 个 MSS，而在此之前，Linux Kernel 采用了 RFC3390 的规定，cwnd 是根据 MSS 的值来动态变化的。简单来说，cwnd 初始化为 10，就是为了允许在慢启动通过往复 RTT “慢慢”提升拥塞窗口前，可以在第一个网络传输回合中就发送或接收 14.2KB（1460 *10 vs 5.7KB 1460* 4）的数据。这对于 HTTP 和 SSL 来讲是非常重要的，因为它给了更多的空间在网络交互初始阶段的数据报文中填充应用协议数据。

对于移动 APP，大部分网络交互都是 HTTP 并发短链接小数据量传输的形式，如果服务器端有 10KB +的数据返回，采用过去的慢启动机制时，效率会低一些，大概需要 2~3 个 RTT 才能完成数据传输，反应到用户体验层面就是慢，而把拥塞窗口 cwnd 初始值提升到 10 后，在大多数情况下都能在 1 个 RTT 的周期内完成应用数据的传输，这在移动网络这样的高时延、不稳定、易丢包的场景下，显得尤其意义重大。
一次就发 10 卡车肥猪，让慢启动歇一会，别问为什么，有钱，任性。

3.1.1.3 调大 SOCKET 读写缓冲区

把 SOCKET 的读缓冲区（亦可称为发送缓冲区）和写缓冲区（亦可称为接收缓冲区）大小设置为 64KB。在 Linux 平台上，可以通过 setsockopt 函数设置 SO_RCVBUF 和 SO_SNDBUF 选项来分别调整 SOCKET 读缓冲区和写缓冲区的大小。

这两个缓冲区跟我们的 TCP/IP 协议栈到底有怎么样的关联呢。我们回忆一下【图六 TCP 数据报格式及首部中的各字段】，里面有个 16 位窗口大小，还有我们前面提到的流量控制机制和滑动窗口的概念，大幕徐徐拉开，主角纷纷粉墨登场。在正式详细介绍之前，按照传统，我们还是先站在猪场老板的角度看一下，读缓冲区就好比买家用来囤货的临时猪圈，如果货到了买家使用部门来不及处理，就先在这里临时囤着，写缓冲区就好比养猪场根据订单装好车准备发货，如果买家说我现在可以收货便可速度发出，有点明白了吧。下面详细展开探讨：

a. 【TCP 窗口】

整个 TCP/IP 协议体系是经典的分层设计，TCP 层与应用层之间衔接的部分，就是操作系统内核为每个 TCP 链路维护的两个缓冲区，一个是读缓冲一个是写缓冲。从数据结构角度讲，这两个缓冲区是环形缓冲区。

读缓冲肩负的使命是把接收到并已 ACK（确认）过的 TCP 报文中的数据缓存下来，由应用层通过系统接口读取消费。就好比买家内部会分原料采购部门和产品加工部门，采购部门收到肥猪后先送到临时猪圈好吃好喝供着，加工部门需要的时候就会拎着屠刀过来提猪。

写缓冲肩负的重任是缓存应用层通过系统接口写入的要发送的数据，然后由 TCP/IP 协议栈根据 cwnd、ssthresh、MSS 和对端通告的 TCP 窗口等参数，择机把数据分报文段发往对端读缓冲。想要在拥塞控制等相关参数都允许的条件下连续发送数据报文，尚需对端通告的 TCP 窗口大小能够容纳它们。就好比猪场老板根据买家订单发货，先调配若干辆卡车，根据高速的限高要求装上肥猪，然后再考虑高速的顺畅情况来分批发货，货可以陆续上路，但还有一个重要前提是发货前买家通告的临时猪圈空间是足够容纳这些肥猪的。

TCP 窗口是用于在接收端和发送端之间动态反映接收端读缓冲大小的变化，它的初始值就是读缓冲区设定的值，单位是字节，这个数字在 TCP 包头的 16 位窗口大小字段中传递，最大 65535 字节，如果嫌不够大，在 TCP 选项中还有一个窗口扩大的选项可供选择。

为什么叫窗口，一窗一风景，英文世界很现实，境界也就到 Window 级了，这与中华文明一沙一世界，一花一天堂的差距甚大。再直观一些的类比就是你拿着一个放大镜，在 1:10000 的军用地图上顺着一条路苦苦寻找东莞某镇，放大镜的范围就是我们说的窗口。

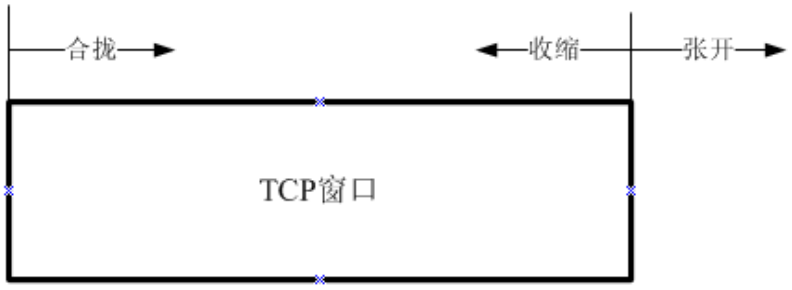
概括而言，TCP 窗口的作用是量化接收端的处理能力，调控发送端的传输节奏，通过窗口的伸缩，可以自如的调节发送端的数据发送速率，从而达到对接收端流量控制的目的。

师傅三藏曾经对悟空说：你想要啊？你想要说清楚不就行了吗？你想要的话我会给你的，你想要我当然不会不给你啦！不可能你说要我不给你，你说不要我却偏要给你，大家讲道理嘛！现在我数三下，你要说清楚你要不要……，嗯，说清楚最重要。

b. 【滑动窗口】

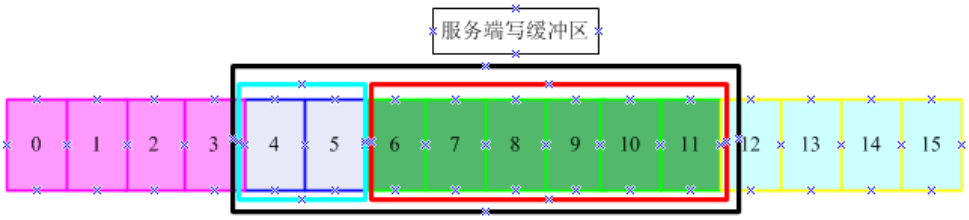
客户端和服务器在 TCP 链接建立的三次握手过程中，会根据各自接收缓冲区大小通告对方 TCP 窗口大小，接收方根据自己接收缓冲区大小初始自己的“接收窗口”，发送方根据对端通告的 TCP 窗口值初始化一个对应的“发送窗口”，接收窗口在此端的接收缓冲区上滑动，发送窗口在彼端的发送缓冲区上滑动。因为客户端和服务器是全双工，同时可收可发，故我们有两对这样的窗口在同时工作。既然是滑动窗口，就意味着可以滑动、伸缩，【图十一 TCP 窗口边沿移动】展示了这些情况，注意 TCP/IP 协议栈规定 TCP 窗口左边沿只能向右滑动，且 TCP 的 ACK 确认模式也在机制上禁止了 TCP 窗口左边沿向左移动。与窗口滑动相关术语有三个：

1) TCP 窗口左边沿向右边沿靠近称为窗口合拢，发生在数据被发送和确认 时。如果左右边沿重合时，则形成一个零窗口，此时发送方不能再发送任何数据；
2) TCP 窗口右边沿向右移动称为窗口张开，也有点类似窗口向右侧横向滑动。这种现象发生在接收方应用层已经读取了已确认过的数据并释放了 TCP 接收缓冲区时；
3) TCP 窗口右边沿向左移动称为窗口收缩，RFC 强烈建议避免使用这种方式；



【图十一 TCP 窗口边沿移动】

我们再来看看滑动窗口与 SOCKET 缓冲区如何结合使用。假设一个客户端设置了 16 个单位的读缓冲区，编号是 0 ~ 15，服务器也相应的设置了 16 个单位的写缓冲区，编号是 0 ~ 15。在 TCP 链接建立的时候，客户端会把自己的读缓冲大小 16 通告给服务器，此时在客户端和服务器就维护了一对收发窗口。在【图十二 服务器 TCP 发送窗口示意】展示了服务端发送缓冲区和其上的滑动窗口，其中大的黑色边框就是著名的滑动窗口。



【图十二 服务器 TCP 发送窗口示意】

发送缓冲和发送窗口一共区隔出四个部分：

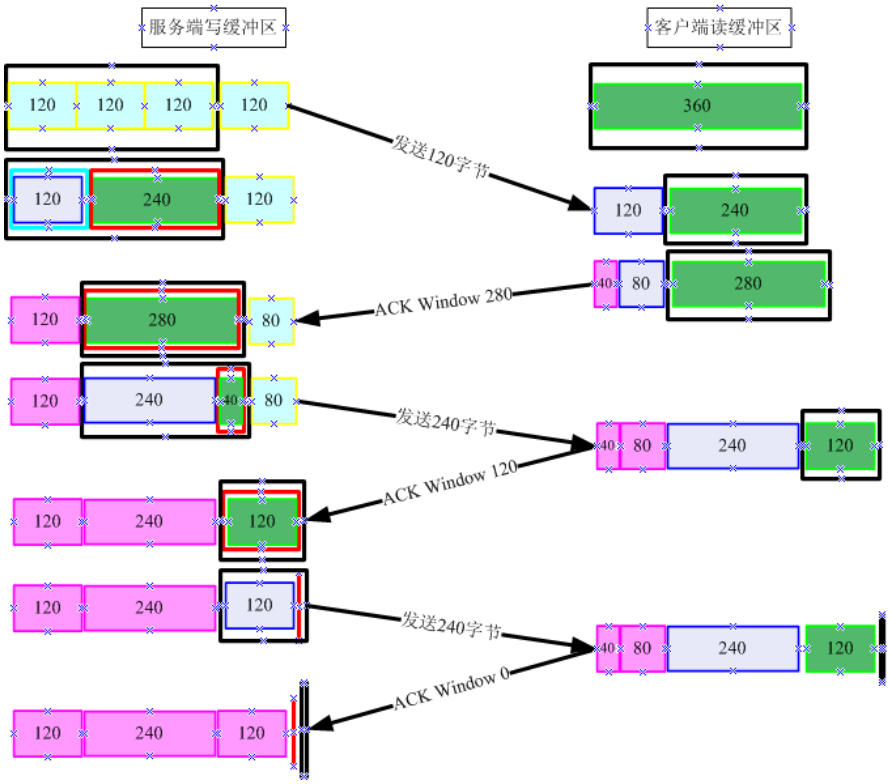
1）已发送并收到 ACK 确认的数据（即已成功到达客户端），单元格边框以粉色标识；

2）已发送还未收到 ACK 确认的数据（即发送但尚未能确认已被客户端成功收到），单元格边框以蓝色标识；

3）处于发送窗口中还未发出的数据（即对端接收窗口通告还可容纳的部分），单元格边框以绿色标识；

4）处于发送窗口以外还未发出的数据（即对端接收窗口通告无法容纳的部分），单元格边框以黄色标识；

为了更好的理解滑动窗口的变化过程，可以观察【图十三 TCP 滑动窗口变迁示例】，它向我们展示了一个服务器向客户端发送数据时读写窗口的变化过程：



【图十三 TCP 滑动窗口变迁示例】

1) 客户端通告了一个 360 字节的 TCP 窗口并在自己的读缓冲区初始化该窗口，服务器在它的写缓冲区初始化了这个窗口；
2) 服务器发送 120 字节到客户端，服务器发送窗口此时包括了两部分，120 字节为等待 ACK 确认的数据、240 字节为等待发送的数据，窗口大小为 360 字节不变；
3) 客户端收到 120 字节数据，放入接收缓冲区，此时应用层马上读取了头 40 字节，接收窗口因此调整为 280（360 - 120 + 40）字节，接收窗口先合拢，然后张开。客户端回复 ACK 确认收到 120 字节数据，并且通告接收窗口调整为 280 字节；
4) 服务器收到客户端的 ACK 确认，发送窗口也先发生合拢，随后根据客户端通告的新接收窗口大小，重新调整发送窗口，此时发送窗口又张开至 280 字节；
5) 服务器发送 240 字节到客户端，服务器发送窗口此时包括了两部分，240 字节为等待 ACK 确认和 40 字节等待发送的数据，窗口大小为 280 字节不变；
6) 客户端收到 240 字节数据，放入接收缓冲区，此时应用层又读取了头 80 字节，接收窗口因此调整为 120（280 - 240 + 80），接收窗口先合拢，然后张开。客户端回复 ACK 确认收到 240 字节数据，并且通告接收窗口调整为 120 字节；
7) 服务器收到客户端的 ACK 确认，发送窗口也先发生合拢，随后根据客户端通告的新接收窗口大小，重新调整发送窗口，此时发送窗口又张开至 120 字节；
8) 服务器发送 120 字节到客户端，服务器发送窗口此时仅包括一部分，即 120 字节等待 ACK 确认的数据；
9) 客户端收到 120 字节数据，放入接收缓冲区，接收窗口因此调整为 0（120 - 120），接收窗口合拢为 0。客户端回复 ACK 确认收到 120 字节数据，并且通告接收窗口调整为 0 字节；
10) 服务器收到客户端的 ACK 确认，发送窗口也发生合拢，随后根据客户端通告的新接收窗口大小，重新调整发送窗口，此时因为接收窗口为 0，发送窗口保持合拢状态；

提升 TCP 吞吐量，最佳状态是在流量控制机制的调控下，使得发送端总是能发送足够的数据报文填满发送端和接收端之间的逻辑管道和缓冲区。其中逻辑管道的容量有专门的学名叫 BDP（Bandwidth Delay Product，带宽时延乘积，BDP = 链路带宽 * RTT），在一个高带宽低时延的网络中，TCP 包头中的 16 位窗口大小可能就不够用了，需要用到 TCP 窗口缩放选项，在 RFC1323 中定义，有兴趣可以研究一下。

猪场老板解读：滑动窗口是从养猪场到买家临时猪圈的出入闸门，猪场养殖场这道出闸门叫发送窗口，买家临时猪圈那道入闸门叫接收窗口，为了不让买家的临时猪圈爆满溢出无法签收新来的肥猪们，进而导致猪场白送一趟货，猪场老板必须要等买家通告自己空闲槽位数量后才可进行生猪发货操作，这个槽位数量就是窗口大小，槽位减少或增加，受到猪场发货速率和买家屠宰部门提货速率的共同影响，表现出类似窗口合拢或张开的滑动状态。我们期待的最佳状态就是高速路上跑满欢快的车队，临时猪圈住满幸福的肥猪。
三藏对小牛精说：所以说做妖就像做人，要有仁慈的心，有了仁慈的心，就不再是妖，是人妖。哎，他明白了，你明白了没有？

3.1.1.4 调大 RTO（Retransmission TimeOut）初始值

将 RTO（Retransmission TimeOut）初始值设为 3s。
TCP 为每一个报文段都设定了一个定时器，称为重传定时器(RTO)，当 RTO 超时且该报文段还没有收到接收端的 ACK 确认，此时 TCP 就会对该报文段进行重传。当 TCP 链路发生超时时，意味着很可能某个报文段在网络路由路径的某处丢失了，也因此判断此时网络出现拥塞的可能性变得很大，TCP 会积极反应，马上启动拥塞控制机制。

RTO 初始值设为 3s，这也是目前 Linux Kernel 版本中 TCP/IP 协议栈的缺省值，在链路传输过程中，TCP 协议栈会根据 RTT 动态重新计算 RTO，以适应当前网络的状况。有很多的网络调优方案建议把这个值尽量调小，但是，我们开篇介绍移动网络的特点之一是高时延，这也意味着在一个 RTT 比较大的网络上传输数据时，如果 RTO 初始值过小，很可能发生不必要的重传，并且还会因为这个事件引起 TCP 协议栈的过激反应，大炮一响，拥塞控制闪亮登场。
猪场老板的态度是什么样的呢：曾经有一份按时发货的合同摆在我的面前，我没有去注意，等到重新发了货才追悔莫及，尘世间最痛苦的事莫过于此，如果上天能给我一个再来一次的机会，我希望对甲方说耐心点，如果非要给这个耐心加一个期限的话，我希望是一万年。

3.1.1.5 禁用 TCP 快速回收

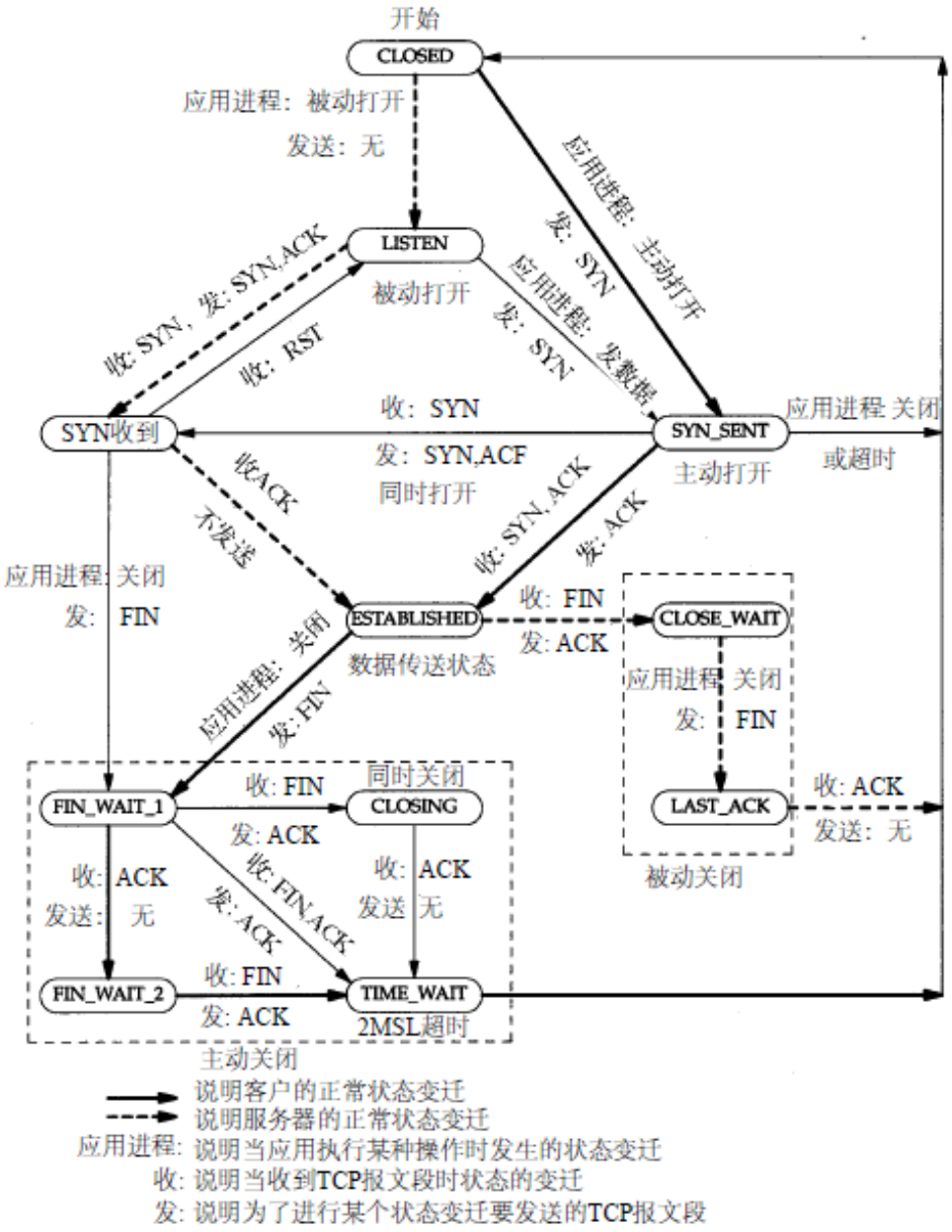
TCP 快速回收是一种链接资源快速回收和重用的机制，当 TCP 链接进入到 TIME_WAIT 状态时，通常需要等待 2MSL 的时长，但是一旦启用 TCP 快速回收，则只需等待一个重传时间（RTO）后就能够快速的释放这个链接，以被重新

使用。Linux Kernel 的 TCP/IP 协议栈提供了一组控制参数用于配置 TCP 端口的快速回收重用，当把它们的值设置为 1 时表示启用该选项：
1) net.ipv4.tcp_tw_reuse = 1
2) net.ipv4.tcp_tw_recycle = 1
3) net.ipv4.tcp_timestamps = 1（tcp_tw_recycle 启用时必须同时启用本项，反之则不然，timestamps 用于 RTT 计算，在 TCP 报文头部的可选项中传输，包括两个参数，分别为发送方发送 TCP 报文时的时间戳和接收方收到 TCP 报文响应时的时间戳。Linux 系统和移动设备上的 Android、iOS 都缺省开启了此选项，建议不要随意关闭）

以上参数中 tw 是 TIME_WAIT 的缩写，TIME_WAIT 与 TCP 层的链接关闭状态机相关。下面我们看看 TIME_WAIT 是谁，从哪里来，往哪里去。
前面我们在介绍基础理论知识的时候，【图九 TCP 链接建立、传输和关闭示意】中最后四个数据报文就是 TCP 链接关闭的过程，俗称四次挥手，分手总是难以割舍的，所以链接建立只需三次握手，分手得要四次回首。

TCP 设计目标是可靠传输，哪怕在分手时也得确保成功。为此，在 TCP 链接关闭阶段设计了繁杂的状态机，在【图十四 TCP 状态变迁图】的左下角虚线框中的四个状态 FIN_WAIT1、FIN_WAIT2、CLOSING、TIME_WAIT，代表着主动关闭 TCP 链接这一方的可能状态，前三个状态最终都会进入到等待响应最后一个 FIN 的 ACK 的这个阶段，即 TIME_WAIT 状态，并且在此停留 2MSL（2 倍 Maximum Segment Lifetime，2 倍报文段最大生存时间，RFC793 规定 MSL 为 2 分钟，Linux Kernel 中 TCP/IP 协议栈采用的是 30 秒，这个值的选择是有讲究的，它是一个物理上的约束，表示一个 IP 数据报文在地球上最长的存活时间，意思就是即便收不到这个 ACK，也会给时间让它最终在地球的某个角落里消失）时长。这样处理的原因是在四次挥手过程中，主动关闭方需要确保自己最后发送响应对端 FIN 的 ACK 能被对端收到，如果对端出现超时重传了 FIN，则意味着自己上次发的 ACK 丢失了，那么自己还有机会再次发送 ACK 确认，乘以 2 就是为了给重传的 ACK 充裕的到达时间。

真是太缠绵了，感天动地。在创造 TCP/IP 的年代，窄带宽、高时延、不稳定的网络状态，这样的设计相当必要，要分手也得大家都确认才行，爱情片里太多这样的误会了，不学习网络知识生活中是要吃大亏的。



【图十四 TCP 状态变迁图】

回归正题，前面的基础知识告诉我们，只有 TCP 链接的主动关闭方会进入 TIME_WAIT 状态，这会给链接主动关闭方所在的 TCP/IP 协议栈带来什么样的影响呢。归纳一下主要有两个方面：

1) TCP/IP 协议栈随机端口资源耗尽

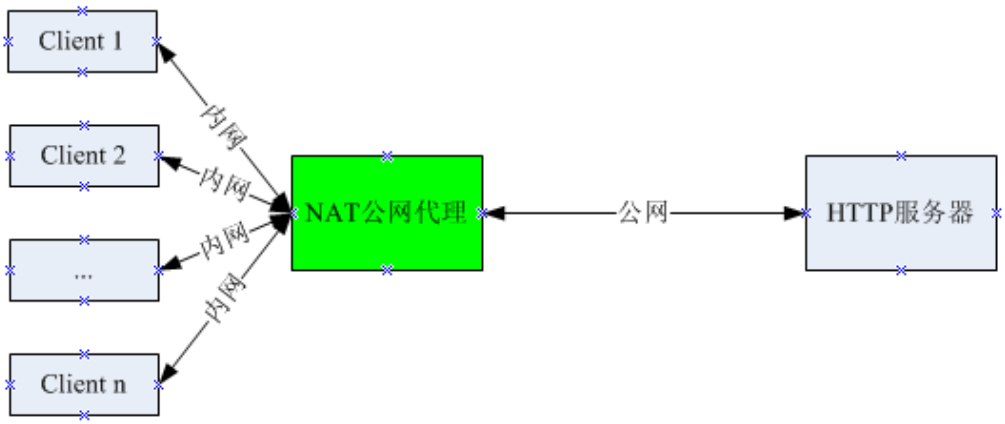
铺垫一个基础知识：TCP 对每个链接用一个四元组（TUPLE）来唯一标识，分别是源 IP、目标 IP、源端口、目标端口。通常在使用一个特定的目标服务时，目标 IP（即服务器 IP）和目标端口（即服务器知名/私有端口）是固定的，源 IP 通常

也是固定的，因此链接主动发起方 TUPLE 的最大数量就由源端口的最大数量决定，TCP/IP v4 规定端口号是无符号短整型，那么这个最大值就是 65535。
假设一个服务器即作为 TCP 链接的主动打开方（通常是作为客户端角色，它使用本地随机分配的临时端口）又是 TCP 链接的主动关闭方，则大量主动关闭的链接会进入到 TIME_WAIT 状态，如果大伙在这个状态都折腾 60 秒（Linux MSL 缺省为 30 秒，2MSL 为 60 秒），这台机器相关的 TUPLE 资源会被快速占用、堆积并很快因为（源）端口的 65535 限制而耗尽，以后该 TCP/IP 协议栈上运行的其程序作为 TCP 链接的主动打开方再想链接同一个目标服务器时，就只能等待 2MSL 释放，从应用角度来看就是链接建立失败，用户要承受精神和肉体双重折磨，无法接受。

2) TCP/IP 协议栈 TUPLE 相关数据结构大量消耗内存
假设一个服务器作为 TCP 链接的被动打开方（通常是作为服务器角色）和主动关闭方，则大量主动关闭的链接会进入到 TIME_WAIT 状态，如果大伙在这个状态都折腾 60 秒，本地机器 TCP/IP 协议栈维护的 TUPLE 数据项会快速堆积并占用大量内核内存资源，最关键的是因为此时 TCP 四元组碰撞概率极低（因为源 IP、源端口大多都是不同的），导致 TUPLE 的积压几乎不受限制而野蛮生长，这对于一个高负载又要求高性能的服务器而言，感情上是相当痛苦的，肉体上勉强能接受。

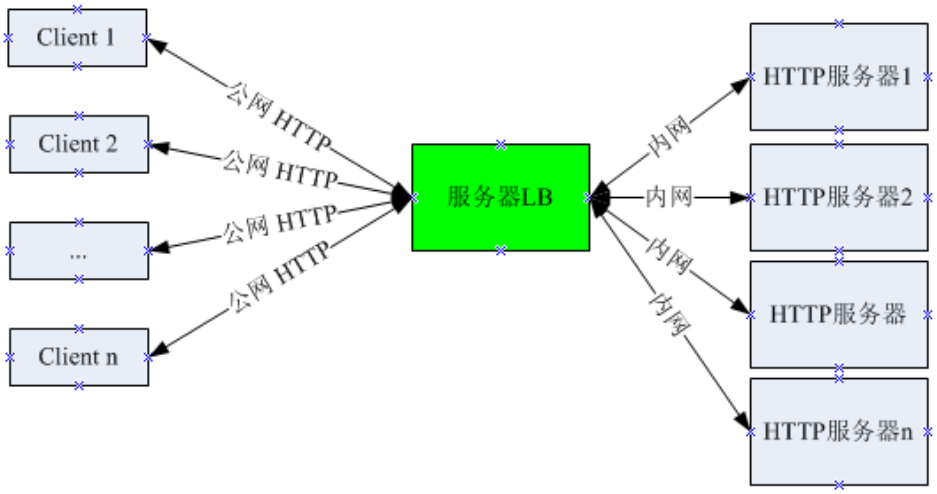
基于以上分析，为了提高服务器网络效能，一些服务器选择配置启用 TCP 快速回收（真的需要配置吗，配置真的有效果吗，后面逐步会谈到）来优化性能。然而，新的问题出现了，三藏说：看，现在是妹妹要救姐姐，等一会那个姐姐一定会救妹妹的……恩恩怨怨何时了啊。

【问题 1】如果客户端通过同一个 NAT 链接应用服务器时，客户端 TCP 链接可能被 RESET 拒绝或者无响应、响应缓慢。我们来具体分析一下成因，NAT 作为代理层面向服务器时，客户端侧的源 IP 会被收敛成 NAT 的地址，通常有三种情况：
1）NAT 为公网代理，比如公司内大伙用手机通过 WIFI 上网就属于这种模式，逻辑结构类似【图十五 客户端通过 NAT 上网示意】。另有一点背景交待：我们上网冲浪时发起的链接绝大多数都是短链接；



【图十五 客户端通过 NAT 上网示意】

2) NAT 为后端服务器集群做四层或七层 Load Balance（以下简称 LB），比如 HAProxy 或 LVS 的四层 NAT 模式、Nginx 的七层 LB 模式，典型场景是客户端 HTTP 请求经过 LB 转发到后端的服务器集群。LB 与服务器集群之间大多也是采用短链接，逻辑结构类似【图十六 服务器通过 NAT 做 LB】；



【图十六 服务器通过 NAT 做 LB】

3) 上述第 1 和第 2 中情况的组合，具体可以参考【图十七 典型客户端连接服务器链路示意】，后面会有专门的讨论，此处不再赘述；

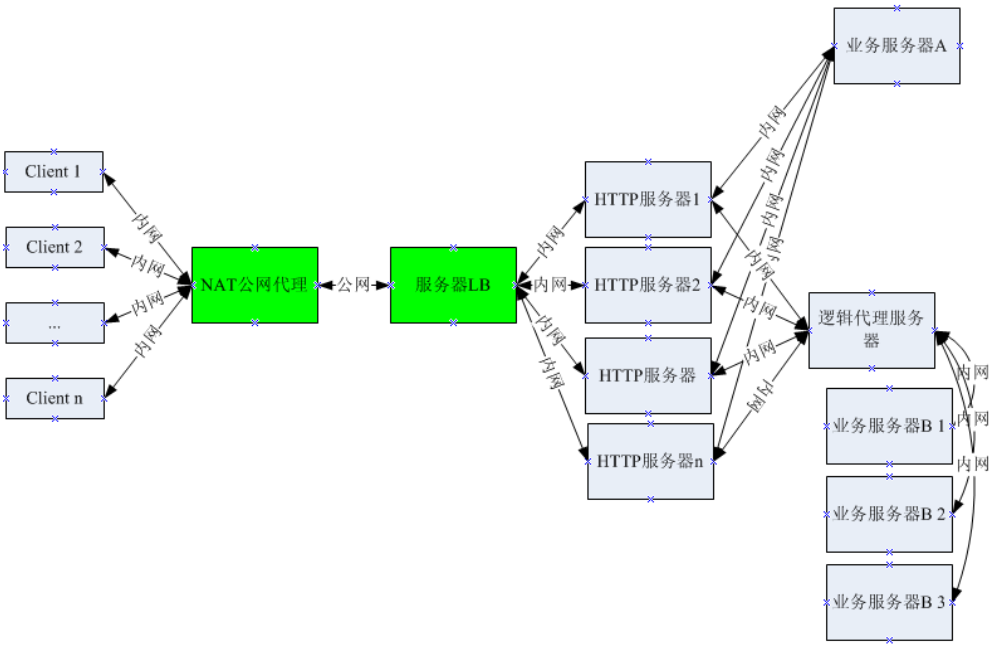
Linux Kernel 的 TCP/IP 协议栈在开启 TCP 链接 TIME_WAIT 状态快速回收时，只需等待一个重传时间（RTO 可能很短，甚至都来不及在 netstat -ant 中看到 TIME_WAIT 状态）后就释放而无需等待通常的 2MSL 超时。被释放的 TCP 链接的 TUPLE 信息同时也就就清除了。那么，问题来了，如果短时间内有新的 TCP 链接复用了这个 TUPLE，就有可能会因为收到之前已释放的链接上，因延迟而

刚刚到达的 FIN，从而导致新链接被意外关闭。实际上，还会有链路被串接的问题。

为了规避这些问题，TCP/IP 协议栈在快速回收释放 TUPLE 后，又利用 IP 层 PEER（TCP/IP 协议栈中维护的链接对端数据结构）信息中的对端 IP、PEER 最后一次 TCP 数据报文时间戳等信息（注：对端端口信息此时已经在 TCP 层被清除掉了），对 TCP 链接通过快速回收和重用 TUPLE 到新链接上做了一系列约束，在 RFC1323 中有相应的描述。简单讲就是在同时满足以下条件时，不能重用从 TIME_WAIT 状态快速回收的 TUPLE，此时的表现是不响应或对 SYN 请求响应 RESET：

1）来自同一台 PEER 机器的 TCP 链接数据报文中携带时间戳字段；
2）之前同一台 PEER 机器（仅仅指 IP，端口信息因链接被 TCP 快速释放而缺失） 的某个 TCP 报文曾在 60 秒之内到过本服务器；
3）新链接的时间戳小于 PEER 机器上次 TCP 到来时的时间戳；
条件已经相当苛刻，碰撞概率应该很低了。但由于只有 PEER 的 IP 而缺少 PEER 的端口信息作为判断 TCP 链接另一端唯一性的约束，不能重用的概率便放大了 65535 倍。假设 PEER 是一台单独的机器，问题不大，因为一台机器上的时间戳是单调增长的，一旦出现时光倒流，则可以确定是旧的数据报文延迟了，直接丢掉即可。但是，如果很多客户端通过同一台 NAT 设备接入进来，那么问题就严重了，因为工作在四层的 NAT 不会修改客户端发送的 TCP 报文内的时间戳，而客户端们各自的时间戳又无法保持一致，服务器只认时间戳最大的那个，其它通通丢掉或者对 SYN 请求直接响应 RESET，太冤了。

我们的业务服务中，典型模式是客户端使用 HTTP 短链接通过接入服务器使用业务服务，且这些接入服务器基本都是以 LB 方式在运行，接入服务器与业务服务器之间则大多为直接链接或通过代理调度，无论是有线互联网的 B/S 架构，还是移动互联网的 C/S 架构都是如此。客户端用户也大多数都是通过 NAT 上网的。参考【图十七 典型客户端连接服务器链路示意】可以有更直观的了解。



【图十七 典型客户端连接服务器链路示意】

基于前述知识，我们以【图十七 典型客户端连接服务器链路示意】为基础来观察，可以分三种情况讨论快速回收配置参数的合理使用：

1）链接主动打开方和主动关闭方均为客户端

a. 如服务器 LB 工作在七层且在公网提供服务，则它与 HTTP 服务器集群之间一般都是短链接，此时，服务器 LB 符合随机端口资源耗尽的模式。因为它的时间戳是单调递增的，故无需担心链接碰撞，符合 TCP 快速回收重用的条件，但由于服务器 LB 部署在公网对客户端提供服务，客户端有可能通过 NAT 代理访问外部网络，便无法保证时间戳单调递增，故建议关闭 TCP 快速回收选项；

b. 如服务器 LB 工作在四层模式，自身不受影响，故关闭 TCP 快速回收选项；

c. HTTP 服务器集群与层级靠后的业务服务器之间大多都是短链接，HTTP 服务器的情况与前述第 a 点类似，如果它在七层服务器 LB 之后部署，且与层级靠后的业务服务器之间没有 NAT，则可以考虑启用 TCP 快速回收选项，除此之外，都建议关闭 TCP 快速回收选项；

2）内网服务器（业务服务器、逻辑代理服务器等）之间有相互调用时，建议优先采用长链接方案。如果确实需要使用短链接方案时，则层级靠前的服务器往往即是链接的主动打开方，又是链接的主动关闭方，符合随机端口资源耗尽的模式。考虑到单台服务器能确保自己时间戳单调递增，开启 tcp_tw_recycle 也能符合 TCP 快速回收重用的条件，且不用担心碰撞，因此建议启用 TCP 快速回收选项。

这里需要注意两个特殊情况：
a. 如果层级靠前的服务器有一端直接在公网为客户端提供服务，而客户端有可能通过 NAT 代理访问外部网络，则不宜启用 TCP 快速回收选项；
b. 如果层级靠前的服务器与层级靠后的服务器之间有四层 NAT 隔离，也需要谨慎考虑。除非服务器间系统时钟同步精准，能确保层级靠前的服务器集群总体时间戳在毫秒级的精度上能单调递增，否则建议关 闭 TCP 快速回收选项；
3) 服务器集群被模拟客户端逻辑攻击，此时服务器会主动关闭链接，从而导致大量出现 TIME_WAIT 状态，服务器因此符合 TCP/IP 协议栈 TUPLE 相关数据结构内存大量消耗的模式但，考虑到客户端可能处在 NAT 之后，建议保持关闭 TCP 快速回收选项。我们应利用提前部署的安全机制在 TCP 三次握手期间及早拒绝链接来解决此类问题；

服务端系统架构千变万化，较难穷举，总结一下上述的讨论：
1) 服务器如果直接在公网服务于客户端时，因为客户端有可能通过 NAT 代理访问外部网络，故建议关闭 TCP 快速回收选项；
2) 服务器各层级在内网互联时，同时作为链接的主动发起方和链接的主动关闭方，建议开启 TCP 快速回收。上述建议例外场景是：如服务器层级之间有 4 层 NAT，则需要考察层级靠前的服务器集群时钟同步的精度水平是否能到毫秒级，通常建议关闭 TCP 快速回收选项；

【问题 2】CMWAP 转发的包时间戳有乱跳的情况，也会遇到类似问题 1 的现象。因为现在 WAP 的用户越来罕见，就不展开了；
⑥ HTTP 协议：打开 SOCKET 的 TCP_NODELAY 选项
TCP/IP 协议栈为了提升传输效率，避免大量小的数据报文在网络中流窜造成拥塞，设计了一套相互协同的机制，那就是 Nagle's Algorithm 和 TCP Delayed Acknoledgement。
Nagle 算法（Nagle's Algorithm）是以发明人 John Nagle 的名字来命名。John Nagle 在 1984 年首次用这个算法来尝试解决福特汽车公司的网络拥塞问题（RFC 896），该问题的具体描述是：如果我们的应用程序一次产生 1 个字节的数据（典型的如 telnet、XWindows 等应用），而这个 1 个字节数据又以网络数据包的形式发送到远端服务器，那么就很容易使网络中有太多微小分组而导致过载。
因为传输 1 个字节有效数据的微小分组却需花费 40 个字节的额外开销(即 IP 包

头 20 字节 + TCP 包头 20 字节），这种有效载荷利用率极其低下的情况被统称为愚蠢窗口症候群（Silly Window Syndrome），前面我们在谈 MSS 时也提到过，如果为一头猪开个大卡车跑一趟，也够愚钝的。对于轻负载广域网或者局域网来说，尚可接受，但是对于重负载的广域网而言，就极有可能引起网络拥塞导致瘫痪。

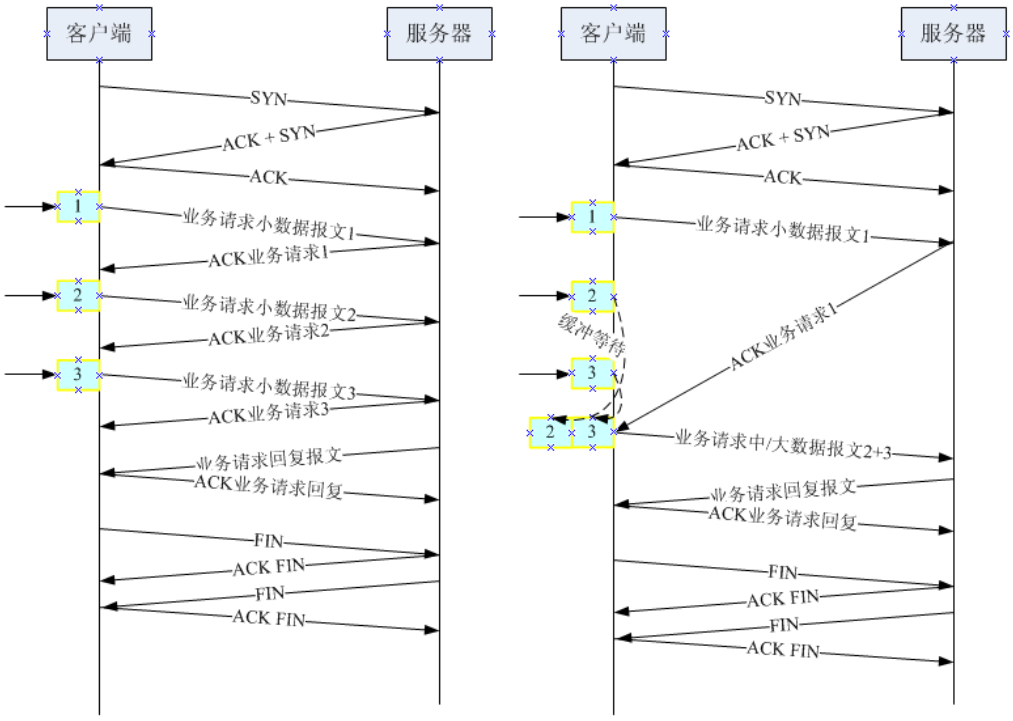
Nagle 算法要求一个 TCP 链接上最多只能有一个未被确认的小分组（数据长度小于 MSS 的数据包），在该分组的确认到达之前不能再发送其它小分组。此时如果应用层再有新的写入数据，TCP/IP 协议栈会搜集这些小分组并缓存下来，待以下时机发出：

1）收到接收端对前一个数据报文的 ACK 确认；

2）当前数据属于紧急数据；

3）搜集的数据达到或超过 MSS；

【图十八 Nagle 算法未开启和开启数据报文交互示意】对比了 Nagle 算法未开启（左侧图示）和开启（右侧图示）的数据报文交互过程。



【图十八 Nagle 算法未开启和开启数据报文交互示意】

TCP Delayed Acknoledgement 也是为了类似的目的被设计出来的，它的作用就是延迟 ACK 包的发送，使得 TCP/IP 协议栈有机会合并多个 ACK 或者使 ACK

可以随着响应数据一起返回，从而提高网络性能。TCP Delayed Acknoledgement 定义了一个超时机制，默认超时时间是 40ms，超过这个时间，则不再等待立即发送延迟的 ACK。

如果一个 TCP 连接的一端启用了 Nagle's Algorithm，而另一端启用了 TCP Delayed Acknoledgement，而发送的数据包又比较小，则可能会出现这样的情况：发送端在等待接收端对上一个数据报文的 ACK 才发送新的数据报文，而接收端则正好延迟了这个 ACK 的发送，那么正要被发送的新数据报文也就同样被延迟了。

上述情况出现的前提是 TCP 连接的发送端连续两次调用写 SOCKET 接口，然后立即调用读 SOCKET 接口时才会出现。那么为什么只有 Write-Write-Read 时才会出现问题，我们可以分析一下 Nagle's Algorithm 的伪代码：

```
if there is new data to send
  if the window size >= MSS and available data is >= MSS
    send complete MSS segment now
  else
    if there is unconfirmed data still in the pipe
      enqueue data in the buffer until an acknowledge is received
    else
      send data immediately
    end if
  end if
end if
```

代码显示，当待发送的数据比 MSS 小时，先判断此时是否还有未 ACK 确认的数据报文，如果有则把当前写的数据放入写缓冲区，等待上个数据报文的 ACK 到来。否则立即发送数据。对于 Write-Write-Read 的调用秩序，发送端第一个 Write 会被立刻发送，此时接收端 TCP Delayed Acknoledgement 机制期待更多的数据到来，于是延迟 ACK 的发送。发送端第二个 Write 会命中发送队列中还有未被 ACK 确认的数据的逻辑，所以数据被缓存起来。这个时候，发送端在等待接收端的 ACK，接收端则延迟了这个 ACK，形成互相等待的局面。后面等到接收端延迟 ACK 超时（比如 40ms），接收端就会立即发出这个 ACK，这才能触使发送端缓存的数据报文被立即发出。

现代 TCP/IP 协议栈默认几乎都启用了这两个功能。

我们在移动 APP 的设计实现中，请求大部分都很轻（数据大小不超过 MSS），为了避免上述分析的问题，建议开启 SOCKET 的 TCP_NODELAY 选项，同时，我们在编程时对写数据尤其要注意，一个有效指令做到一次完整写入（后面会讲协议合并，是多个指令一次完整写入的设计思想），这样服务器会马上有响应数据返回，顺便也就捎上 ACK 了。

3.1.2. 接入调度

3.1.2.1 就快接入

在客户端接入服务器调度策略的演化过程中，我们最早采用了“就近接入”的策略，在距离客户端更近的地方部署服务器或使用 CDN，期望通过减少 RTT 来提高网络交互响应性能。这个策略在国内的落地执行还需要加一个前缀：“分省分运营商”，这就给广大负责 IDC 建设的同学带来了巨大的精神和肉体折磨。
在持续运营的过程中，根据观察到的数据，发现并非物理距离最近的就是最快的。回忆一下前面谈到的吞吐量指标 BDP，它与链路带宽和 RTT 成正比关系，而 RTT 是受物理距离、网络拥塞程度、IDC 吞吐量、跨网时延等诸多因素综合影响的，单纯的就近显然不够精细了。

“就快接入”在“就近接入”策略的基础上改善提升，它利用客户端测速和报告机制，通过后台大数据分析，形成与客户端接入 IP 按就快原则匹配接入服务器的经验调度策略库，令客户端总能优先选择到最快的服务器接入点。
对于接入服务器，我们按照访问目标数据属性纬度的不同，可以分为至少两个集合，它们分别是：
1）业务逻辑服务器集合；
2）富媒体服务器集合，富媒体包括头像、图片和视频等尺寸比较大的数据；
这两类服务器集合通常由独立的接入调度 FSM 管理。
客户端在访问不同的数据类型时使用不同的服务器集合，这样的划分体现了轻重分离、信令和数据分离的架构理念。
每个服务器集合又可按接入调度的优先秩序划分为三个子列表：

1）【动态服务器列表】
服务器按策略（比如就快接入）并结合设备负载情和容量情况、网络容量情况综合计算下发的一系列服务器 IP 地址，某些产品还会在动态服务器列表靠后的部

分加上动态服务器域名（该域名与静态服务器域名列表内容不同，是一种动态扩展方式），对于下载类业务，动态服务器列表最后会包含动态回源服务器 IP 地址等。客户端应当持久化存储动态服务器列表，并在 APP 启动时加载到内存缓存中，其缓存索引的 KEY 通常是网络类型，对于 WIFI 网络，KEY 的内容中再加上一个 SSID，以便区分不同的 WIFI 热点。客户端在持久化和内存中基于不同的 KEY 缓存 3 ~ 5 组（建议值，可根据业务特点灵活选择和配置）动态服务器列表数据，并按照 LRU 方式做更新淘汰；

2) 【静态服务器域名列表】
预埋在客户端持久化存储中，在首次启动 APP 或动态服务器列表访问全部失败时使用；

3) 【静态服务器 IP 列表】
预埋在客户端持久化存储中，其主要价值在于，当使用客户端遇到动态服务器列表和静态服务器域名列表访问都出现异常时，有最低限度的可用性保障。静态服务器 IP 列表贵精不贵多，能分别服务国内和海外用户即可。对于下载类业务，静态服务器 IP 列表最后还有包含静态回源服务器 IP 地址；
每个服务器列表都包含一批列表项，一般为 2 ~ 3 个。每个服务器列表中的列表项按照优先顺序从前到后排列，故也需维护一个自己独立的调度机制，我们称之为服务器列表调度 FSM。

基于以上的分类基础，客户端和服务器接入调度机制的具体的做法通常为：
1) 客户端实现接入调度 FSM 模型和服务器列表调度 FSM 模型，这两个 FSM 是嵌套关系，可以理解为外循环和内循环的关系，就好比地球围着太阳公转的时候也没耽误自转；
2) 客户端存储预埋业务逻辑和富媒体两个服务器集合，每个服务器集合都包含静态服务器域名列表和静态服务器 IP 列表；
3) 服务器实现就快接入调度算法，依托异步计算持续更新的经验调度策略库，进行动态匹配计算；
4) 客户端和服务器共同实现一套动态服务器列表下发和更新机制；
5) 实践中有些服务器还要求客户端支持 302 跳转的能力，这个逻辑机制上可以有，策略上不提倡；

我们先考察接入调度 FSM，如【图十九 接入调度 FSM 示意】，它的状态变迁驱动力来自：

1）当前状态下相应的服务器列表无有效数据（数据项为空或全部试完一轮）；

2）服务器下发了新的动态服务器列表；

接入调度 FSM 状态变迁的原则是：

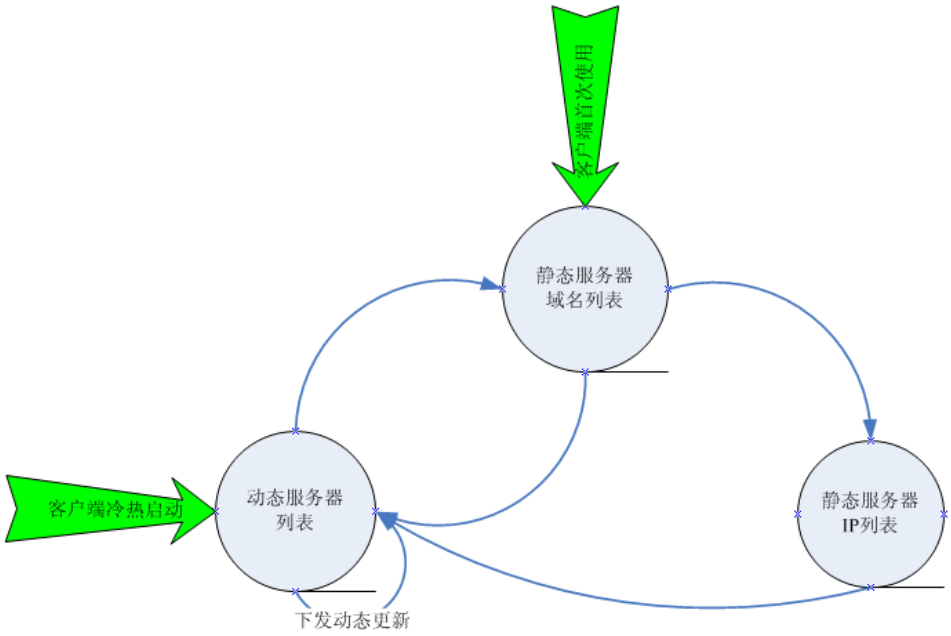
1）客户端首次使用时，接入调度 FSM 状态入口在静态服务器域名列表；

2）客户端在冷启动（除首次使用）、热启动时，接入调度 FSM 状态入口在动态服务器列表。动态服务器列表通常在冷启动时从本地持久化缓存加载，在内存缓存中会被服务器下发的数据更新，一旦更新，客户端应择机持久化到本地存储中；

3）接入调度 FSM 状态变迁时，以进入服务器下发的动态服务器列表状态为最高优先级，即三个服务器列表发生状态变迁时，都先向服务器动态列表跳转；

4）第 3 点之特例：当刚从动态服务器列表变迁到静态服务器域名列表且未收到服务器下发新的动态服务器列表时，静态服务器域名列表变迁的下一站是静态服务器 IP 列表。这里要特别谈一下前面那个时间限定词“刚”，这个前提设定的原因是移动网络易抖动，1 分钟前动态服务器服务器列表不可用不代表 5 分钟后依然不可用，因此，我们把这个“刚”设定为：一直在前台运行的 5 分钟以内的时间；

5）特别的，如果是因为服务器下发新的动态服务器列表导致状态变迁，那么接入调度 FSM 状态要置位还原，重新按第 2 条原则执行；



【图十九 接入调度 FSM 示意】

我们以动态服务器列表为例来考察服务器列表调度 FSM，先说明一下，同其他两个列表不同的是，动态服务器列表中的列表项数量完全由服务器下发时控制。如【图二十 动态服务器列表调度 FSM】所示，服务器列表调度 FSM 的状态变迁驱动力来自：

1) 链接建立失败或超时；

2) 链接建立成功但收发数据错误（包含网络类型切换、无网络等）或超时；

3) 服务器下发新的动态服务器列表；

服务器列表调度 FSM 状态变迁的原则为：

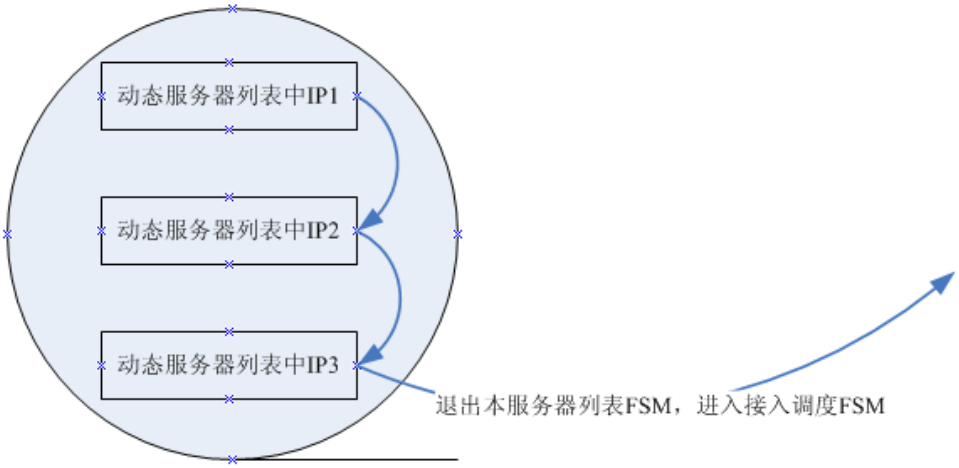
1) APP 冷启动时，服务器列表调度 FSM 状态全部重新置位，按第 2 条原则 执行；

2) 客户端由前到后顺序尝试服务器列表中的数据项，不可逆向执行、不可乱序执行；

3) 客户端尝试一遍本服务器列表所有数据项，如果全部失败，则退出这个服务器列表调度 FSM，进入到接入调度 FSM；

4) 链接建立失败（建议要再做 1 ~ 2 次重试，重试间隔 3 ~ 5s，这两个参数 云端应该可配可控，相关详细讨论可参考 3.1.3 链路管理）或超时、链接 建立成功收发数据错误或超时、服务器下发新的动态服务器列表时，服务 器列表调度 FSM 状态要变迁；

5) 特别的，如果是因为服务器下发新的动态服务器列表导致状态变迁，那么服务器列表调度 FSM 状态要置位还原，重新按第 2 条原则执行；



【图二十 动态服务器列表调度 FSM】

客户端接入调度首要目标是确保可用性，其次是选择最快的链路。客户端无论同哪个集合中哪个服务器列表的接入服务器建立链接，服务器都应按照就快策略的标准评判此时客户端选择的服务器接入点是否符合要求，有没有更快的接入点，如果有，就随着业务数据响应一并下发至客户端，客户端同步更新动态服务器列表的数据，驱动调度 FSM 和服务器列表调度 FSM 发生状态变迁，使得下次再发起服务器访问时能使用更优的接入服务，接入链路切换时机这里有三个方案可供探讨（后续链路管理也会有相关的讨论）：

1）直接关闭当前链路，立即尝试使用新的动态服务器列表建立链接；

2）直接关闭当前链路，当有网络访问时尝试使用新的动态服务器列表建立链接；

3）保持当前链路，立即尝试使用新的动态服务器列表建立链接，一旦成功，马上切换新的业务请求到新链路上，然后在旧链路空闲时将其关闭；

实践中可以根据 APP 的特点来选择链路切换方案。

那么，客户端报告什么样的数据可以作为服务器调度策略计算的依据呢？

1）网络类型，比如 WIFI、2/3/4G 等，WIFI 时多提供一份 SSID 信息；

2）接入 IP 归属，比如电信、联通、移动、海外及其所属省市等，注：归属由服务器判断；

3）目标域名，用于服务端校验访问目标和自己提供的服务是否匹配；

4）访问目标服务时的测速数据 (IO 次数、每次 IO 字节和耗时、RTT 估算值等) 和服务质量数据（如接入调度 FSM 状态、服务器地址、链接成功或失败、链接成功所需时长、链接失败错误码、重试次数等）；

说了半天，这一切的基础是我们要部署足够多和广的服务器接入点，也可以使用 CDN，依托在一个分省市分运营商甚至覆盖全球的 IP 库和通过大量客户端测速报告的业务质量统计数据计算出来的、接入 IP 按就快原则匹配接入服务器的经验调度策略库之上。

总结一下与就快接入相关的内容：

1）服务器分省分运营商分国内外的部署及使用 CDN，广度和深度并举；

2）客户端测速报告及服务质量监控报告，测速这个话题，稍微多探讨一下，在有 线网络，实时测速并调整调度策略数据是非常普通的方案，但放在移动网络条件下，就有重新思考的必要。移动网络易抖动和移动应用大部分短链接轻量交互的特点，使得我们很难在一个短的时间内做出网络速度的有效判断，即便有初步的判断，也可能因为没有马上使用的时机而导致过期失效。因此，我们更倾向

于把这些质量数据报告到后台，通过大量的数据归并分析，形成接入速度调度策略的判断依据；

3) 客户端接入 IP 库与接入服务器就快调度匹配库需要持续更新；

4) 服务器调度中尽量减少 302 跳转，做到一击即中；

3.1.2.2 去 DNS 的 IP 直连

DNS 不但需要 1 个 RTT 的时间消耗，而且移动网络下的 DNS 还存在很多其它问题：

1) 部分 DNS 承载全网用户 40%以上的查询请求，负载重，一旦故障，影响巨大，这样的案例在 PC 互联网也有很多，Google 一下即可感受触目惊心的效果；

2) 山寨、水货、刷 ROM 等移动设备的 LOCAL DNS 设置错误；

3) 终端 DNS 解析滥用，导致解析成功率低；

4) 某些运营商 DNS 有域名劫持问题，实际上有线 ISP 也存在类似问题。域名劫持对安全危害极大，产品设计时要注意服务端返回数据的安全校验（如果协议已经建立在安全通道上时则不用考虑，安全通道可以基于 HTTPS 或者私有安全体系）。对于劫持的判断需要客户端报告实际拉取服务数据的目标地址 IP 等信息；

5) DNS 污染、老化、脆弱；

综上就是在前述就快接入小节中，接入调度 FSM 会优先使用动态服务器列表的原因。

3.1.2.3 网络可达性探测

在连接建立过程中如果出现连接失败的现象，而终端系统提供的网络状态接口反馈网络可用时，我们需要做网络可达性探测（即向预埋的 URL 或者 IP 地址发起连接尝试），以区别网络异常和接入服务异常的情况，为定位问题，优化后台接入调度做数据支持。

探测数据可以异步报告到服务器，至少应该包含以下字段：

1) 探测事件 ID，要求全局唯一不重复；

2) 探测发生时间；

3) 探测发生时网络类型和其它网络信息（比如 WIFI 时的 SSID 等）；

4) 本地调度的接入服务器集合类型；

5) 本地调度的接入服务器 IP（如使用域名接入，可忽略）；

6) 探测的目标 URL 或 IP 地址

7) 本次探测的耗时；

3.1.3. 链路管理

链路就是运肥猪的高速路，就快接入是选路，链路管理就是如何高效的使用这条路。下面是一些实践总结：

3.1.3.1 链路复用

我们在开篇讨论无线网络为什么慢的时候，提到了链接建立时三次握手的成本，在无线网络高时延、频抖动、窄带宽的环境下，用户使用趋于碎片化、高频度，且请求响应又一次性往返居多、较频繁发起等特征，建链成本显得尤其显著。因此，我们建议在链路创建后可以保持一段时间，比如 HTTP 短链接可以通过 HTTP Keep-Alive，私有协议可以通过心跳等方式来保持链路。具体要点建议如下：

1）链路复用时，如果服务端按就快策略机制下发了新的接入动态服务器列表，则应该按照接入调度 FSM 的状态变迁，在本次交互数据完成后，重建与新的接入服务器的 IP 链路，有三个切换方案和时机可选择：
a. 关闭原有链接，暂停网络通讯，同时开始建立与新接入服务器的 TCP 链路，成功后恢复与服务器的网络交互；
b. 关闭原有链接，暂停网络通讯，待有网络交互需求时开始建立与新接入服务器的 IP 链路；
c. 原有链接继续工作，并同时开始建立与新接入服务器的 TCP 链路，成功后新的请求切换到新建链路上，这个方式或可称为预建链接，原链接在空闲时关闭；

2）链路复用时区分轻重数据通道，对于业务逻辑等相关的信令类轻数据通道建议复用，对于富媒体拉取等重数据通道就不必了；
3）链路复用时，如与协议合并（后面会讨论）结合使用，效果更佳；

3.1.3.2 区分网络类型的超时管理

在不同的网络类型时，我们的链路超时管理要做精细化的区别对待。链路管理中共有三类超时，分别是连接超时、IO 超时和任务超时。我们有一些经验建议，提出来共同探讨：
1）连接超时：2G/3G/4G 下 5 ~ 10 秒，WIFI 下 5 秒（给 TCP 三次握手留下 1 次超时重传的机会，可以研究一下《TCP/IP 详解 卷一：协议》中 TC P 的超时与重传部分）；

2) IO 超时：2G/3G/4G 下 15 ~ 20 秒（无线网络不稳定，给抖动留下必要的恢复和超时重传时间），WIFI 下 15 秒（1 个 MSL）；
3) 任务超时：根据业务特征不同而差异化处理，总的原则是前端面向用户交互界 面的任务超时要短一些（尽量控制在 30 秒内并有及时的反馈），后台任务可以长一些，轻数据可以短一些，重数据可以长一些；
4) 超时总是伴随着重试，我们要谨慎小心的重试，后面会讨论；
超时时间宜短不宜长，在一个合理的时间内令当前链路因超时失效，从而驱动调度 FSM 状态的快速变迁，效率要比痴痴的等待高得多，同时，在用户侧也能得到一个较好的正反馈。
各类超时参数最好能做到云端可配可控。

3.1.3.3 优质网络下的并发链路

当我们在 4G、WIFI（要区分是 WIFI 路由器还是手机热点）等网络条件较优时，对于请求队列积压任务较多或者有重数据（富媒体等下载类数据）请求时，可以考虑并发多个链路并行执行。
对于单一重数据任务的多链接并发协同而言，需要服务器支持断点续传，客户端支持任务协同调度；

3.1.3.4 轻重链路分离

轻重链路分离，也可以说是信令和数据分离，目的是隔离网络通讯的过程，避免重数据通讯延迟而阻塞了轻数据的交互。在用户角度看来就是信息在异步加载，控制指令响应反馈及时。
移动端大部分都是 HTTP 短链接模式工作，轻重数据的目标 URL 本身就不同，比较天然的可以达到分离的要求，但是还是要特别做出强调，是因为实践中有些轻数据协议设计里面还会携带类似头像、验证码等的实体数据。

3.1.3.5 长链接

长链接对于提升应用网络交互的及时性大有裨益，一方面用户使用时，节省了三次握手的时间等待，响应快捷；另一方面服务器具备了实时推送能力，不但可以及时提示用户重要信息，而且能通过推拉结合的异步方案，更好的提升用户体验。
长链接的维护包括链接管理、链接超时管理、任务队列管理等部分，设计实施复杂度相对高一些，尤其是在移动网络环境下。为了保持链路还需要做心跳机制（从另外一个角度看，这也是针对简单信息一个不错的 PULL/PUSH 时机，，但需注

意数据传输要够轻，比如控制在 0.5KB 以内），而心跳机制是引入长链接方案复杂度的一个重要方面，移动网络链路环境复杂，国内网关五花八门，链路超时配置各有千秋，心跳时长选择学问比较大，不但要区分网络类型，还得区分不同运营商甚至不同省市，历史上曾经实践了 2 分钟的心跳间隔，最近比较多的产品实践选择 4.5 分钟的心跳间隔。而且长链接除了给移动网络尤其是空中信道带来负担外，移动设备自身的电量和流量也会有较大的消耗，同时还带来后端带宽和服务器投入增加。所以，除了一些粘性和活跃度很高、对信息到达实时性要求很高的通讯类 APP 外，建议谨慎使用长链接，或可以考虑采用下面的方式：

1) 退化长链接：即用户在前台使用时，保持一个长链接链路，活跃时通过用户使 用驱动网络 IO 保持链路可用；静默时通过设置 HTTP Keep-Alive 方式，亦或通过私有协议心跳方式来保持链路。一旦应用切换后台，且在 5~10 分钟内没有网络交互任务则自行关闭链路，这样在用户交互体验和资源消耗方面取得一个平衡点；

2) 定时拉取/询问：对于一些有 PUSH 需求的 APP，我们可以采用一个云端可配置间隔时长的定时拉取/询问方案。有三个重点，一是定时的间隔云端可以配置，下发更新到客户端后下次生效；二是拉取/询问时，如果下发的指令有要求进一步 PULL 时，可以复用已建立的链路，即前述退化长链接的模式；三是定时拉取/询问时机在客户端要做时间上的均匀离散处理，避免大的并发查询带来带宽和负载的巨大毛刺；

3) 如果可能，优先使用 OS 内置的 PUSH 通道，比如 iOS 的 APNS、Andriod 的 GCM（Google 这个以工程师文化著称的公司，在做 OS 级基础设施建设时，却表现出了很差的前瞻性和系统思考的能力，GCM 的前身 C2DM 都没怎么普及使用就被替换了，这也意味着 Android 各种版本 PUSH 能力不 一致的问题。但无论怎么说，OS 级的基础设施无论在性能、稳定性还是在效率上都会优于 APP 层自己实现的方案），实施推拉结合的方案。特别要提到的一点是，中国特色无所不在，国内运营商曾经封过 APNS 的 PUSH 端口 2195，也会干扰 GCM 的端口 5528，更别提这些底层服务的长链接会被运营商干扰。对于 Android 平台，还存在系统服务被各种定制修改的问题。别担心，办法总比问题多，保持清醒；

3.1.3.6 小心重试

自动重试是导致后台雪崩的重要因素之一。在移动网络不稳定的条件下，大量及时的重试不但不能达到预期，反而无谓的消耗移动设备的电量甚至流量。因此，我们在重试前要有一些差异化的考虑：

1）当前移动设备的网络状况如何，如果没有网络，则不必重试；
2）重试设定必要的时间间隔，因为移动接入网络抖动到恢复可能需要一点时间，马上重试并非最佳策略，反而可能无谓的消耗电量。实践中，可以在一次连接或IO失败（立即失败或超时）时，过3～5秒后再试；
3）重试应设定必要的总时限，因为三个服务器列表比较长，每个服务器地址都要重试和等待若干次，最终可能导致接入调度FSM和服务器列表调度FSM流转耗时过长，此时用户侧体验表现为长时间等待无响应。总时限参数可以参考前述区分网络类型的超时管理中的任务超时值。一旦某次重试成功，重试总时限计时器要归零；
4）服务器下发特定错误码（比如服务器故障、过载或高负载）时，提示客户端停止重试并告知安抚用户，我们在强监控这个主题下有详细的讨论；
每个目标服务器地址的重试次数、重试总时限和重试时间间隔最好能做到云端可配可控。

特别需要提出的一点是，移动APP采用HTTP短链接模式实现CS交互时，广泛的使用了系统原生组件或者开源组件，这些友好的模块把超时和重试都封装起来，其缺省值是否适合自己的业务特点，需要多多关注。使用前，最好能知其然更知其所以然。

3.1.3.7 及时反馈

透明和尊重，会带来信任和默契，家庭如此、团队如此、用户亦如此。欲盖弥彰和装傻充愣也许短暂取巧，拉长时间轴来看，肯定要付出惨重的代价。及时和真诚的告知状况，赢得谅解和信任，小付出，大回报，试过都知道。

当发现因为网络不存在或者其它属于移动端设备链路的异常时，应该及时和显著的提示用户，让用户注意到当前有诸如网络不存在、FREE WIFI接入认证页面需确认等等问题，使用户可以及时处理或理解问题状态。

当发现是服务器问题时，应及时、显著和真诚的告知用户，争取用户的谅解。网络异常提示或服务器故障通告等信息的呈现要做到一目了然，无二义和二次交互。
我们在强监控这个主题下有详细的方法讨论。

3.1.4 IO 管理

基于一个快速和高效管理的链路之上，做好 IO 调度和控制，也是提升效能和改善用户体验的重要环节。要探讨的内容包括：

3.1.4.1 异步 IO

异步化 IO 的目的就是避免资源的集中竞争，导致关键任务响应缓慢。我们在后面差异服务个大的分类中会重点探讨。这里特别先提出来，是建议在程序架构顶层设计时，要在整体机制上支持异步化，设计必要的异步总线来联系各个层级模块，总线可能会涉及包括队列管理（优先级、超时、CRUD 等）、事件驱动、任务调度等。

异步 IO 除了网络方面外，对移动设备，我们还特别要考虑一下磁盘 IO 的异步。因为频繁、大吞吐量的磁盘 IO 会造成 APP 的 UI 卡顿，从用户体验上看就是交互响应迟钝或者滑动帧率下降。一般来说，磁盘 IO 异步会选用空间换时间的方案，即缓存数据批量定时写入磁盘。

3.1.4.2 并发控制

有了异步 IO，并发控制就显得尤为重要。把异步机制当作银弹任意使用，就如同我们给移动 APP 设计了一个叫“发现”的地方一样，很可能各种膨胀的需求、不知道如何归类的需求就纷至沓来，期待有朝一日被“发现”。

异步 IO 提供了一个很好的发射后不用管的机制，这就会造成使用者的膨胀，无论是否必要、无论轻重缓急，把请求一股脑的丢给异步队列，自己潇洒的转身就走。这样不但会带来效率和交互响应性能的下降，也会造成资源的无谓消耗。
在后面多异步这个大分类的讨论中会涉及到轻重缓急的话题，在前述异步 IO 的磁盘 IO 的时空效率转换话题中，还应该包括 IO 并发的控制，我们即不能因为并发过多的链路造成网络带宽的独占消耗影响其它 APP 的使用，也不可因快速、大量的异步数据造成缓写机制形同虚设或是占用过大的内存资源。

3.1.4.3 推拉结合

PUSH 机制应该是苹果公司在移动设备上取得辉煌成就的最重要两个机制之一，另外一个是移动支付体系。我们这里的讨论不包括 iOS 和 APPLE 移动设备的拟人化交互体验，只侧重根基性的机制能力。APNS 解决了信息找人的问题，在过去，只有运营商的短信有这个能力，推送和拉取使得我们具备了实时获取重要信息的能力。

为何要推拉结合。因为系统级的推送体系也必须维持一个自己的链路，而这个链路上要承载五花八门的 APP 推送数据，如果太重，一方面会在设计上陷入个性化需求的繁琐细节中，另外一方面也会造成这条链路的拥堵和性能延迟。因此，通过 PUSH 通知 APP，再由 APP 通过自己的链路去 PULL 数据，即有效的利用了 PUSH 机制，又能使得 APP 能按需使用网络，不但简化了链路管理，而且节省了电量和流量。

3.1.4.4 断点续传

一方面，在讨论链路管理时，我们建议了优质网络下的并发链路来完成同一个重数据拉取任务。这就会涉及到任务的拆分和并行执行，基础是后台能支持断点续传。

另外一方面，从客户端的角度而言，移动网络的不稳定特点，可能会造成某个重数据拉取任务突然失败，无论是自动重试还是用户驱动的重试，如果能从上次失效的上下文继续任务，会有省时间、省电量和省流量的效果，想想也会觉得十分美好。

3.2 轻往复

“技”止此尔。强调网络交互的“少”，更应强调网络交互的“简”。 我们在一条高时延易抖动的通道上取得效率优势的关键因素就是减少在其上的往复交互，最好是老死不相往来（过激），并且这些往复中交换的数据要尽量的简洁、轻巧，轻车简从。这个概念是不是有点像多干多错，少干少错，不干没错。
把我们实践过的主要手段提出来探讨：

3.2.1 协议二进制化

二进制比较紧凑，但是可读性差，也因此形成可维护性和可扩展性差、调测不便的不良印象。这也造成了大量可见字符集协议的出现。计算机是 0 和 1 的世界，她们是程序猿的水和电，任何一个整不明白，就没法愉快的生活了。

3.2.2 高效协议

高效的协议可以从两个层面去理解，一是应用层标准协议框架，二是基于其上封装的业务层协议框架，有时候也可以根据需要直接在 TCP 之上把这两个层面合并，形成纯粹的业务层私有协议框架。不过，为了简化网络模块的通讯机制和一些通用性、兼容性考虑，目前大多数情况下，我们都会选择基于 HTTP 这个应用层标准协议框架之上承载业务层协议框架。下面我们针对上述两个层面展开探讨。首先是应用层的标准协议优化，比如 HTTP/1.1 的 Pipeline、WebSocket（在 HTML5 中增加）、SPDY（由 Google 提出）、HTTP/2 等，其中特别需要关注的是处在试验阶段的 SPDY 和草案阶段的 HTTP/2。

SPDY 是 Google 为了规避 HTTP/1.1 暨以前版本的局限性开展的试验性研究，主要包括以下四点：

1）链路复用能力，HTTP 协议最早设计时，选择了一问一答一连接的简单模式，这样对于有很多并发请求资源或连续交互的场景，链路建立的数量和时间成本就都增加了；

2）异步并发请求的能力，HTTP 协议最早的设计中，在拉取多个资源时，会对应并发多个 HTTP 链路（HTTP/1.1 的 Pipeline 类似）时，服务端无法区分客户端请求的优先级，会按照先入先出（FIFO）的模式对外提供服务，这样可能会阻塞客户端一些重要优先资源的加载，而在链路复用的通道上，则提供了异步并发多个资源获取请求指令的能力，并且可以指定资源加载的优先级，比如 CSS 这样的关键资源可以比站点 ICON 之类次要资源优先加载，从而提升速度体验；

3）HTTP 包头字段压缩（注：特指字段的合并删减，并非压缩算法之意）精简，HTTP 协议中 HEAD 中字段多，冗余大，每次请求响应都会带上，在不少业务场景中，传递的有效数据尺寸远远小于 HEAD 的尺寸，带宽和时间成本都比较大，而且很浪费；

4）服务器端具备 PUSH 能力，服务器可以主动向客户端发起通信向客户端推送数据；

HTTP/2 由标准化组织来制定，是基于 SPDY 的试验成果开展的 HTTP 协议升级标准化工作，有兴趣了解详细情况可以参考 HTTP/2 的 DRAFT 文档。

其次是业务层的协议框架优化，它可以从三个方面考察，一是协议处理性能和稳定性好，包括诸如协议紧凑占用空间小，编码和解码时内存占用少 CPU 消耗小计算快等等，并且 bad casae 非常少；二是可扩展性好，向下兼容自不必说，向上兼容也并非不能；三是可维护性强，在协议定义、接口定义上，做到可读性强，把二进制协议以可读字符的形式展示，再通过预处理转化为源码级文件参与工程编译。可能会有同学强调协议调测时的可阅读、可理解，既然读懂 01 世界应该是程序员的基本修养，这一项可能就没那么重要了。

高效的业务层协议框架从分布式系统早期代表 Corba 的年代就有很多不错的实践项目，目前最流行的开源组件应属 ProtoBuf，可以学习借鉴。正所谓殊途同归、心有灵犀、不谋而合，英雄所见略同……，说来说去，高效协议的优化思路也都在链路复用、推拉结合、协议精简、包压缩等等奇技淫巧的范畴之内。

3.2.3 协议精简

协议精简的目的就是减少无谓的数据传输，提升网络效能。俗话说“千里不捎针”，古人诚不我欺也。我们实践总结以下三点供参考：

1）能不传的就不传。把需要的和希望有的数据都列出来，按照对待产品需求的态 度，先砍掉一半，再精简一半，估计就差不多了。另外，高效协议提供了比较好的扩展性，预留字段越少越好，移动互联网演化非常快，经常会发现前瞻的预留总是赶不上实际的需求；

2）抽象公共数据。把各协议共性的属性数据抽象出来，封装在公共数据结构中，即所谓包头一次就传一份，这个想法不新鲜，TCP/IP 的设计者们早就身体力行了。除了带来数据冗余的降低外，还降低了维护和扩展的复杂度，一石二鸟，且抽且行；

3) 多用整数少用字符，数字比文字单纯，即简洁又清晰，还不需要担心英文不好被后继者 BS；

4) 采用增量技术，通知变化的数据，让接收方处理差异，这是个很好的设计思想，实践中需要注意数据一致性的校验和保障机制，后面会有专门的细节讨论；

3.2.4 协议合并

协议合并的目标是通过将多条交互指令归并在一个网络请求中，减少链路创建和数据往复，提升网络效能。把实战总结的六点提出来供参考：

1) 协议合并结合协议精简，效率翻番；
2) 协议合并的基础是业务模型的分析，在分类的基础上去做聚合。首先得区分出来缓急，把实时和异步的协议分类出来分别去合并；其次得区分出来轻重，协议请求或协议响应的数据规模（指压缩后），尽量确保在一个数据报文中可完成推拉；
3) 协议合并在包的封装上至少有两种选择，一是明文协议合并后统一打包（即压缩和解密）；二是明文协议分别打包，最后汇总；前者效率高一些，在实战中用的也较普遍；后者为流式处理提供可能；
4) 协议合并对服务器的异步处理架构和处理性能提出了更高的要求，特别需要权衡网络交互效率和用户对后台处理返回响应期待之间的取舍；
5) 协议间有逻辑顺序关系时，要认真考虑设计是否合理或能否合并；
6) 重数据协议不要合并；

3.2.5 增量技术

增量技术准确分类应该算是协议精简的一个部分，它与业务特点结合的非常紧密，值得单独讨论一下。增量技术在 CS 数据流交互比较大的时候有充分发挥的空间，因为这个技术会带来客户端和服务器计算、存储的架构复杂度，增加资源消耗，并且带来许多保障数据一致性的挑战，当然，我们可以设计的更轻巧，容许一些不一致。

我们用一个案例来看看增量技术的运用。
在应用分发市场产品中，都有一个重要功能，叫更新提醒。它的实现原理很简单，以 Android 设备为例，客户端把用户移动设备上安装的 APP 包名、APP 名称、

APP 签名、APP 版本号等信息发送到服务器，服务器根据这些信息在 APP 库中查找相应 APP 是否有更新并推送到客户端。这个过程非常简单，但如果用户手机上装了 50 个 APP，网络上交互的数据流就非常客观了，即浪费流量和电量，又造成用户体验的缓慢，显得很笨重。
这个时候，增量技术就可以派上用场了，比如下面的方案：

1) 每个自然日 24 小时内，客户端选择一个时间（优先选择驻留在后台的时候）上报一次全量数据；
2) 在该自然日 24 小时的其它时间，客户端可以定时或在用户使用时发送增量数据，包括卸载、安装、更新升级等带来的变化；
3) 作为弱一致性的保障手段，客户端在收到更新提示信息后，根据提醒的 APP 列表对移动设备上实际安装和版本情况做一次核对；
4) 上述择机或定时的时间都可以由云端通过下发配置做到精细化控制；

3.2.6 包压缩

前面精打细算完毕，终于轮到压缩算法上场了。选择什么算法，中间有哪些实战的总结，下面提出来一起探讨：

1) 压缩算法的选择，我们比较熟悉的压缩算法 deflate、gzip、bzip2、LZO、Snappy、FastLZ 等等，选择时需要综合考虑压缩率、内存和 CPU 的资源消耗、压缩速率、解压速率等多个纬度的指标，对于移动网络和移动设备而言，建议考虑使用 gzip。另外需要注意的是，轻数据与重数据的压缩算法取舍有较大差异，不可一概而论；

2) 压缩和加密的先后秩序，一般而言，加密后的二进制数据流压缩率会低一些，建议先压缩再加密；

3) 注意一些协议组件、网络组件或数据本身是否已经做过压缩处理，要避免重复工作，不要造成性能和效率的下降。比如一些图片格式、视频或 APK 文件都有自己的压缩算法。说到这，问题又来了，如果应用层标准协议框架做了压缩，那么基于其上封装的业务层协议框架还需要压缩吗，压缩技术到底哪家强？这个问题真不好回答，考虑到 HTTP/2 这样的应用层标准协议框架定稿和普及尚需时日，建议在业务层协议框架中做压缩机制。或者追求完美，根据后端应用层标准

协议框架响应是否支持压缩及在支持时的压缩算法如何等信息，动态安排，总的原则就是一个字：只选对的，不选贵的；

3.3. 强监控

可监方可控，我们在端云之间，要形成良好的关键运营数据的采集、汇总和分析机制，更需要设计云端可控的配置和指令下发机制。本篇重点讨论与主题网络方面相关关键指标的“监”和“控”。
以就快接入为例来探讨一下强监控能力的构建和使用。

1) 接入质量监控，客户端汇总接入调度 FSM 执行过程元信息以及业务请求响应结果的元信息，并由此根据网络类型不同、运营商不同、网络接入国家和省市不同分析接入成功率、业务请求成功率（还可细化按业务类型分类统计）、前述二者失败的原因归类、接入 302 重定向次数分布暨原因、接入和业务请求测速等；
2) 建设云端可控的日志染色机制，便于快速有针对性的定点排查问题；
3) 终端硬件、网络状态的相关参数采集汇总；
4) 建设云端可控的接入调度（比如接入 IP 列表等）和网络参数（比如连接超时、IO 超时、任务超时、并发链接数、重试间隔、重试次数等）配置下发能力；
5) 服务器根据汇总数据，通过数据分析，结合服务器自身的监控机制，可以做到：
a. 支持细粒度的接入调度和网络参数的优化云控；
b. 支持服务器的部署策略优化；
c. 发现移动运营商存在的一些差异化问题比如 URL 劫持、网络设备超时配置不当等问题便于推动解决；
d. 发现分省市服务器服务质量的异常情况，可以动态云端调度用户访问或者降级服务，严重时可以及时提示客户端发出异常安抚通告，避免加剧服务器的负载导致雪崩。安民告示的快速呈现能力，考验了一个团队对可“控”理解的深度，我们在实践中，提供了三级措施来保障：第一级是服务器端通过协议或跳转 URL 下发的动态通告，这在非 IDC 公网故障且业务接入服务器正常可用时适用；第二级是预埋静态 URL（可以是域名或 IP 形式，优先 IP）拉取动态通告，适用其它故障，静态 URL 部署的 IP 地址最好同本业务系统隔离，避免因为业务服务所在 IDC 公网故障不可用时无法访问；第三级是客户端本地预埋的静态通告文案，内容会比较模糊和陈旧，仅作不时之需；
e. 支持异步任务的云端可配可控，比如下载类 APP 的下载时间、下载标的和下

载条件约束（磁盘空间、移动设备电量、网络类型等）的差异化配置，通过错峰调度，达到削峰平谷并提升用户体验的效果；
特别需要注意的是，客户端数据报告一定要有数据筛选控制和信息过滤机制，涉及用户隐私的敏感信息和使用记录必须杜绝采样上报。在我们的日志染色机制中要特别注意，为了排查问题极可能把关键、敏感信息记录报告到后端，引入安全风险。

3.4. 多异步

经过前面不懈的努力，初步打造了一个比较好的技术根基，好马配好鞍，好车配风帆，怎么就把领先优势拱手送与特斯拉了。

用户欲壑难平，资源供不应求，靠“术”并无法优雅的解决。跳出来从产品角度去观察，还有些什么能够触动我们思考的深度呢。根据不同的需求和使用场景，用有损服务的价值观去权衡取舍，用完美的精神追求不完美，此乃道的层面。
所谓大道至简，完美之道，不在无可添加，而在无可删减。通过多异步和各类缓存机制，提供区分网络、区分业务场景下的差异化服务，是我们孜孜以求的大“道”。
下面通过一些实践案例的总结，来探索简洁优雅的弱联网体验改善之道（开始肆无忌惮的吹嘘了）。

3.4.1【网络交互可否延后】

微博客户端某个版本启动时，从闪屏加载到 timeline 界面需要 6 秒+。这样的体验是无法接受的，与用户 2 秒以内的等待容忍度是背道而驰的。从技术角度去分析，很容易发现问题，诸如我们在启动时有 10+个并发的网络请求（因为是 HTTP 短链接，意味着 10+个并发的网络链接）、闪屏加载、主 UI 创建、本地配置加载、本地持久化数据加载至 Cache 等等程序行为，优化的目标很自然就集中在网络请求和本地配置、持久化数据加载上。

梳理并发网络请求，可以从以下三个方面考察：
1）哪些请求是要求实时拉取的，比如 timeline & 提及 & 私信的数字、身份校验；
2）哪些请求是可以异步拉取的，比如 timeline、用户 Profile、云端配置、双向收听列表、闪屏配置、timeline 分组列表、相册 tag 列表等；
3）哪些请求是可以精简或合并的，比如 timeline & 提及 & 私信的数字与身份

校验合并；
此时，取舍就非常简单和清晰了，启动时 1~2 个网络请求足够应对。所做的仅仅是把一些请求延后发起，这是一种异步机制。
在移动 APP 里面还有大量类似的场景，比如用户更新了 APP 的某个设置项或者自己 Profile 的某个字段，是停在界面上转菊花等网络交互返回后再提示结果，亦或是把界面交互马上还给用户，延后异步向服务器提交用户请求，这里面的价值取向不同，“快”感也便不同。

3.4.2【网络内容可否预先加载】

微博客户端在 timeline 刷新时，用户向上快速滑屏，到达一个逻辑分页（比如 30 条微博消息）时，有两个取舍，一是提前预加载下个分页内容并自动拼接，给用户无缝滑动的体验；二是等到用户滑动到达分页临界点时现场转菊花，卡不卡看当时的网络状况。实践中选择了方案一。用户在滑动浏览第一个逻辑分页时，APP 就利用这个时间窗主动预先拉取下一个逻辑分页的内容，使得用户能享受一个顺畅的“刷”的体验。

所做的仅仅是把一个请求提前发起了，这也是一种异步机制。思考的要点是：
1）预先加载的内容是用户预期的吗，预先加载和自动下载之间，失之毫厘谬以千里；
2）预先加载的内容对用户移动设备的资源（比如流量、电量等）和后端服务器的资源（比如带宽、存储、CPU 等）消耗要做好估算和判断，体贴和恶意之间，也就一步之遥；
3）预先加载区分轻重数据，轻数据可以不区分网络状况，重数据考虑仅限优质网络下执行，最好这些策略云端可以控制；
4）预先通过网络拉取加载或存储的过程中，不要打搅用户的正常使用；
在移动 APP 中，预加载有大量的实践，比较典型的就是升级提醒，大家都采用了先下载好升级包，再提示用户有新版本的策略，让你顺畅到底。

3.4.3【用户体验可否降级】

微博客户端在香港公共 WIFI 下刷新 timeline 总是失败，通过后台用户接入请求和响应日志分析，判断是香港 IDC 到香港公共 WIFI 的汇接口带宽窄、时延大，此时该如何应对。

从前面探讨的 TCP/IP 网络知识，可以知道，在一个窄带宽高时延网络中，吞吐量 BDP 必然很小，也就是说单位大小的数据传输所需的时间会很长。如果按照通常一次下发一个逻辑分页 timeline 数据的策略，那么从服务器到客户端传输，整个数据需要拆分成多个 TCP 数据报文，在缓慢的传输过程中，可能一个数据报文还未传输完成，客户端的链路就已经超时了。

如果在弱网络（需要在应用层有测速机制，类似 TCP/IP 的 RTT 机制，测速时机可以是拉取微博消息数字时）下，把逻辑分页的微博消息数由 30 调整为 5 会如何，如果方案成立，用户刷微博的体验是不是会下降，因为滑动一屏就要做一次网络交互，即便是配合预加载，也可能因为网络太慢，操控太快而又见菊花。外团在香港实测了这个版本，感叹，终于可以刷了。

在饥渴难耐和美酒佳肴之间，似乎还有很多不同层级的体验。聊胜于无，这个词很精准的表述了服务分层，降级取舍的重要性。思考的要点是：
1）产品的核心体验是什么，即用户最在乎的是什么，在做宏观分层设计时要充分保障核心体验；
2）每个产品交互界面中，什么数据是无法容忍短时间不一致的，即什么是用户不能容忍的错误，在做微观分层设计时要充分考虑正确性；
3）在宏观和微观分层的基础上，开始设想在什么条件下，可以有什么样的降级取舍，来保障可用，保障爽快的体验；
4）分层不宜太多太细，大部分产品和场景，3 层足矣；
在移动弱网络条件下，处处可见降级取舍的案例。比如网络条件不佳时，降低拉取缩略图的规格，甚至干脆不自动拉取缩略图等等，分层由心，降级有意。

3.4.4【端和云孰轻孰重】

移动 APP 时代，绝对的轻端重云或者轻云重端都是不可取的，只有端云有机的配合，才能在一个受限的网络通道上做出更好的用户体验。正所谓东家之子，胖瘦有致。
比如移动网游 APP，如取向选择轻端重云，那么玩家的战斗计算就会大量的通过网络递交给服务器处理并返回，卡顿家常便饭，操控感尽失。
比如微博客户端，如果取向选择重端轻云，微博 timeline 所有的消息都拉取元数据（比如微博正文包括文字、各类 URL、话题、标签、@、消息的父子关系、消息中用户 profile、关系链等等），由客户端实时计算拼装，不但客户端用户需

要消耗大量流量计算量，而且给后端服务器带来巨大的带宽成本和计算压力，如果过程中网络状况不佳，还会非常卡顿。

通过实践总结，端和云孰轻孰重，取舍的关键是在数据计算规模可控和数据安全有保障的前提下：
1）减少网络往复，要快；
2）减少网络流量，要轻；
端云有机结合，可以很好的演绎机制与策略分离的设计思想，从而使系统具备足够的柔韧性。

不得不再次特别提到的一点是，缓存技术是异步化的基础，它渗透在性能和体验提升的方方面面，从持久化的 DB、文件，到短周期的内存数据结构，从业务逻辑数据，到 TCP/IP 协议栈，它无所不在。缓存涉及到数据结构组织和算法效能（耗时、命中率、内存使用率等）、持久化和启动加载、更新、淘汰、清理方案等，有机会我们可以展开做专题的介绍。牢记一个字，缓存是让用户爽到极致的利器，但千万别留下垃圾。

提倡多异步，实际上是要求团队认真审视产品的核心能力是什么，深入思考和发现什么是用户最关心的核心体验，把有限的资源聚焦在它们身上。通过考察用户使用产品时的心理模型，体验和还原用户使用场景，用追求完美的精神探索不完美之道。
互联网服务核心价值观之一“不要我等”，在移动互联网时代仍应奉为圭臬，如何面对新的挑战，需要更多的学习、思考、实践和总结，这篇文章即是对过去实践的总结，亦作为面对未来挑战的思考基点。

深入浅出扒一扒 Android Crash 的全过程

腾讯互娱高级工程师-许敏华

导语：大家都知道 C/C++程序出错会引起崩溃，但是知道崩溃发生时系统处理的整个过程的人并不多。深度了解 crash 捕获原理的人估计就更加少了。本文详细的描述了崩溃发生时 android 系统的处理过程和 crash 捕获、堆栈还原的基本原理。

一、哪些情况会让程序崩溃?

在 Unix-like 系统中，所有的崩溃都是编程错误或者硬件错误相关的，系统遇到不可恢复的错误时会触发崩溃机制让程序退出，如除零、段地址错误等。
异常发生时，CPU 通过异常中断的方式，触发异常处理流程。不同的处理器，有不同的异常中断类型和中断处理方式。linux 把这些中断处理，统一为信号量，可以注册信号量向量进行处理。

1.1 ARM 处理器空指针

常见的空指针奔溃时，我们看看 ARM 平台下的 linux 是如何处理的。ARM 支持 7 种异常。空指针属于数据中止异常，尝试读写一个非法内存地址时会发生。

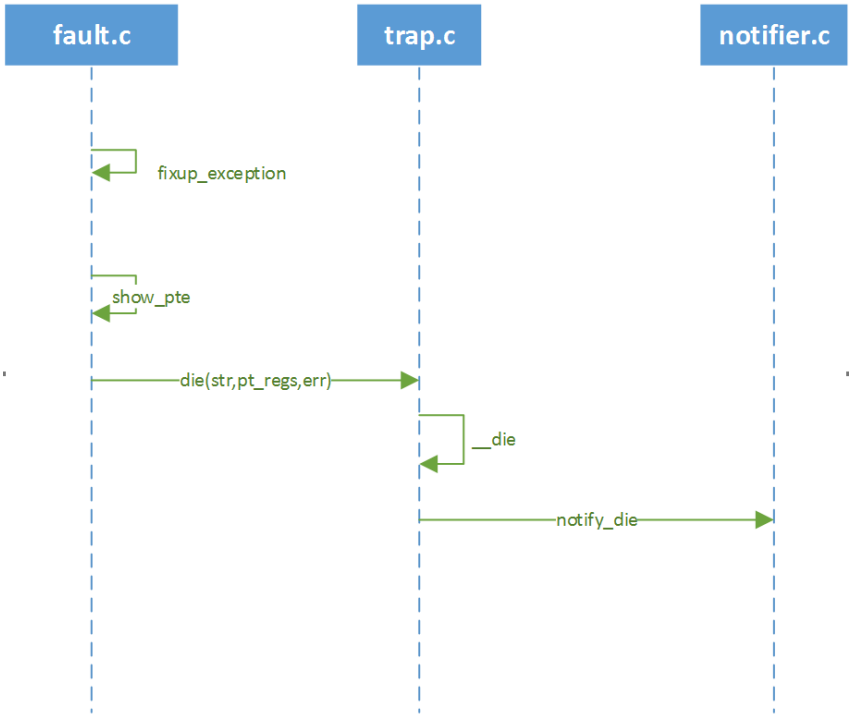
异 常 类 型	处理器模式	异常向量(普通地址)	异常向量(高端地址)	具 体 含 义
FIQ	FIQ	0x0000001C	0xFFFF001C	当处理器的快速中断请求引脚有效，且 CPSR 中的 F 位为 0 时，产生 FIQ 异常
IRQ	IRQ	0x00000018	0xFFFF0018	当处理器的外部中断请求引脚有效，且 CPSR 中的 I 位为 0 时，产生 IRQ 异常
数据中止	中止	0x00000010	0xFFFF0010	访问数据空间异常
预取指中止	中止	0x0000000C	0xFFFF000C	访问指令空间异常
软件中断 SWI	Supervisor	0x00000008	0xFFFF0008	该异常由执行 SWI 指令产生，可用于用户模式下的程序调用特权操作指令
未定义指令	未定义	0x00000004	0xFFFF0004	当 ARM 处理器或协处理器遇到不能处理的指令时，产生未定义指令异常
复位	Supervisor	0x00000000	0xFFFF0000	当处理器的复位电平有效时，产生复位异常

step 1:中断向量调用 **do_kernel_fault**

空指针在 ARM 平台上属于 data abort 异常。当中断发生时，会在中断模式下，保存现场信息，设置相关标志位。然后进入 SVC 模式，根据中断向量表跳转至指定的异常处理函数。内核处理数据中止的模块为 dabt_svc，调用 do_DataAbort，do_DataAbort 调用过程不细表。

Linux 根据 data abort 最后调用的链为 ***arch/arm/mm/fault.c 中 do_DataAbort -> do_translation_fault -> do_page_fault -> __do_kernel_fault***

step 2:do_kernel_fault 及其调用



arm/mm/fault.c 中会尝试进行异常恢复，如果无法恢复 show_pte 打印 page table 相关的信息，然后会调用 arch/arm/kernel/traps.c 中的 die()。如果不是用户模式下的，则报告内核的一个 bug，然后调用 die。__die 会打印基本的错误信息，然后将信号量设置为 SIGSEGV，调用 kernel/notifier.c 中的 notify_die，调用 die_chain 上注册的回调函数。

1.2 软件异常中断

arm/mm/fault.c中的arm_syscall，如果传入的no未0，同样也会抛出SIGSEGV错误。

```
asmlinkage int arm_syscall(int no, struct pt_regs *regs)
{
    struct thread_info *thread = current_thread_info();
    siginfo_t info;
    if ((no >> 16) != (__ARM_NR_BASE>> 16))
        return bad_syscall(no, regs);
    switch (no & 0xffff) {
    case 0: /* branch through 0 */
        info.si_signo = SIGSEGV;
        info.si_errno = 0;
        info.si_code  = SEGV_MAPERR;
        info.si_addr  = NULL;
        arm_notify_die("branch through zero", regs, &info, 0, 0);
        return 0;
        ......
}

void arm_notify_die(const char *str, struct pt_regs *regs,
        struct siginfo *info, unsigned long err, unsigned long trap)
{
    if (user_mode(regs)) {
        current->thread.error_code = err;
        current->thread.trap_no = trap;
        force_sig_info(info->si_signo, info, current);
    } else {
        die(str, regs, err);
    }
}
```

1.3 哪些信号量会让程序 crash

debuggerd.c 是 Android 自带的的程序异常退出诊断的程序，在侦测到程序奔溃时将进程状态输出，供开发人员使用。Android 的连接程序 linker 会通过 debugger_init（）注册信号处理函数。

```
void debugger_init()
{
    struct sigaction act;
    memset(&act, 0, sizeof(act));
    act.sa_sigaction = debugger_signal_handler;
    act.sa_flags = SA_RESTART | SA_SIGINFO;
    sigemptyset(&act.sa_mask);
    sigaction(SIGILL, &act, NULL);
    sigaction(SIGABRT, &act, NULL);
    sigaction(SIGBUS, &act, NULL);
    sigaction(SIGFPE, &act, NULL);
    sigaction(SIGSEGV, &act, NULL);
#if defined(SIGSTKFLT)
    sigaction(SIGSTKFLT, &act, NULL);
#endif
    sigaction(SIGPIPE, &act, NULL);
}
```

信号量	Value	描述	例子
SIGABRT	6	进程发现错误或者调用了abort();	很多C的库函数中，如果发现异常会调用abort。如strlen
SIGBUS	10,7,10	不存在的物理地址，硬件有误。	更多的是因为硬件或者系统引起的。
SIGFPE	8	浮点数运算错误	如除0操作，余0，整形溢出
SIGILL	4	非法指令	损坏的可执行文件或者代码区损坏
SIGSEGV	11	段地址错误	空指针，访问不存在的地址空间，访问内核区，写只读空间。栈溢出，数组越界，野指针。
SIGSTKFLT	16	//未使用	
SIGPIPE	13	管道错误，往没有reader的管道中写。	Linux 中的 socket，如果断掉了继续写。signal(SIGPIPE, SIG_IGN);

Google的Breakpad认为，不认为只有Action为Core（终结程序并输出dump）

的是崩溃的信号量，SIGPIPE 是 Term(只终结程序)，所以不认为是 crash 信号。其实对用户来说，这个就是 crash，应该捕获。

二、如何捕获异常

2.1 sigaction 信号注册函数

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act,struct sigaction *oldact);
```

signum：可以是除了 SIGKILL 和 SIGSTOP 外的所有信号量

act:act 如果不是 null,则设置当前为新的信号处理函数。如果 oldact 是非 Null 则保存老的处理函数到该指针。

```
struct sigaction {
   //sa_handler 和 sa_sigaction 只能存一，sa_flags 设置为 SA_SIGINFO 则为后者
       void     (*sa_handler)(int);
       void     (*sa_sigaction)(int, siginfo_t *, void *);
       sigset_t   sa_mask;
       int        sa_flags;
       void     (*sa_restorer)(void);//非用户使用
};
```

sa_handler:sa_handler 可以注册默认的 SIG_DFL 或者 SIG_IGN 忽略，也可以注册只有一个 signum 为参数的处理函数。

sa_mask:设置屏蔽信号量，该信号处理函数时，其他信号量应该被屏蔽。如果 SA_NODEFER 被设置则无效。

sa_flags：修改信号量的处理行为。

SA_ONSTACK:在 sigalstack(2)提供的栈上面运行。如果被选的栈不可用，则在默认的栈上运行。该参数只有在建立信号量处理函数时有用。

SA_SIGINFO：使用 sa_sigaction 而不是，sa_handler。该函数可以设置和查询指定信号量的处理函数。sigaction(SIGSEGV, NULL, &older_handler_tmp)可以获取到老的信号处理函数，sigaction(SIGSEGV, &new_handler, NULL)注册 new_handler 为 SIGSEGV 函数。

2.2 如何注册信号处理函数

2.2.1 设置额外栈空间

我们在前面提到 SIGSEGV 很有可能是栈溢出引起的，如果在默认的栈上运行很有可能会破坏程序运行的现场，无法获取到正确的上下文。因此，我们应该开辟一块新的空间作为运行信号处理函数的栈。只要在注册时，将 sa_flags 设置为 SA_ONSTACK 即可。

创建时，可以先查看是否已经有创建 sigal 栈空间。如果没有，或者创建的大小太小，则需要创建一块足够大的栈空间。

```
stack_t old_stack,new_stack;
if (sigaltstack(NULL, &old_stack) == -1 || !old_stack.ss_sp ||
      old_stack.ss_size < SIGSTKSZ) {
    new_stack.ss_sp = calloc(1, SIGSTKSZ);
    new_stack.ss_size = SIGSTKSZ;

    if (sys_sigaltstack(&new_stack, NULL) == -1) {
      free(new_stack.ss_sp);
      return;
    }
  }
```

2.2.2 注册信号处理函数

```
void SignalHandler(int sig, siginfo_t* info, void* uc){
    ......
}

struct sigaction sa;
memset(&sa, 0, sizeof(sa));
sigemptyset(&sa.sa_mask);
sigaddset(&sa.sa_mask,[signal_value]);//将所有注册的信号均加入信号集，不然 sigaction()中的参数内容可能是未定义的。
sa.sa_sigaction = SignalHandler;
sa.sa_flags = SA_ONSTACK | SA_SIGINFO;//使用sa_sigaction和其他栈上运行
```

sigaction([signal_value], &sa, NULL);

兼容其他 signal 处理

sigaction 一个信号量只能注册一个处理函数，这意味着我们的处理函数会覆盖其他人的处理信号。集成我们的异常检测程序的同时，程序也可能会集成其他应用的异常检测程序。

在前面，我们提到 sigaction 既可以注册也可以获取到原有的处理函数。如果，我们保存老的处理函数，在处理完我们的信号处理函数后，在重新运行老的处理函数就能完成兼容。

因为，我们的处理函数可能会被覆盖掉。所以，我们需要允许我们的程序多次注册而不会出错。其实，检测到老的处理函数就是自己注册的处理函数时，返回不做处理即可。

```
struct sigaction older_handler_tmp;
if (sigaction([signal_value], NULL, &older_handler_tmp) == -1){
    return false;
}

//现在处理程序为自己定义的函数，则无需进一步处理

if(older_handler_tmp.sa_sigaction==SignalHandler){
    LOG_DEBUG("Wetest CrashMonitor has registered");
    return false;
} else{
    sigaction([signal_value], NULL, &old_handlers);
}

//注册自己的处理函数
```

在我们自己的信号处理函数中，调用玩处理后，我们需要重新注册回老的处理函数。信号量将会重新触发老的处理函数。

```
if (sigaction([signal_value], &old_handlers, NULL) == -1) {
    //注册出错，则尝试注册系统默认的处理函数。
}
```

这样，我们就完成了插入，我们自己的 crash 又不影响到其他 crash 检测模块的

运行。**这里有一个巨大的风险，就是当捕获到 crash 的时候，你的 crash 捕获模块不会退出，是否退出完全交给上一个注册的该信号量的函数。**

三、如何还原崩溃现场

在崩溃的瞬间，如果能够提供崩溃的原因、崩溃点（崩溃的代码所在行）、调用栈及崩溃点附近的变量这是最理想的，这样开发人员就能在最短的时间内定位到问题。在这方面 Java、C#等语言，相对处理的比较好，但是 C 和 C++在这方面表现较差。

3.1 获取崩溃原因

崩溃时注册的信号量处理函数 sa_sigaction 会被调用,void (*sa_sigaction)(int, siginfo_t*, void *)。这个函数调用中分别提供信号量、siginfo_t 结构体和一个指针。

siginfo_t:

```
siginfo_t {
  int     si_signo;    /* Signal number */
  int     si_errno;    /* An errno value，Linux 中通常无用*/
  int     si_code;     /* Signal code */
```

文档查看具体原因，如 SEVG_MAPPERR.如果是<0 的表示内核，

```
      ..... }
```

信号量	错误码	描述
SIGABRT		
SIGBUS	BUS_ADRALN	地址无法对齐
	BUS_ADRERR	物理地址有误
	BUS_OBJERR	
	BUS_MCEERR_AR	
	BUS_MCEERR_AO	
SIGFPE	FPE_INIDIV	int除0
	FPE_INTOVF	int溢出
	FPE_FLTDIV	浮点数除0
	FPE_FLTOVF	浮点数溢出
	FPE_FLTUND	浮点数下溢
	FPE_FLTRES	浮点数不精确
	FPE_FLTINV	非法浮点数操作
	FPE_FLTSUB	下表越界
SIGILL	ILL_ILLOPC	Illegal opcode.
	ILL_ILLOPN	Illegal operand
	ILL_ILLADR	Illegal addressing mode
	ILL_ILLTRP	Illegal trap
	ILL_PRVOPC	Privileged opcode
	ILL_PRVREG	Privileged register
	ILL_COPROC	Coprocessor error
	ILL_BADSTK	Internal stack error
SIGSEGV	SEGV_MAPERR	地址无效
	SEGV_ACCERR	地址无访问权限
SIGSTKFLT		
SIGPIPE		

不同的信号量，对应不同的错误码，每个错误码均有自己的含义。根据这些信息，大致能够提供崩溃的错误原因。Fatal signal 11 (SIGSEGV) code 1 (SEGV_MAPERR)这种类型的日志，也是我们最为常见的。

3.2 获取崩溃所在行

如果，我们能够知道崩溃时的 pc，就能知道崩溃时执行的是那条指令。sa_sigaction 在调用时，还会提供第三个指针参数，ucontext_t。在官方文档中，说这是一个很少使用到的参数，这里却恰巧提供了我们所需的内容。

在<ucontext.h>中定义了两个类型 mcontext_t 和 ucontext_t。mcontext_t 是与机器相关的，并且不透明的。ucontext_t 类型至少包含以下信息。

```
typedef struct ucontext {
        struct ucontext *uc_link;
        sigset_t        uc_sigmask;
        stack_t         uc_stack;
        mcontext_t      uc_mcontext;
        ...
} ucontext_t;
```

sigset_t 和 stack_t 定义在<signal.h>。uc_link 指向另一个 ucontext_t，当前上下文被毁时，可以恢复。uc_sigmask 是信号量屏蔽的集合。uc_stack 是当前使用的栈地址。uc_mcontext 是 cpu 相关的上下文，包括当前线程的寄存器信息。uc_context 如果是信号量获取到的，获取后程序是处于中断状态，还是继续执行并没有详细描述(因此，crash 进程的现场可能是会被破坏掉的)。

3.2.1 ARM 寄存器

如果是手机上市 ARM 的 CPU，那么 uc_context 保存即为 ARM 相关的内容。ARM 处理器包含 31 个通用寄存器，包括程序计数器 pc。ARM 处理器在 7 中模式（用户模式，系统模式，特权模式，中止模式，未定义指令，外部中断模式，快速中断模式）。

未备份（R0-R7）：所有模式公用。异常中断切换时，现场会被破坏。

备份(R8-R14):两个寄存器，FIQ 使用独有寄存器。（FIQ，快速中断，比如说数据读取）。R9，称之为 SB 栈底指针；R10，称之为 SL 栈限制;R11，称之为 FP 栈帧指针;R12，称之为 IP 内部过程调用寄存器；R13，通常称之为堆栈指针（SP），

会指向当前模式独有的堆栈。异常模式时，可以把通用寄存器压入堆栈，返回时出栈，保证完成性。R14，称之为连接寄存器，保存子程序返回地址。异常模式时，可以保存异常返回地址，如处理嵌套中断。

程序计数器 pc：PC 寄存器，总是存储下一条指定的地址。

程序状态寄存器（CPSR）：保存当前程序状态寄存器。最后 5 位，状态。

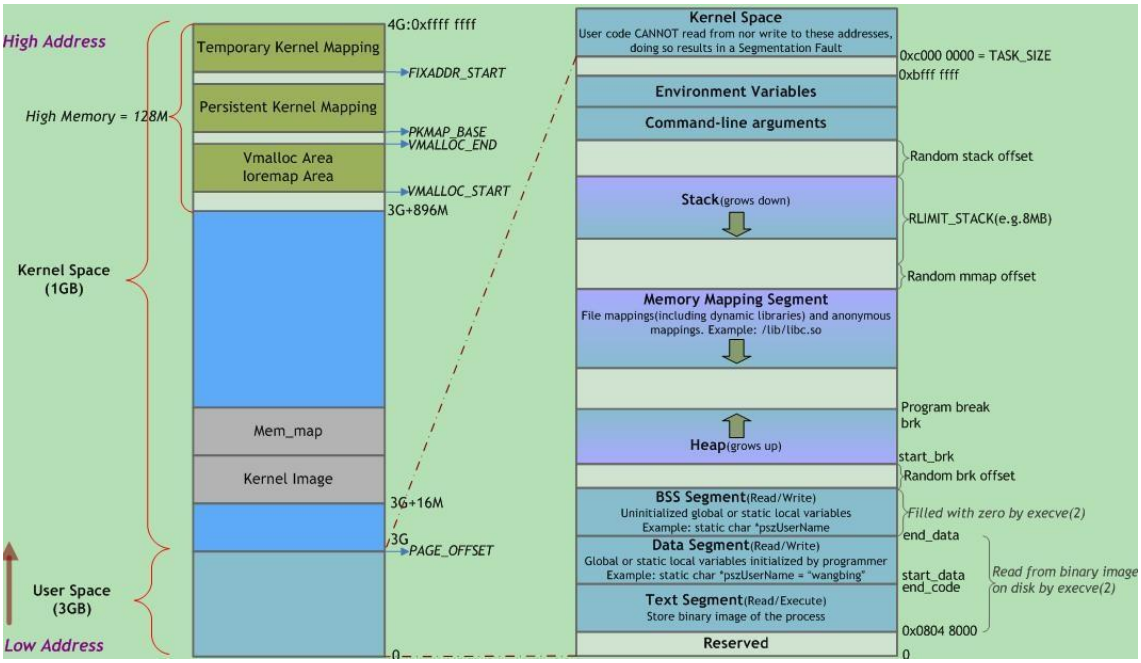
CPSR 处理器模式位

M[4:0]	处理器模式	可访问的寄存器
0b10000	User	PC, R14～R0, CPSR
0b10001	FIQ	PC, R14_fiq～R8_fiq, R7～R0, CPSR, SPSR_fiq
0b10010	IRQ	PC, R14_irq～R13_irq, R12～R0, CPSR, SPSR_irq
0b10011	Supervisor	PC, R14_svc～R13_svc, R12～R0, CPSR, SPSR_svc
0b10111	Abort	PC, R14_abt～R13_abt, R12～R0, CPSR, SPSR_abt
0b11011	Undefined	PC, R14_und～R13_und, R12～R0, CPSR, SPSR_und
0b11111	System	PC, R14～R0, CPSR（ARM v4及更高版本）

3.2.2 进程内存获取代码奔溃处

通过寄存器中的 PC 程序计数器我们就可以知道，程序崩溃时执行的指令地址。不过，这个地址并不是我们能够阅读的。我们需要通过进程内存解析出，我们能够阅读的信息。

任何一个程序通常都包括代码段和数据段，这些代码和数据本身都是静态的。程序要想运行，首先要由操作系统负责为其创建进程，并在进程的虚拟地址空间中为其代码段和数据段建立映射。光有代码段和数据段是不够的，进程在运行过程中还要有其动态环境，其中最重要的就是堆栈。图 3 所示为 Linux 下进程的地址空间布局：



其中，用户地址空间中的蓝色条带对应于映射到物理内存的不同内存段，灰白区域表示未映射的部分。这些段只是简单的内存地址范围，与处理器的段没有关系。上图中 Random stack offset 和 Random mmap offset 等随机值意在防止恶意程序。Linux 通过对栈、内存映射段、堆的起始地址加上随机偏移量来打乱布局，以免恶意程序通过计算访问栈、库函数等地址。execve(2)负责为进程代码段和数据段建立映射，真正将代码段和数据段的内容读入内存是由系统的缺页异常处理程序按需完成的。另外，execve(2)还会将 BSS 段清零。

用户进程部分分段存储内容如下表所示(按地址递减顺序)：

名称	存储内容
栈	局部变量、函数参数、返回地址等
堆	动态分配的内存
BSS段	未初始化或初值为0的全局变量和静态局部变量
数据段	已初始化且初值非0的全局变量和静态局部变量
代码段	可执行代码、字符串字面值、只读变量

在将应用程序加载到内存空间执行时，操作系统负责代码段、数据段和 BSS 段的加载，并在内存中为这些段分配空间。栈也由操作系统分配和管理；堆由程序员自己管理，即显式地申请和释放空间。

BSS 段、数据段和代码段是可执行程序编译时的分段，运行时还需要栈和堆。PC 寄存器的内容必然是指向代码段（libc.so 在映射区）的，获取进程的虚拟地

址空间后，我们就能够解析出代码段所对应的库或者是程序本身(比如，libc.so)。Linux 程序会把对应进程的虚拟地址空间结构以文件的方式保存在/proc/[pid]/maps 下面。

该文件有 6 列，分别为

地址：库在进程里地址范围

权限：虚拟内存的权限，r=读，w=写,x=,s=共享,p=私有；

偏移量：库在.so 文件里的地址范围

设备：映像文件的主设备号和次设备号；

节点：映像文件的节点号；

路径: 映像文件的路径（如 lib/libc.so）。

各共享库的代码段，存放着二进制可执行的机器指令，是由 kernel 把该库 ELF 文件的代码段 map 到虚存空间；

各共享库的数据段，存放着程序执行所需的全局变量，是由 kernel 把 ELF 文件的数据段 map 到虚存空间；

用户代码段，存放着二进制形式的可执行的机器指令，是由 kernel 把 ELF 文件的代码段 map 到虚存空间；

用户数据段之上是代码段，存放着程序执行所需的全局变量，是由 kernel 把 ELF 文件的数据段 map 到虚存空间；

用户数据段之下是堆(heap)，当且仅当 malloc 调用时存在，是由 kernel 把匿名内存 map 到虚存空间，堆则在程序中没有调用 malloc 的情况下不存在；

用户数据段之下是栈(stack)，作为进程的临时数据区，是由 kernel 把匿名内存 map 到虚存空间，栈空间的增长方向是从高地址到低地址。

自己写了一个，会产生 nullptr 的代码，然后运行

```
char* str=0;
str[0]='1';
```

```
  r0 00000016  r1 d2e10014  r2 00000031  r3 00000000

  r4 7a2cf210  r5 4003188d  r6 7afbea40  r7 00000000

  r8 00000000  r9 77e55228  sl 7d273950  fp 77e54e20

  ip 00000003  sp 77e54e10  lr 400383c1  pc 7a2cdfcc  cpsr 600f0030
```

```
cat /proc/26478/maps | grep libnativecpp
7a2cd000-7a2d0000 r-xp 00000000 b3:1a 265232     /data/app-lib/com.tencent.buggame-90/libnativecpp.so
7a2d0000-7a2d1000 r--p 00002000 b3:1a 265232     /data/app-lib/com.tencent.buggame-90/libnativecpp.so
7a2d1000-7a2d2000 rw-p 00003000 b3:1a 265232     /data/app-lib/com.tencent.buggame-90/libnativecpp.so
```

3.3 调用栈获取

通过解析 lib*.so 的异常处理机制，来完成堆栈的还原。

四、子进程输出堆栈

在 crash 的进程中输出堆栈信息，存在两个问题。1、crash 进程的关键数据或堆栈可能已经遭到破坏，在当前线程中输出堆栈信息存在风险；2、signalaction 函数里面使用的函数有非常大的限制。

Crash 进程仅仅简单的创建子进程，子进程能够共享 crash 进程的地址空间、文件资源、信号处理函数等。通过 ptrace 获取 crash 进程信息。

4.1 clone

```
#include <sched.h>

 int clone(int (*fn)(void *), void *child_stack,int flags, voi
d *arg, ...
/* pid_t *ptid, struct user_desc *tls, pid_t *ctid */ );
```

以类似于 fork 的方式创建一个新的进程。不同于 fork，clone 允许子进程共享父进程的部分上下文，如内存、文件描述符表、信号量处理表等。

其中一个应用是，多个线程跑在一个内存空间里面。

fn:子线程创建完成后，会执行 fn 函数。fn 是在子进程执行前执行。当 fn 返回时，子进程将会终结。fn 返回的 int，是子进程的退出码。子进程也可以显式的调用 exit 或者处理错误信号量来退出。

child_stack：该参数用于设定子进程的堆栈。虽然，子进程可以与父进程共享内存，但是在共享同一个栈是不允许的。所以，必须要指定该栈区。

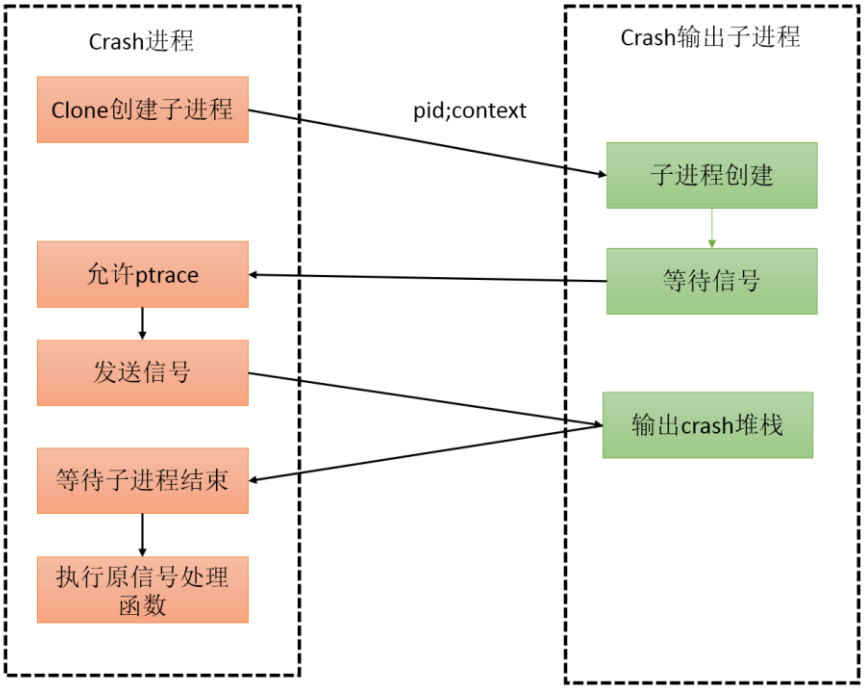
flags:中的低字节保存子进程结束码。如果信号不是 SIGCHLD，那么父进程必须指定_WALL 或者__WCLONE 选项。如果，没有选项设置，则子进程退出时，父进程不会有任何信号量。

CLONE_FILFS：父子进程共享文件描述符列表。父进程或者子进程创建的文件描述符，两边都能使用。如果进程调用 execve，描述符表将不再共享。

CLONE_FS:父子进程共享文件系统信息。共享的文件系统包括，现在的工作目录，权限。任何改变都会影响到其他进程。

CLONE_UNTRACED:如果该参数被指定，将不能再为 CLONE_PTRACE。如果是 CLONE_PTRACE,则父进程将被追踪，子进程也是。

4.2 子进程创建过程



创建子进程之后，需要在 crash 进程设置为允许 ptrace，通过管道的方式模拟信号。创建子进程的时候必须要设置为 CLONE_FILFS、CLONE_FS 能够共享父进程的文件目录，这样才能获取到对应的.so 共享文件。

《天天酷跑》游戏同步方案分析

腾讯互娱高级工程师-周长寿

伴随着手游浪潮，实时 PK 已成为很多游戏的标配。但由于网络延迟的存在，玩家本地状态、在服务器的状态和在其他玩家机器上的状态，三者不可能一致。同步问题本质上是一致性问题，游戏中不同玩家看到的状态需要是一致的。假如游戏中 A、B 两个玩家移动，并同时向对方发出射击的指令。如果没有合适的同步机制，那么可能出现的情况有：A 屏幕显示 B 已经被杀死，B 屏幕显示 A 已经被杀死。因此需要有同步机制来解决由于网络延迟和渲染延迟等造成的玩家状态不一致问题，给玩家更好的游戏体验。同步机制有许多种，根据游戏类型、技术条件甚至时代背景的不同，选择的同步机制也会不同。同步机制的选择，还可能涉及服务器架构的调整。

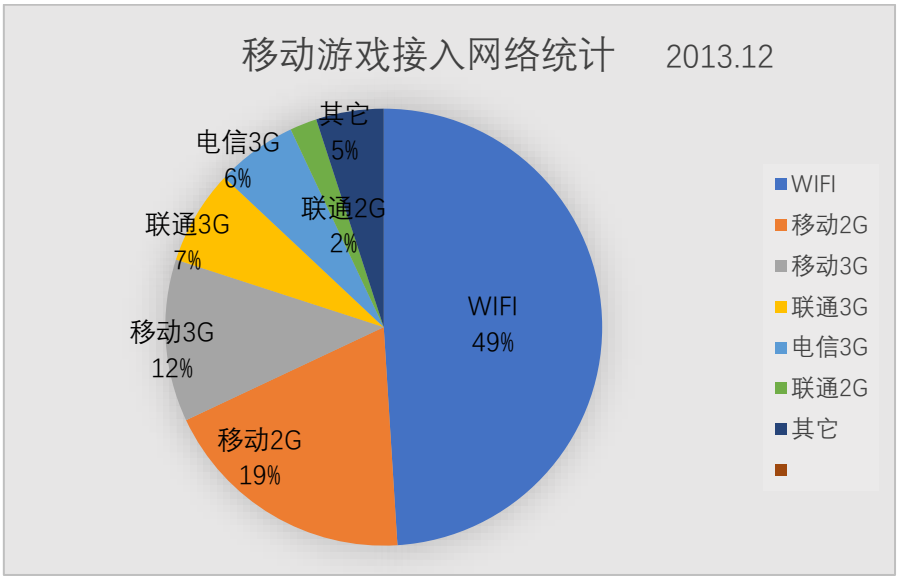
网络游戏同步，常用的两种方案是帧同步和状态同步。
帧同步是同步玩家的指令，服务器负责转发客户端的操作，每个客户端以固定的逻辑帧执行所有客户端的操作指令，通过在严格一致的时间轴上执行同样的命令序列获得同样的结果。
状态同步跟帧同步的最大区别是服务器不在进行切逻辑帧，而是同步玩家状态信息，比如位置、属性、跟玩法相关的数据。通常主逻辑在服务器运行，客户端只是作为一个显示。采用状态同步的游戏有 CFM、LOL 等。
帧同步的网络流量较小，但防外挂、断线重连的难度比较大。状态同步中服务器有所有玩家的状态，安全性较高，游戏运营更可控。是否选择状态同步，需要看同步的实体数量。在大场景中，同步的单位比较多时，往往会放弃状态同步。比如星际争霸中玩家可操作的实体多达上百个，如采用状态同步的话网络流量将非常大。

《天天酷跑》(以下简称“酷跑”) 是一款横版跑酷游戏，先后推出了多种 PVP 玩法，包括世界对战、多人战、团队赛等等。下面介绍游戏几种典型 pvp 玩法在服务器端的同步方案和对应的架构调整。

一、世界对战

为了增加玩家之间的互动，酷跑上线 3 个月后，规划了世界对战的玩法。具体玩法是：参与游戏的玩家两两匹配进行对战，最后根据比赛分数、距离等决定胜负。

酷跑作为一个全民游戏，必须考虑玩家的网络状况。下图是酷跑世界对战推出时移动游戏接入网络的统计。2G/3G 网络比例很高，网络质量不稳定，且都是按照流量收费。游戏过程中大规模的数据同步在当时还是不适合的。同时早期移动游戏的人力投入都比较小，版本迭代节奏也很快。



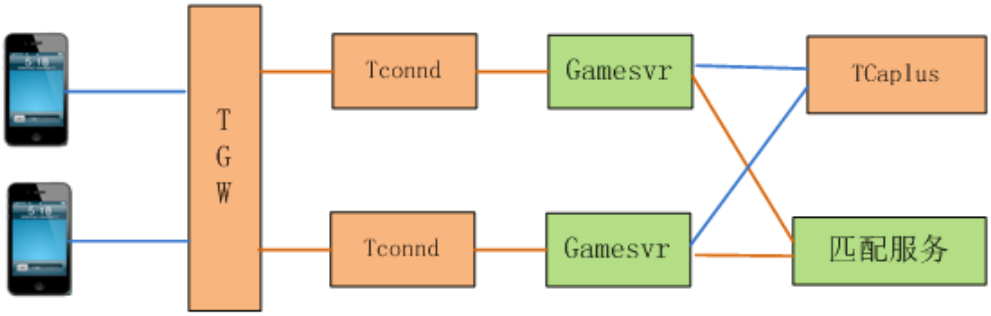
因此世界对战玩法提出时，酷跑选择了简单的同步方案——通过 TCaplus 数据库（腾讯自研 NOSQL 数据库）来实现玩家之间的数据同步。世界对战属于异步 PVP，为了节省流量，玩家在游戏过程中没有交互，只在重要节点同步。包括匹配成功、开局、结算、领奖，同步数据量小、协议交互频率低。同时 TCaplus 数据库具有很高的读写性能，并且有专业的运维团队维护，使得开发团队的工作量相对较小，又能满足世界对战的同步需求。

服务器设计了 TCaplus 数据库表 TB_PVPMATCH，用于记录玩家世界对战数据。每次参加世界对战会在 TB_PVPMATCH 表中增加一条记录，记录中定义了一个 status 字段，用于记录玩家找到对手、确认对手、付入场费、上报分数、领奖状态的变迁。

玩家匹配成功后会新增一条记录到 TB_PVPMATCH 表中，status 状态标记为初始化。匹配成功后，客户端会固定每秒 1 次的频率发送对战状态协议到服务器，直到双方都扣入场费后开局。服务器收到拉取对战状态协议后，从 TCaplus 中获取 TB_PVPMATCH 表对应自己和对手的记录，根据双方的状态更新自己的记录并写入 TCaplus 和发送客户端。当客户端拉取的响应消息中标记入场费已扣除则开局，开始游戏。

游戏结束后，再次通过对战状态协议通知游戏已结束，服务器获取本局游戏数据并更新到 TB_PVPMATCH 表对应的记录中。如果对方已结束游戏，则可以直接结算。如果对方还在游戏中，玩家可以选择在结算界面等待对方或直接退出，下次再查看比赛结果。在结算的时候，或拉取对战双方在 TB_PVPMATCH 表中的记录，根据双方数据来判定胜负。

世界对战版本服务器架构



世界对战上线时，服务器架构比较简单。全局服务只有匹配服务，涉及的异步流程很少，其它逻辑在 Gamesvr 上处理即可。

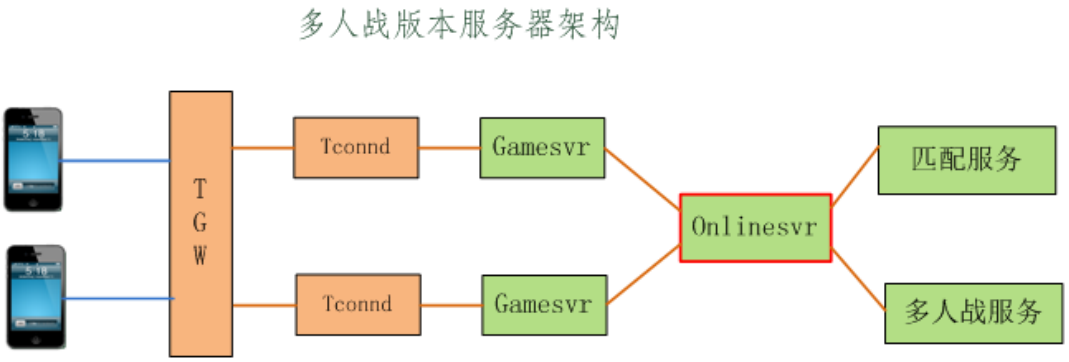
二、多人战

2014 年底酷跑推出了实时同屏多人对战模式的玩法。玩家要努力战胜对手，将速度最大化，用最短的时间跑完全程。多人战分经典战和道具战两种，经典战比拼速度，道具战通过道具增加或减少各种效果，玩家可以利用各种功能各异的道具保护自己，干扰对手。

多人战属于同步 PVP，4 个玩家匹配到一起后即可开局。由于对战中玩家数不多，同时服务器需要做一些逻辑，这里选用了状态同步的同步方案。游戏过程中玩家上报自己的操作、位置和状态给服务器，服务器做一些逻辑处理，再把该玩家上报的包广播给其他玩家，其他玩家收到包后，知道该玩家的状态，做相应的逻辑。游戏过程中，根据游戏场景，动态调节同步频率，达到节省同步包量，降低服务器负载的目的。玩家在游戏中平均每秒 1 次左右的同步。

版本灰度上线过程中，发现多人战服务所在机器的通讯进程进程 TBus CPU 占用很高。多人战服务是用来创建房间、同步包转发、对战结算的服务。需要与所有 gamesvr 和 TConnd 建立通道，单机 Tbus 通道达到上千个。

TBusd 占 CPU 高，但网络流量又不大。最后发现 Tbus 通道达到一定数目，性能会急剧下降。Tbusd 需要轮询所有通道进行收发包，通道过多时导致大量 CPU 空转。考虑到后续全局服务会越来越多，通道数会进一步膨胀，Tbus 占用的内存、CPU 会进一步上升。需要降低通道数，引入中转服务，可以使通道数收敛，同时前后端关系也解耦。



因此对架构进行了调整，逻辑层由 2 层变成 3 层，中间加了一个转发层，gamesvr 与全局服务，全局服务之间不再直接通信，而是通过转发层中转，这样 Tbus 通道数得到了收敛，服务器之间的通信关系由网状变成了树状。

三、团队战

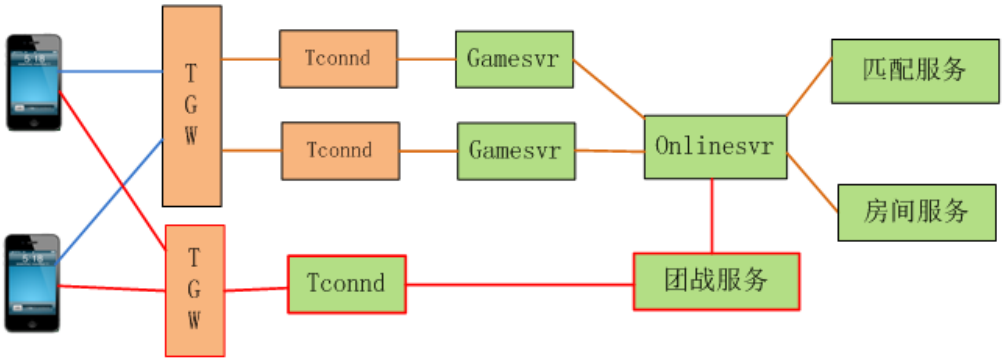
2017 年初，为了加强玩家间的互动，设计了局内交互更频繁的团队战模式，强化玩家间配合，增加玩家对游戏的粘性。团队战是 3V3 对战，在比赛结束时，计算队友获得的总分，得到总分较高的团队获胜。玩家在挑战时，队友之间相互有联系。比赛中每过一段时间玩家就会进入到一个小游戏，如在极速前进中需要配合完成吃金币和空心银币的任务、在进击模式中需共同完成砍怪、在天空之城中队友之间分配合理的飞行线路可以获得较高的分数、队友通过巅峰挑战会落下高分流星雨作用于团队所有的队员等。

这是一个强交互的需求，队友间需要共享妖怪总血量、关卡中出现的金币等，给妖怪最后一击的只能是一个玩家、所有队友吃到的金币总和不能超过金币总量等。最初要求每秒 15 帧的数据同步，同时服务器需要做逻辑，因此团队战选择了状态同步的方案。服务器采取了简单的处理原则，先来先处理，处理时加上校验即

可。同时针对不同关卡，动态调整同步频率。

团队战需要有团战服务来同步玩家数据，原有架构上下行包需要在客户端、Gamesvr 和团战服务之间流转，团队战中高频的同步请求包对流量和 CPU 都会带来挑战。因此对架构进行了调整，将同步包独立出来，游戏过程中客户端直接与团战服务通讯。

团队赛版本服务器架构



这是目前的游戏架构，增加了客户端与团战服务直接通讯的能力。3v3 匹配成功后，通知团战服务建立房间，同时下发客户端房间所在团战服务的机器 ip 和端口（映射到 tgw 的 vip 和端口），客户端再通过下发的地址信息连接团战服务。游戏过程中，客户端直接向团战服务发同步包，团战服务转发给队伍中的其他玩家。

总体来说，酷跑定位为轻度游戏，PVP 中涉及的同步方案比较简单。在不同时期，根据游戏玩法、网络状况等选择了不同的同步机制和服务器架构，满足了设计需求。

《仙剑奇侠传 online》游戏后台优化

腾讯互娱工程师-王杰

一、服务器 CPU 性能优化

1.1 寻路算法 JPS 优化

MMORPG 游戏中服务器中需要对 NPC 寻路，然而 A*算法及其各种优化并不让人满意，因此寻路算法也成为瓶颈之一。

因此，本文介绍 JPS 的效率、多线程、内存、路径优化算法。为了测试搜索算法的优化性能，实验中设置游戏场景使得起点和终点差距 200 个格子，需要寻路 104 次。结果发现，A*寻路总时间约 $2.6074x10^{11}$ 纳秒（一秒为 10^9 纳秒）；基础版 JPS 寻路总时间 $1.7037x10^{10}$ 纳秒；利用位运算优化的 JPS（下文称 JPS-Bit）寻路总时间 $3.2364x10^9$ 纳秒；利用位运算和剪枝优化的 JPS（下文称 JPS-BitPrune）寻路总时间 $2.3703x10^9$ 纳秒；利用位运算和预处理的 JPS（下文称 JPS-BitPre）寻路总时间 $2.0043x10^9$ 纳秒；利用位运算、剪枝和预处理三个优化的 JPS（下文称 JPS-BitPrunePre）寻路总时间 $9.5434x10^8$ 纳秒。

上述结果表明，寻路 200 个格子的路径，JPS 的五个版本，平均消耗时间分别为 1.7 毫秒、0.32 毫秒、0.23 毫秒、0.02 毫秒、0.095 毫秒，寻路速度分别为 A*算法的 15 倍、81 倍、110 倍、130 倍、273 倍，大幅度超越 A*算法，标志着寻路已经不会成为性能的瓶颈。

事实上，在 2012 到 2014 年举办的三届（目前为止只有三届）基于 Grid 网格寻路的比赛 GPPC（The Grid-Based Path Planning Competition）中，JPS 已经被证明是基于无权重格子，在没有预处理的情况下寻路最快的算法。

1.1.1 JPS 算法介绍

JPS 又名跳点搜索算法（Jump Point Search），是由澳大利亚两位教授于 2011 年提出的基于 Grid 格子的寻路算法。A*算法整体流程如表 1.1.1.1.1 所示，JPS 算法在保留 A*算法的框架的同时，进一步优化了 A*算法寻找后继节点的操作。为了说明 JPS 在 A*基础上的具体优化策略，我们在图 1.1.1.1.1 中给出 A*和 JPS 的算法流程图对比。由图 1.1.1.1.1 看出，JPS 与 A*算法主要区别在后继节点拓展策略上，不同于 A*算法中直接获取当前节点所有非关闭的可达邻居节点来进行拓展的策略，JPS 根据当前结点 current 的方向、并基于跳点的策略来扩展后继节点，遵循“两个定义、三个规则”（见表 1.1.1.1.2，两个定义确定强迫邻居、跳点，三个规则确定节点的拓展原则），具体流程如下：

一，若 current 当前方向是直线方向：
（1）如果 current 左后方不可走且左方可走（即左方是强迫邻居），则沿 current 左前方和左方寻找不在 closedset 的跳点；
（2）如果 current 当前方向可走，则沿 current 当前方向寻找不在 closed 集合的跳点；
（3）如果 current 右后方不可走且右方可走（右方是强迫邻居），则沿 current 右前方和右方寻找不在 closedset 的跳点；

二，若 current 当前方向为对角线方向：
（1）如果 current 当前方向的水平分量可走（例如 current 当前为东北方向，则水平分量为东），则沿 current 当前方向的水平分量寻找不在 closedset 的跳点；
（2）如果 current 当前方向可走，则沿 current 当前方向寻找不在 closedset 的跳点；
（3）如果 current 当前方向的垂直分量可走（例如 current 当前为东北方向，则垂直分量为北），则沿 current 当前方向的垂直分量寻找不在 closedset 的跳点。

JPS 寻找跳点的过程有三种优化：一，位运算；二；预处理；三；剪枝中间跳点。

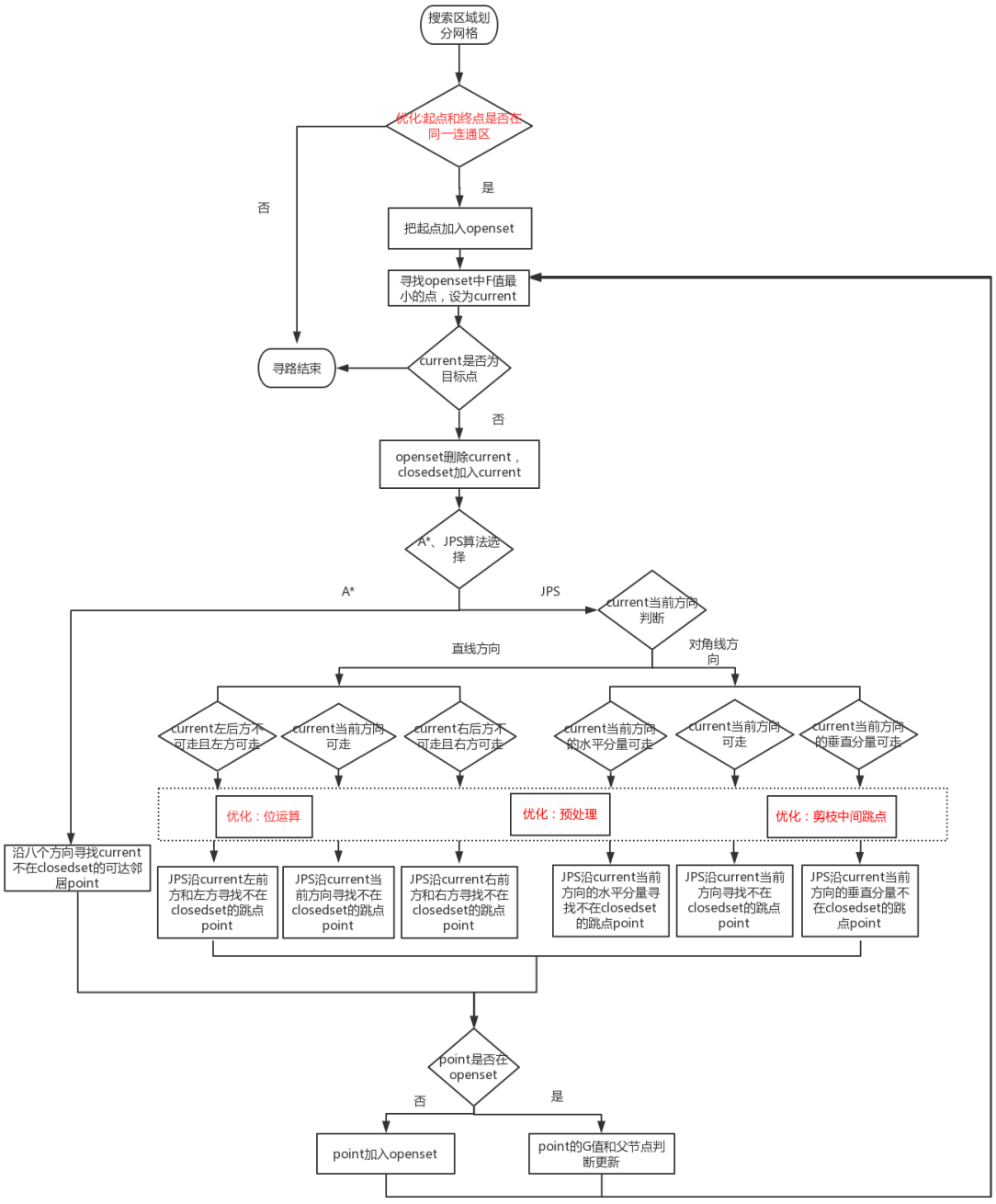


图 1.1.1.1.1A*和 JPS 的算法流程图对比

表 1.1.1.1.1 A*算法流程

Step 1. 将起始点 start 加入开启集合 openset **Step 2.** 重复以下工作： 当 openset 为空，则结束程序，此时没有路径。 寻找 openset 中 F 值最小的节点，设为当前点 current 从 openset 中移出当前点 current 关闭集合 closedset 中加入当前点 current 若 current 为目标点 goal，则结束程序，此时有路径生成，此时由 goal 节点开始逐级追溯路径上每一个节点 x 的上一级父节点 parent(x)，直至回溯到开始节点 start，此时回溯的各节点即为路径。 对 current 的八个方向的每一个相邻点 neighbor 如果 neighbor 不可通过或者已经在 closedset 中，略过。 如果 neighbor 不在 openset 中，加入 openset 中 如果 neighbor 在 openset 中，G 值判定，若此路径 G 值比之前路径小，则 neighbor 的父节点为 current，同时更新 G 与 F 值。反之，则保持原来的节点关系与 G、F 值。G 值表示从起点到当前点路径耗费；H 值表示不考虑不可通过区域，当前点到终点的理论路径耗费；F=G+H。
术语参考： current 当前节点 openset 开启节点集合，集合内节点有待进一步探索拓展 closedset 关闭节点结合，集合内节点后续不再进行拓展 neighbor 邻居，与当前节点相邻的节点 parent(x) 节点 x 的父节点，即算法寻得的路径中节点 parent(x)的下一节点为 x

表 1.1.1.1.2 JPS 算法的“两个定义、三个规则”

定义一，**强迫邻居**（forced neighbour）：如果点 n 是 x 的邻居，并且点 n 的邻居有阻挡（不可行走的格子），并且从 parent(x)、x、n 的路径长度比其他任何从 parent(x)到 n 且不经过 x 的路径短，其中 parent(x)为路径中 x 的前一个点，则 n 为 x 的强迫邻居，x 为 n 的跳点），例如图 2 中，寻找从 S 到 E 的路径时，K 为 I 的强迫邻居（I 为 K 的跳点）。这里不认为从 H 到 K 能走，因为对角线有阻挡（这点和论文不一致，但和代码一致，因为如果 H 到 K 能直接到达，会走进 H 右边的阻挡区，大部分的 JPS 开源代码根据论文都认为 H 到 K 能走，所以存在穿越阻挡的情况），如果需要 H 到 K 可走，则 K 为 H 的强迫邻居（H 为 K 的跳点）。 **定义二**，**跳点**（jump point）：（1）如果点 y 是起点或目标点，则 y 是跳点，例如图 2 中，S 是起点也是跳点，E 是目标点也是跳点；（2）如果 y 有邻居且是强迫邻居则 y 是跳点， 例如 I 是跳点，请注意此类跳点和强迫邻居是伴生关系，从上文强迫邻居的定义来看 n 是强迫

邻居，x 是跳点，二者的关系是伴生的，例如图 2 中 K 的邻居只有 I 是跳点，M 虽然也是 K 的邻居，但 M 不是跳点，因为 K 不是 M 的强迫邻居；（3）如果 parent(y)到 y 是对角线移动，并且 y 经过水平或垂直方向移动可以到达跳点，则 y 是跳点，例如图 2 中 G 是跳点，因为 parent(G)为 S，S 到 G 为对角线移动，从 G 到跳点 I 为垂直方向移动，I 是跳点，所以 G 也是跳点。

规则一，JPS 搜索跳点的过程中，如果直线方向（为了和对角线区分，直线方向代表水平方向、垂直方向，下文所说的直线均为水平方向和垂直方向）、对角线方向都可以移动，则首先在直线方向搜索跳点，再在对角线方向搜索跳点。

规则二，（1）如果从 parent(x)到 x 是直线移动，n 是 x 的邻居，若有从 parent(x)到 n 的路径不经过 x 且路径长度小于或等于从 parent(x)经过 x 到 n 的路径，则走到 x 后下一个点不会走到 n；（2）如果从 parent(x)到 x 是对角线移动，n 是 x 的邻居，若有从 parent(x)到 n 的路径不经过 x 且路径长度小于从 parent(x)经过 x 到 n 的路径，则走到 x 后下一个点不会走到 n（相关证明见论文）。

规则三，只有跳点才会加入 openset，因为跳点会改变行走方向，而非跳点不会改变行走方向，最后寻找出来的路径点也只会是跳点集合的子集。

1.1.1.2 JPS 算法举例

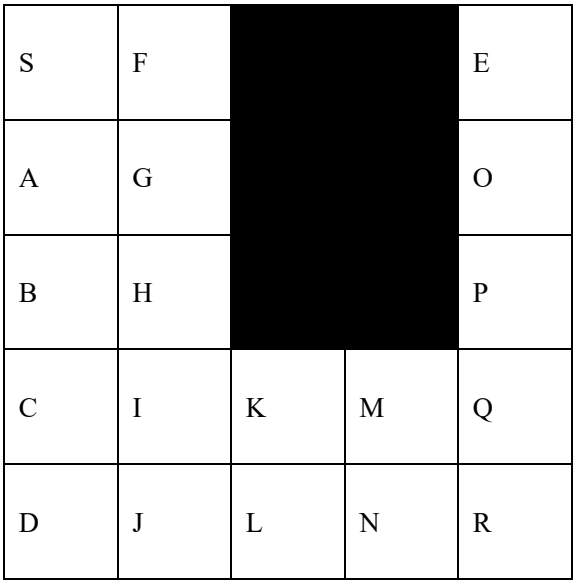


图 1.1.1.2.1 寻路问题示例场景（5*5 的网格）

下面举例说明 JPS 具体的寻路流程。问题示例如图 1.1.1.2.1 所示，5*5 的网格，黑色代表阻挡区，S 为起点，E 为终点。JPS 要寻找从 S 到 E 的最短路径，首先初始化将 S 加入 openset。从 openset 取出 F 值最小的点 S，并从 openset 删除，加入 closedset，S 的当前方向为空，则沿八个方向寻找跳点，在该图中只

有下、右、右下三个方向可走，但向下遇到边界，向右遇到阻挡，因此都没有找到跳点，然后沿右下方向寻找跳点，在 G 点，根据上文定义二的第（3）条，parent(G)为 S，praent(G)到 S 为对角线移动，并且 G 经过垂直方向移动（向下移动）可以到达跳点 I，因此 G 为跳点 ，将 G 加入 openset。从 openset 取出 F 值最小的点 G，并从 openset 删除，加入 closedset，因为 G 当前方向为对角线方向（从 S 到 G 的方向），因此在右、下、右下三个方向寻找跳点，在该图中只有向下可走，因此向下寻找跳点，根据上文定义二的第（2）条找到跳点 I，将 I 加入 openset。从 openset 取出 F 值最小的点 I，并从 openset 删除，加入 closedset，因为 I 的当前方向为直线方向（从 G 到 I 的方向），在 I 点时 I 的左后方不可走且左方可走，因此沿下、左、左下寻找跳点，但向下、左下都遇到边界，只有向左寻找到跳点 Q（根据上文定义二的第（2）条)），因此将 Q 加入 openset。从 openset 取出 F 值最小的点 Q，并从 openset 删除，加入 closedset，因为 Q 的当前方向为直线方向，Q 的左后方不可走且左方可走，因此沿右、左、左上寻找跳点，但向右、左上都遇到边界，只有向左寻找到跳点 E（根据上文定义二的第（1）条)），因此将 E 加入 openset。从 openset 取出 F 值最小的点 E，因为 E 是目标点，因此寻路结束，路径是 S、G、I、Q、E。

注意，本文不考虑从 H 能走到 K 的情况，因为对角线有阻挡（这点和论文不一致，但和代码一致，因为如果 H 到 K 能直接到达，会走进 H 右边的阻挡区，大部分的 JPS 开源代码根据论文都认为 H 到 K 能直接到达，所以存在穿越阻挡的情况)，如果需要 H 到 K 能走，则路径是 S、G、H、K、M、P、E，修改跳点的计算方法即可。

上述的 JPS 寻路效率是明显快于 A*的，原因在于：在从 S 到 A 沿垂直方向寻路时，在 A 点，如果是 A*算法，会将 F、G、B、H 都加入 openset，但是在 JPS 中这四个点都不会加入 openset。对 F、G、H 三点而言，因为从 S、A、F 的路径长度比 S、F 长，所以从 S 到 F 的最短路径不是 S、A、F 路径，同理 S、A、G 也不是最短路径，根据上文规则二的第（1）条，走到 A 后不会走到 F、G，所以 F、G 不会加入 openset，虽然 S、A、H 是 S 到 H 的最短路径，但因为存在 S、G、H 的最短路径且不经过 A，据上文规则二的第（1）条，从 S 走到 A 后，下一个走的点不会是 H，因此 H 也不会加入 openset；对 B 点而言，根据上文规则三，B 不是跳点，也不会加入 openset，直接走到 C 即可。

表 1.1.1.2.1 所示为 A*和 JPS 在寻路消耗中的对比，D. Age: Origins、D. Age 2、StarCraft 为三个游戏龙腾世纪：起源、、龙腾世纪 2、星际争霸的场景图集合，M.Time 表示操作 openset 和 closedset 的时间，G.Time 表示搜索后继节点的时间。可见 A*大约有 58%的时间在操作 openset 和 closedset，42%时间在搜索后继节点；而 JPS 大约 14%时间在操作 openset 和 closedset，86%时间在搜索后继节点。避免在 openset 中加入太多点，从而避免过多的维护最小堆是 JPS 比 A*快的原因（(最小堆插入新元素时间复杂度 log(n)，删除最小元素后调整堆，时间复杂度也为 log(n))），实际上在从 S 到 E 的寻路过程中，进入 openset 的只有 S、G、I、Q、E

表 1.1.1.2.1 A*和 JPS 的寻路消耗对比

	A*		JPS	
	M.Time	G.Time	M.Time	G.Time
D. Age: Origins	58%	42%	14%	86%
D. Age 2	58%	42%	14%	86%
StarCraft	61%	39%	11%	89%

1.1.2 JPS 五个优化算法

1.1.2.1 JPS 优化之一 JPS-Bit：位运算优化

利用位运算优化的 JPS-Bit 的关键优化思路在于利用位运算来优化 JPS 中节点拓展的效率。下面以图 1.1.2.1.1 中的场景示例说明如何将位运算融合于 JPS 算法中，其中黑色部分为阻挡，假设当前位置为 I（标蓝位置），当前方向为右，位运算中使用 1 代表不可走，0 代表可走，则 I 当前行 B 的八位可以用八个 bit：00000100 表示，I 上一行 B-的八位可以用八个 bit：00000000 表示，I 的下一行 B+的八位可以用八个 bit：00110000 表示。在当前行寻找阻挡的位置可以用 CPU 的指令__builtin_clz(B)（返回前导 0 的个数），即当前阻挡在第 5 个位置（从 0 开始）。寻找当前行的跳点可以用__builtin_clz(((B->>1) && !B-) ||((B+>>1) && !B+)) 寻找，例如本例中(B+>>1) && !B+为：(00110000 >> 1) && 11001111，即 00001000，而(B->>1) &&!B 为 00000000，所以__builtin_clz(((B->>1) && !B-) ||((B+>>1) && !B+)) 为__builtin_clz(00001000)为 4，所以跳点为第 4 个位置 M（从 0 开始）。注意论文中使用_builtin_ffs(((B-<<1) && !B-) ||((B+<<1) && !B+))，__builtin_ffs(x)

返回 x 的最后一位 1 是从后向前第几位，比如 7368（1110011001000）返回 4，因为论文对格子的 bit 编码采用小端模式，而这里对格子的 bit 编码采用大端模式。

由于 JPS-Bit 使用运算效率更高的位运算和 CPU 指令运算来优化原始 JPS 节点扩展过程中的遍历操作，JPS-Bit 的算法效率高于原始的 JPS，实测中 JPS-Bit 的寻路时间比 JPS 缩短 5 倍左右。

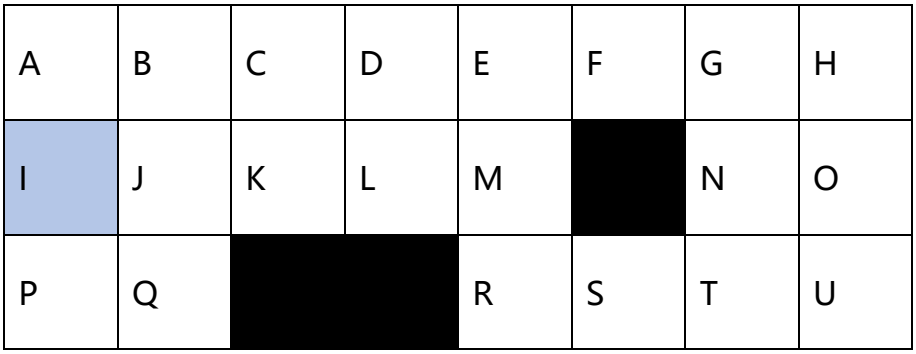


图 1.1.2.1.1 寻路问题示例场景（3*8 的网格）

1.1.2.2 JPS 优化之二 JPS-BitPrune：位运算与剪枝优化

利用位运算和剪枝优化的 JPS-BitPrune 在 JPS-Bit 的基础上进一步进行剪枝优化，剪掉不必要的中间跳点（见表 1.1.1.1.2，定义二第(3)条定义），根据定义二，中间跳点在节点拓展过程中只具有简单的“承接”作用，不具备拓展价值，将中间跳点放入 openset 会增大扩展的次数，因此 JPS-BitPrune 将中间跳点全部删除，将中间跳点后继跳点中的非中间跳点的父跳点改为中间跳点的父跳点，可以有效避免冗余的节点拓展运算。

拐点获取：值得一提的是，JPS-BitPrune 由于删除了中间跳点，因此 JPS-BitPrune 需要在搜索到完整的路径之后以一定的策略在最后寻得的路径中加入中间拐点，使得每两个相邻的路径节点之间都是垂直、水平、对角线方向可达的。对此，JPS-BitPrune 采用的具体方法如下：

假设目前搜索到的路径为 start(jp_1)、jp_2、jp_3...jp_k..end(jp_n)，对每两个相邻的跳点 jp_i、jp_{i+1}，一，如果 jp_i、jp_{i+1} 的 x 坐标或者 y 坐标相等，说明这两个跳点在同一个水平方向或垂直方向，可以直线到达，无需在这两个跳点之间加入拐点；二，如果 jp_i、jp_{i+1} 的 x 坐标和 y 坐标都不相等，（1）如果 x 坐标的差 dx（即 jp_i

的 x 坐标减去 jp_{i+1} 的 x 坐标）和 y 坐标的差 dy 的绝对值相等，说明这两个跳点在对角线方向，也可以直线到达，无需在这两个跳点之间加入拐点；（2）如果 x 坐标的差 dx 和 y 坐标的差 dy 的绝对值不相等，说明这两个跳点不在对角线方向，并且有可能不能直线到达（因为跳点附近有阻挡），此时 jp_i、jp_{i+1} 之间只需要加入一个从 jp_i 出发离 jp_{i+1} 最近的对角线上的点即可（jp_i、jp_{i+1} 不能水平、垂直、对角线到达，说明 jp_i、jp_{i+1} 之间一定存在被剪枝的中间跳点，只需要补上离 jp_{i+1} 最近的一个中间跳点充当拐点即可，该拐点即为 jp_i 沿对角线方向走 min(dx,dy)步到达的点）。

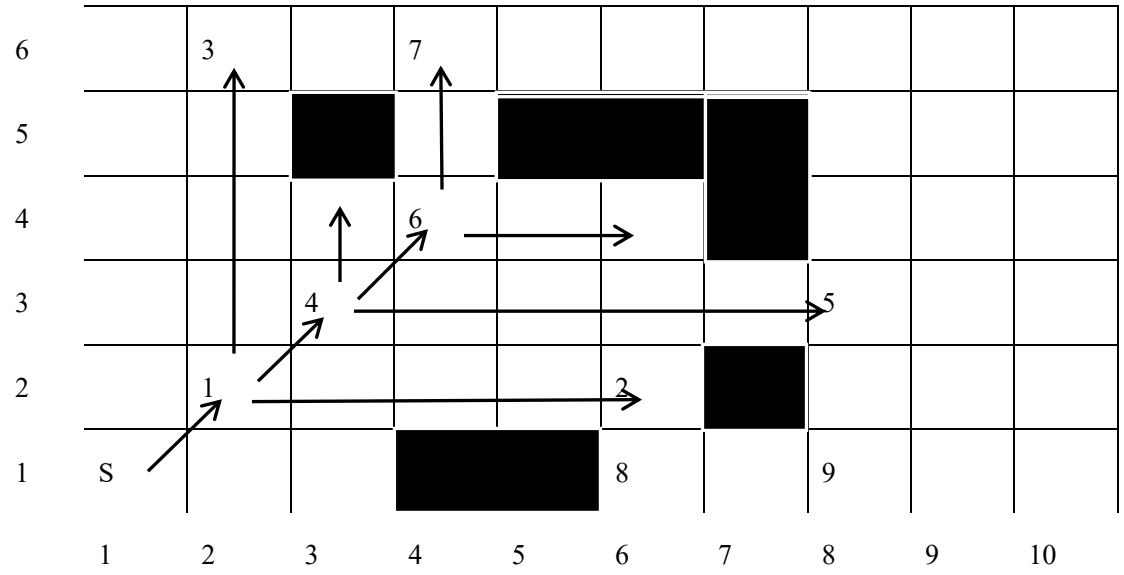


图 1.1.2.2.1 JPS-BitPrune 的剪枝优化示例

下面以图 1.1.2.2.1 的问题场景示例 JPS-BitPrune 如何在剪枝的同时进行寻路。起点为 S（坐标为(1,1)，即 S(1,1)），节点 1、4、6 均为中间跳点：因为节点 2、3 是满足定义二第(2)条的跳点，所以节点 1 是为了到达节点 2、3 的中间跳点，同理节点 4、6 也为中间跳点。在剪枝中间跳点之前，要将中间跳点的后继节点的父节点调整为该中间跳点的父节点。例如图 1.1.2.2.1 中，节点 1 的后继跳点为节点 2、3、4，其中节点 4 也为中间跳点，删掉中间跳点中的节点 1 后，节点 2、3 的父跳点由节点 1 改为节点 S，删除中间跳点中的节点 4 后，节点 4 的后继跳点 5 的父跳点由节点 4 改为节点 S（节点 4 的父跳点为节点 1，但节点 1 已经被删掉，因此回溯到节点 S），删除中间跳点中的节点 6 后，节点 6 的后继跳点 7 的父跳点由节点 6 改为节点 S（节点 6 的父跳点为节点 4，但节点 4 被删，节点 4 的父跳点节点 1 也被删，因此回溯到节点 S）。

上述过程是简化的逻辑描述，实际运行中的做法是从节点 S 寻找跳点，首先找到中间跳点节点 1，然后在水平方向和垂直方向寻找到跳点节点 2、3，将节点 2、3 的父跳点设为节点 S；继续沿对角线方向寻找跳点，走到节点 4 后，沿水平方向和垂直方向寻找到跳点节点 5，将节点 5 的父跳点设为节点 S；继续沿对角线方向寻找跳点，走到节点 6 后，沿水平方向和垂直方向寻找到跳点 7，将跳点 7 的父跳点设为节点 S。因此 JPS-BitPrune 获得路径 S(1,1) 、节点 7(4,6)。

因为路径中 S(1,1)无法垂直、水平、对角线方向走到节点 7(4,6)，需要加入中间拐点，根据上述的拐点添加策略，有 dx 为 3，dy 为 5，需要从 S 沿对角线走 3 步，即节点 6(4,4)可作为中间拐点，因此，在图 1.1.2.2.1 的示例中，JPS-BitPrune 最后构建的完整路径为 S(1,1) 、节点 6(4,4) 、节点 7(4,6)。

1.1.2.2.1 剪枝的优化效率

下面通过对比剪枝前后的 JPS 节点拓展的情况来说明剪枝操作的优化效率：

场景一（无剪枝） 如果不对中间跳点进行剪枝，那么从节点 S 寻路到节点 7 将经历如下过程：
从节点 S 搜索跳点，找到跳点节点 1，openset 此时只有节点 1；
从 openset 取出 F 值最小跳点节点 1，并搜索节点 1 的后继跳点，水平方向和垂直方向找到跳点节点 2、3，对角线方向找到跳点节点 4，此时 openset 有节点 2、3、4；
从 openset 取出 F 值最小跳点节点 4，并搜索节点 4 的后继跳点，水平和垂直方向找到跳点节点 5，对角线方向找到跳点 6，此时 openset 有节点 2、3、5、6；
从 openset 取出 F 值最小跳点节点 6，垂直方向找到跳点 7，此时 openset 有节点 2、3、5、7；
从 openset 取出 F 值最小的跳点节点 7，为目的点，搜索结束，因此完整路径为节点 S(1,1)、节点 1(2,2) 、节点 4(3,3) 、节点 6(4,4) 、节点 7(4,6)。JPS 在到达目的节点 7 之前，需要接连拓展中间跳点 1，4，6。

场景二（剪枝中间跳点） 在剪枝中间跳点之后，从节点 S 寻路到节点 7 的流程得到了明显简化：
从节点 S 寻找跳点，首先找到中间跳点节点 1，然后在水平方向和垂直方向寻找

到跳点节点 2、3，将节点 2、3 的父跳点设为节点 S；继续沿对角线方向寻找跳点，走到节点 4 后，沿水平方向和垂直方向寻找到跳点节点 5，将节点 5 的父跳点设为节点 S；继续沿对角线方向寻找跳点，走到节点 6 后，沿水平方向和垂直方向寻找到跳点 7，将跳点 7 的父跳点设为节点 S；继续沿对角线方向寻找跳点，遇到阻挡，搜索终止，此时 openset 有节点 2、3、5、7；
从 openset 取出 F 值最小的跳点节点 7，为目的点，搜索结束，此时获得的路径为 S(1,1) 、节点 7(4,6)。不同于无剪枝的 JPS 需要拓展中间跳点 1、4、6，在 JPS-BitPrune 中，节点 1、4、6 作为中间跳点均被剪枝，有效避免了冗余的节点拓展，寻路效率得到大大提升。

1.1.2.3 JPS 优化之三 JPS-BitPre：位运算与预处理

本优化中的预处理在一些文章被称为 JPS+。JPS-BitPre 和 JPS-BitPrunePre 都不支持动态阻挡，因为动态阻挡的出现会导致八个方向最多能走的步数发生变化，从而导致预处理的结果不再准确。利用位运算和预处理的 JPS-BitPre 依旧采用 JPS-Bit 中的位运算，而其中的预处理则是对每个点存储八个方向最多能走的步数 step，这个 step 值将影响 JPS 中节点的拓展顺序和拓展“跨度”，加快寻路效率。由于预处理版本的 JPS 需要存储八个方向最多能走多少步，如果地图大小是 N*N，每个方向最多能走的步数用 16 位数表示，则需要存储空间 N*N*8*16bit，如果 N 为 1024，则大概需要存储空间为 16M，存储空间占用较大，使用该版本 JPS 时需要权衡是否以空间换时间，另外预处理的时间对小于 1024*1024 个格子的图可以在 1 秒内处理完，但对于 2048*2048 个格子的图需要一小时左右处理完。

其中，step 值由跳点、阻挡、边界等决定，如果遇到跳点，则 step 为走到跳点的步数；否则 step 为走到阻挡或边界的步数。
例如对图 1.1.2.3.1 中的 N 点，向北最多走到节点 8，即 2 步；
向南最多走到节点 4，即 4 步；
向西最多走到节点 6，即 3 步；
向东最多走到节点 2（节点 2 是满足定义二第（2）条的跳点），即 5 步；
西北最多走到节点 7，即 2 步；
东北最多走到节点 1（节点为 1 是上文定义二第（3）条定义的跳点），即 1 步；
西南最多走到节点 5，即 3 步；
东南最多走到节点 3（节点 3 是上文定义二第（3）条定义的跳点），即 3 步。

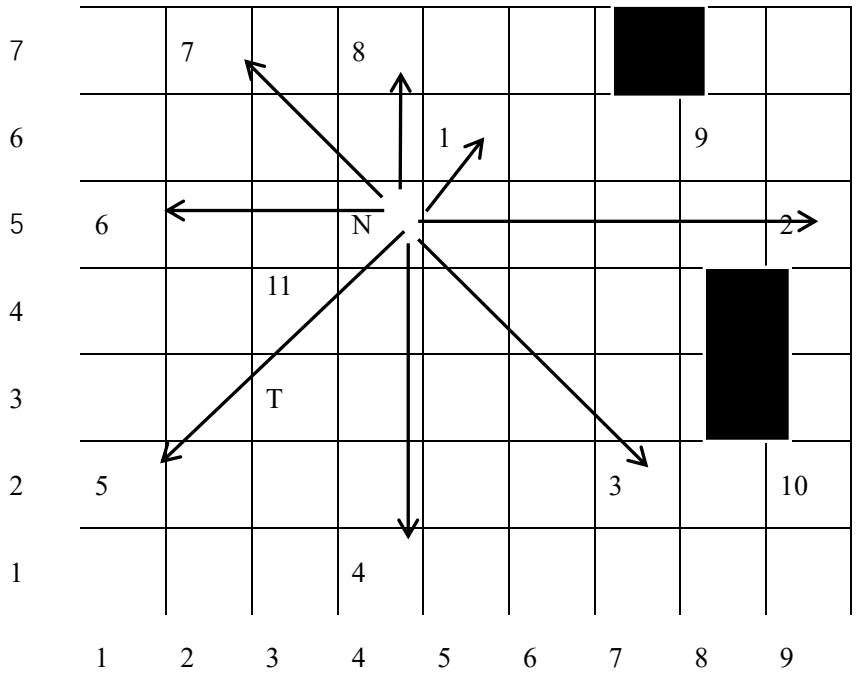


图 1.1.2.3.1　JPS-BitPre 寻路的场景示例

以图 1.1.2.3.1 中的场景为例，要寻找从节点 N 到节点 T 的路径，**JPS-BitPre** 的寻路流程如下：

从 openset 取出节点 N，　从 N 沿八个方向寻找跳点，根据预处理得到的各方向最远可走的 step 值，可以快速确定八个方向最远能到达的节点{1，2，3，4，5，6，7，8}，如图 1.1.2.3.1 所示，其中，节点 1、2、3 均为满足定义二的跳点，直接加入 openset，对于节点 4、5、6、7、8，首先判断终点 T 位于以 N 为中心的南、西南、西、西北、北的哪部分，因为 T 位于西南方向，只有节点 5 位于西南方向，因此节点 4、6、7、8 直接略过，在从 N 到 5 的方向上，N 可走 3 步，而 N 和 T 的 x 坐标差绝对值 dx 为 1，y 坐标差绝对值 dy 为 2，因此将从节点 N 到节点 5 方向上走 min(dx,dy)步即节点 11，加入 openset；

从 openset 取出 F 值最小节点 11，垂直方向找到跳点 T，加入 openset；三，从openset取出F值最小节点T，为目的点，搜索结束，此时获得的路径为N(4,5)、节点 11(3,4) 、节点 T(3,3)。

为了说明 JPS-BitPre 寻路的准确性与高效率，这里给出原始 **JPS-Bit** 从 N 到 T 的寻路流程作为对比：

从 openset 取出节点 N 后，需要沿八个方向寻找跳点，节点 1、3、11 为上文

定义二第（3）条定义的跳点，加入 openset，节点 2 为上文定义二的第（2）条定义的跳点，加入 openset；
从 openset 取出 F 值最小节点 11，垂直方向找到跳点 T，加入 openset；
从 openset 取出 F 值最小跳点 T，为目的点，搜索结束，此时获得的路径也为 N(4,5)、节点 11(3,4) 、节点 T(3,3)。

对比发现，经过预处理的 JPS-BitPre 和只使用位运算的 JPS-Bit 最后寻得的路径是一样的，然而，由于 JPS-BitPre 无需在每一步节点拓展过程中沿各方向寻找跳点，其可以根据预处理得到的 step 值快速确定 openset 的备选节点，从而大大提升寻路效率。

1.1.2.4 JPS 优化之四：不可达两点提前判断

如图 1.1.2.4.1 所示，起点 S 不可到达终点 E，然而寻路算法仍然会花费时间去寻找 S、E 之间的路径，而且失败情况下寻路花费的时间远大于成功情况下寻路花费的时间，因为失败情况下需要遍历所有的路径，才能确定两点不可达。因此为了避免这种情况，在每次寻路之前，判断起点和终点是否可达：如果起点和终点在同一连通区域，则起点和终点可达，否则不可达。只有起点和终点可达，才需要去寻路。

首先计算 Grid 网格的连通区域，算法如表 1.1.2.4.1 所示，算法只能采用宽度优先搜索，深度优先搜索的递归层次太深，会导致栈溢出。按照表 1.1.2.4.1 的算法，图 1.1.2.4.1 的点 S、1、2 的连通区域编号均为 1，点 3、4、E 的连通区域编号均为 2，S、E 连通区域编号不同，因此 S、E 不在同一连通区域，不需要寻找路径。表 1.1.2.4.1 的算法在程序启动时计算一次即可，算法复杂度为 O(N)，N 为 Grid 网格数目，运行时只需要查询两点是否在同一连通区域，算法复杂度为 O(1)。

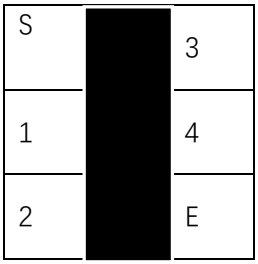


图 1.1.2.4.1 不可达的两点 S、E

表 1.1.2.4.1 计算连通区域

Step1. 当前连通区域编号 num 初始化为 0
Step2. 对 Grid 网格每个点 current 重复以下工作：
一、 num++。
二、 如果 current 是阻挡点，跳过。
三、 如果 current 被访问过，跳过。
四、 current 的连通区域编号记为 num，标记已访问过。
五、 宽度优先搜索和 current 四连通的所有 Grid 网格点，连通区域编号均记为 num，并标记已访问过。

1.1.2.5 JPS 优化之五：空间换时间

openset 采用最小堆实现，最小堆的底层数据结构是一个数组，从最小堆中插入、删除时间复杂度为 O(logn)。除了删除还需要查找操作，每次找到一个跳点，都需要判断在最小堆中是否有，如果有，则判断是否更新 G 值、F 值、父跳点等，如果没有，则加入 openset。在最小堆的中查找操作时间复杂度 O(n)，因此需要优化。closedset 存的是已经从 openset 中弹出的跳点，实际只需要对每个跳点加个标记即可，如果跳点打上标记，则表示是 closedset 中跳点，否则不是。

综合上述需求，针对 1km*1km 的地图，构建 2k*2k 的二维数组 matrix，数组每个元素 pnode 均为一个指针，指针的对象类型包括节点 id、是否扩展过 expanded(即是否在 closedset 中)、G 值、F 值、父跳点指针 parent、在最小堆中的索引 index 等 12 个 byte。如果地图(x,y)处是搜索到的跳点，首先检查在二维数组 matrix 对应的(x,y)处指针 pnode 是否为空，如果为空，表示该跳点之前未搜索过，从内存池 new 出一个跳点，将指针加到最小堆 openset 中，并在执行 shift up、shift down 操作之后，记录在最小堆中的索引 index；如果不为空，则表示该跳点之前搜索过，首先检查 expand 标记，如果为真，则表示在 closedset 中，直接跳过该跳点；否则表示在 openset 中，通过 matrix(x,y)记录的在 openset 中的索引 index 找到对应的指针，检查 matrix(x,y)和 openset(index)的指针是否相等进行二次确认，然后检查判断是否需要更新 G 值、F 值、父跳点等，采用空间换时间的方法可以将 openset 和 closedset 中查找操作降为 O(1)。游戏中有很多场景，无需为每个场景构建一个 matrix，以最大的场景的大小构建一个 matrix 即可。

1.1.3 多线程支持

游戏服务器普遍采用单进程多线程架构，多线程下，不能对 JPS 寻路加锁，否则寻路串行化，失去了多线程的优势，为了支持多线程 JPS 寻路，需要将一些变量声明为线程独有 thread_local，例如上文提到的为了优化 openset 和 closedset 的查找速度，构建的二维跳点指针数组 matrix。该数组必须为线程独有，否则，不同线程在寻路时，都修改 matrix 元素指向的跳点数据，例如 A 线程在扩展完跳点后，将 expanded 标记为真，B 线程再试图扩展该跳点时，发现已经扩展过，就直接跳过，导致寻路错误。

1.1.4 JPS 内存优化算法

1.1.4.1 分层

JPS 的地图格子粒度如果采用 0.5m*0.5m，每个格子占 1bit，则 1km*1km 的地图占用内存 2k*2k/8 个 byte，即 0.5M；为了向上、向下也能通过取 32 位数获得向上、向下的 32 个格子阻挡信息，需要存将地图旋转 90 度后的阻挡信息；

上文 JPS 优化之四：不可达两点提前判断，需要存连通信息，假设连通区数目最多 15 个，则需内存 2k*2k/2 个 byte，即 2m，则内存为：原地图阻挡信息 0.5m、旋转地图阻挡信息 0.5m、连通区信息 2m，即 3m。另外，上文提到用空间换时间的方法，为了优化 openset 和 closedset 的查找速度，构建二维跳点指针数组 matrix。1km*1km 的地图，格子粒度为 0.5m*0.5m，构建出的 matrix 指针数组大小为 2k*2k*4byte 即为 8m，为了支持多线程，该 matrix 数组必须为 thread_local，即线程独有，16 个线程共需内存 16*8m 即为 128m，内存空间太大，因此需要优化这部分内存。

首先将 2k*2k 分成 100*100 的块，即 20*20 个块，20*20 个块为第一层数组 firLayerMatrix，100*100 为第二层数组 secLayerMatrix，firLayerMatrix 的 400 个元素为 400 个指针，每个指针初始化为空，当遍历到的跳点属于 firLayerMatrix 中(x,y)的块时，则从内存池 new 出 100*100*4byte 的 secLayerMatrix，secLayerMatrix 每个元素也是一个指针，指向一个从内存池 new 出来的跳点。

例如，搜索 2k*2k 的地图时，在(231,671)位置找到一个跳点，首先检查 firLayerMatrix 的(2,6)位置指针是否为空，如果为空，则 new 出 100*100*4byte

的 secLayerMatrix，继续在 secLayerMatrix 查找(31,71)位置检查跳点的指针是否为空，如果为空，则从内存池 new 出来跳点，加入 openset，否则检查跳点的 expanded 标记，如果标记为真，表示在 closedset 中，直接跳过该点，否则表示在 openset 中，判断是否更新 G 值、F 值、父节点等。因为游戏中 NPC 寻路均为短距离寻路，JPS 寻路区域最大为 80*80，一个 secLayerMatrix 是 100*100，因此只需要一个 secLayerMatrix，则两层 matrix 大小为：20*20*4byte+100*100*4byte 即为 0.04m。

所以 16 个线程下，总内存为：原地图阻挡信息 0.5m、旋转地图阻挡信息 0.5m、连通区信息 2m、两层 matrix0.04m*16，共 3.64M，游戏中场景最多不到 20 个，所有场景 JPS 总内存为 72.8M。

1.1.4.2 内存池

在 JPS 搜索过程中，每次将一个跳点加入 openset，都需要 new 出对应的节点对象，节点对象中存节点 id、父节点、寻路消耗等共 12 个 byte，为了减少内存碎片，以及频繁 new 的时间消耗，需要自行管理内存池，每次 new 节点对象时，均从内存池中申请，内存池详解请见下文服务器内存优化，为了防止内存池增长过大，需要限制搜索步数。

本文的内存池共有两个：
一，跳点的内存池，初始大小为 800 个跳点，当 new 的跳点数目超出 800 个，即停止寻路，因为服务器用 JPS 进行 NPC 的寻路，NPC 不会进行长距离寻路，假设 NPC 寻路上限距离是 20m，则寻路区域面积是 40m*40m，格子数 80*80 即 6400，经统计跳点数目占所有格子数目的比例不到 1/10， 即跳点数目少于 640，因此 800 个跳点足够使用，800 个跳点共占内存 800byte*12，即为 9.6k，忽略不计；
二，secLayerMatrix 指向的 100*100*4byte 的内存池，因为每次寻路都需要至少一个 secLayerMatrix，如果每次寻路都重新申请，寻路完后再释放，会造成开销，因此 secLayerMatrix 指向的 100*100*4byte 的空间也在内存池中，申请时，从内存池拿出，释放时，放回内存池即可，secLayerMatrix 内存池占内存 0.04m。

1.1.5 路径优化

如图 1.1.4.1 所示，绿色格子为起点，红色格子为终点，灰色格子为跳点，蓝线为 JPS 搜出来的路径，灰色虚线为搜索过程。可以看出，从绿色格子到红色格子可以直线到达，而 JPS 搜索出来的路径却需要转折一次，在游戏表现上，会显得比较奇怪。因此在 JPS 搜索出来路径后，需要对路径进行后处理。

比如 JPS 搜出来的路径有 A、B、C、D、E、F、G、H 八个点，走到 A 时，需要采样检查 A、C 是否直线可达，如果 A、C 直线可达，再检查 A、D 是否直线可达，如果 A、D 直线可达，继续检查 A、E，如果 A、E 直线不可达，则路径优化为 A、D、E、F、G、H，走到 D 时，再检查 D、F 是否直线可达，如果 D、F 直线可达，继续检查 D、G，如果 D、G 直线不可达，则路径优化为 A、D、F、G、H。依此类推，直到走到 H。因为采样检查的速度很快，大约占 JPS 寻路时间的 1/5，而且只有当走到一个路点后，才采样检查该路点之后的路点是否可以合并，将采样的消耗平摊在行走的过程中，因此采样的消耗可以忽略。

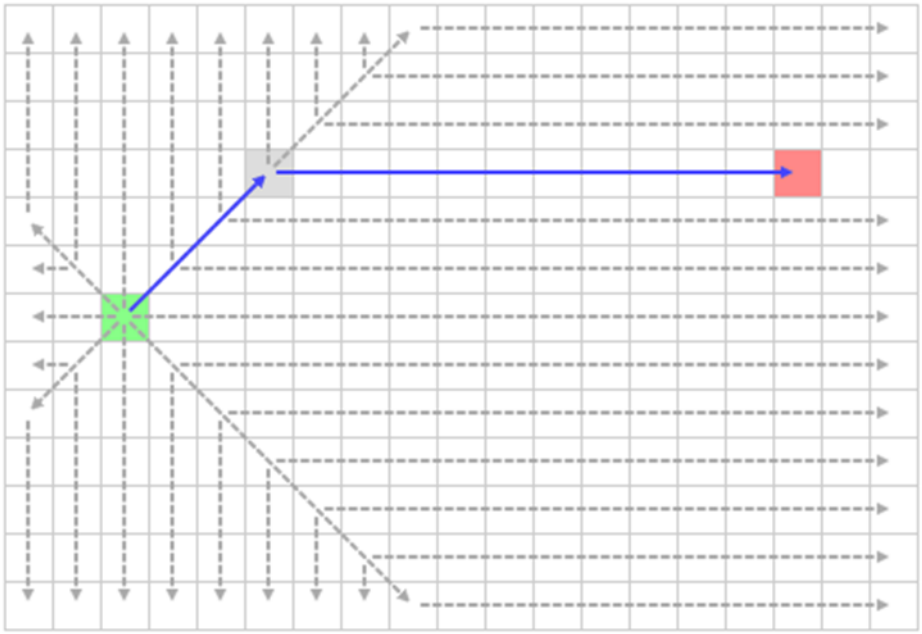


图 1.1.4.1

1.2 视野管理算法的优化

1.2.1 视野管理算法的背景

1.2.1.1 九宫格

游戏中地图用来承载阻挡、静态建筑、NPC（非玩家控制角色：Non-Player-Controlled Character）、WRAP 点等。玩家在地图上移动，其可见的其他玩家即发生变化，如果玩家的每次移动，都更新视野列表，时间成本太高，因此只有当玩家离开某个区域时，才更新视野列表，而在这个区域内的移动，并不更新视野列表。为了划分这个区域，引入九宫格概念，如图 1 所示，九个格子的总面积大于一个手机屏幕，小于两个手机屏幕。大于一个手机屏幕的原因是，可以预先计算当前屏幕外的一些玩家，但又没有必要预先计算太多的屏幕外玩家，因此小于两个手机屏幕，玩家可见的范围为以玩家为中心周围九个格子内的其他玩家。

如果玩家 Me 在格子 5 内移动，则不主动更新视野列表，玩家可见范围为红色和绿色格子内的玩家（如果玩家 Me 视野列表内的玩家 He 从一个格子移动到另一个格子，导致 Me 和 He 不可见，也会导致玩家 Me 的视野列表发生更新，称为被动更新），如果玩家 Me 从格子 5 移动到格子 8 则主动更新视野列表，玩家可见范围为紫色和绿色格子内的玩家。

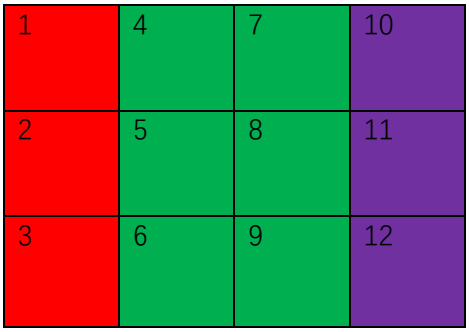


图 1.2.1.1.1 玩家从九宫格的格子 5 移动到格子 8

1.2.1.2 视野管理的必要性

在大型多人在线游戏 MMO（Massively Multiplayer Online）中，多个玩家在同一场景，此时玩家需要能看到其附近的玩家，同时不需要看到与其距离远的玩家。这就是视野管理需要做的事情：为每个玩家维护一个视野列表，管理每个玩家可见视野内的其他玩家。

MMO 游戏中，视野对服务器造成的压力主要来源于两点：一，玩家频繁移动造成视野列表的频繁更新的压力；二，广播视野列表的带宽压力。因为视野列表中

的玩家频繁变化，有的玩家离开当前玩家的视野，有的玩家新进入当前玩家的视野，因此当前玩家的视野列表需要进行频繁的增、删、查操作，因此增、删、查操作的时间复杂度要尽可能的低，从而缓解视野列表频繁更新的压力。

如果当前视野列表中有 100 个玩家，每个玩家都移动了一段距离，为了让其他玩家看到自己的移动，每个玩家都需要被通知其他 99 个玩家的移动，这就需要广播 100*99 个数据包，随着地图中玩家数目增加，造成广播量急剧增加，对带宽造成极大压力，因此玩家的视野列表需要有规模限制，从而缓解带宽压力。

本文提出一种利用全局内存池、双向链表、位标记进行视野管理的算法，可以将每次增、删、查视野列表的复杂度降为 O(1)。

1.2.2 全局内存池

全局内存池中存放的元素如图 1.2.1.1 所示。ViewLinkNodePair 中有两个 ViewLinkNode 类型元素，ViewLinkNode 定义如图 1.2.1.2 所示，记录 ViewLinkNodePair 指针类型的 m_Parents，以及 Obj_User 指针类型的 m_pUser，m_pUser 即为指向玩家的指针。ViewLinkNodePair 两个 ViewLinkNode 元素各自包含一个 m_pUser 指针，这两个 m_pUser 指针存放相互可见的两个玩家。采用内存池好处是避免内存碎片、以及频繁分配的消耗。

```
class ViewLinkNodePair: public PoolBase
{
public:
    ViewLinkNodePair()
    {
        m_LinkNodeA.GetData().m_Parents = this;
        m_LinkNodeB.GetData().m_Parents = this;
    }
    void CleanUp()
    {
        m_LinkNodeA.DisConnect();
        m_LinkNodeA.GetData().CleanUp();
        m_LinkNodeB.DisConnect();
        m_LinkNodeB.GetData().CleanUp();
    }
    CChainItem<ViewLinkNode> m_LinkNodeA;
    CChainItem<ViewLinkNode> m_LinkNodeB;
    virtual const char *GetClassName() const
    {
        return "ViewLinkNodePair";
    }
};
```

图 1.2.1.1

```
class ViewLinkNode
{
public:
    ViewLinkNode() : m_Parents(), m_pUser()
    {
    }

    void CleanUp()
    {
        m_pUser = NULL;
    }
    ViewLinkNodePair *m_Parents;
    Obj_User *m_pUser;
};
```

图 1.2.1.2

1.2.3 双向链表

每个玩家的视野列表是一个双向链表，双向链表中每个元素包含 ViewLinkNodePair 中的两个 ViewLinkNode 之一，ViewLinkNode 中记录 ViewLinkNodePair 和可见的玩家。当玩家 A 和 B 相互可见时，申请 ViewLinkNodePair，A 的 ViewLinkNodeA 记录 ViewLinkNodePair 和 B，B 的 ViewLinkNodeB 记录 ViewLinkNodePair 和 A。当玩家 A 和 B 不相互可见时，在 A 的双向链表中找到记录 B 的 ViewLinkNodeA，然后从 A 的双向链表中删除，并在记录 B 的 ViewLinkNodeA 中找到 ViewLinkNodePair，在 ViewLinkNodePair 中找到记录 A 的 ViewLinkNodeB，在 B 的双向链表中删除。

双向链表每个元素是 CChainItem<ViewLinkNode> 的指针，类模板 CChainItem 定义如图 1.2.2.1 所示，data 存的是 ViewLinkNode 的指针。如图 1.2.2.2 所示，当两个玩家 rObjA、rObjB 相互可见时，首先从全局内存池 g_ViewLinkPool 新建一个对象，对象的指针为 pPair，将 pPair 的 m_LinkNodeA 的 m_pUser 指向 rObjB 的地址，m_LinkNodeB 的 m_pUser 指向 rObjA 的地址，m_Parents 指向 pPair。当两个玩家 rObjA、rObjB 相互不可见时，只需要在 rObjA 的双向链表中找到对应的 CChainItem<ViewLinkNode>的指针，该指针的 m_LinkNodeA 中的 m_pUser 指向 rObjB 的地址，并将对应的双向链表的元素从 rObjA 的双向链表中删除，然后根据 m_LinkNodeA 的 m_Parents 找到对应的 pPair，再从 pPair 中找到 m_LinkNodeB，m_LinkNodeB 的 m_pUser 指向 rObjA 的地址，将 m_LinkNodeB 从 rObjB 的双向链表中删除。

```
template <typename T>
class CChainItem
{
public:
    typedef CChainItem<T> Node;

    Node *next;
    Node *prev;
    T data;
```

图 1.2.2.1

```
ViewLinkNodePair *pPair = g_ViewLinkPool.NewObj();
if (NULL == pPair)
{
    Assert(false);
    return false;
}

pPair->m_LinkNodeA.data.m_pUser = &rObjB;
pPair->m_LinkNodeB.data.m_pUser = &rObjA;

rListA.m_Header.Insert(&pPair->m_LinkNodeA);
rListB.m_Header.Insert(&pPair->m_LinkNodeB);
```

图 1.2.2.2

1.2.4 位标记

游戏中需要频繁的判断两个玩家是否相互可见，然而采用全局内存池+双向链表的数据结构，最快只能采用遍历双向链表的方法，该时间复杂度为 O(n)，因此采用第三个数据结构：位标记辅助完成这项工作。每个场景中的 Obj 数量是有限的，我们游戏每个场景的 Obj 数目最大为 2048，ObjID 编号从 0 到 2047，每个玩家是否可见用一个 bit 表示。所以每个玩家共需要 2047 个 bit 表示是否与其他 2047 个 Obj 可见，即 0.25K，3000 个玩家同时在线，位标记占 0.75M。假设 He 的 ObjID 为 10，判断 Me 是否可见 He，只需要查看 Me 的第 10 个位标记是否为 1 即可。

1.2.5 视野管理流程

如图 1.2.1.1.1 所示，玩家 Me 从格子 5 移动到格子 8，老视野可见的玩家为红色和绿色格子内的玩家，新视野可见的玩家为紫色和绿色格子内的玩家。首先遍历 Me 的双向链表，对所有老视野列表的玩家打上标签 1，然后遍历紫色和绿色格子内的玩家，如果玩家 He 已打标签 1，则将玩家 He 打上标签 2，说明玩家 He 在新视野和老视野都可见；如果玩家 He 没打标签 1，则说明玩家 He 是新进视野的玩家，加入 EnterList；重新遍历 Me 的双向链表，如果玩家 He 仍然是标签 1，说明玩家 He 只在老视野，没在新视野中，加入 LeaveList，同时记录玩家 He 在玩家 Me 视野数组中的索引。

例如 Me 在格子 5 时老视野列表里的玩家为：User1、User2、User3、User4、User5、User6；Me 移动到格子 8 时，紫色和绿色格子内的玩家有 User3、User4、User5、User6、User7、User8。首先对双向链表 User1 到 User6 六个玩家打标签 1；然后对 User3 到 User8 打标签，因为 User3 到 User6 已打标签 1，所以对这 4 个玩家打标签 2，而 User7、User8 没打标签 1，所以这两个玩家加入 EnterList；再遍历双向链表 User1、User2 因为仍然是标签 1，所以将这两个玩家加入 LeaveList，同时记录这两个玩家的 pPair。

对 LeaveList 的两个玩家 User1、User2，因为已知 pPair，所以已知 Me 和 User1、User2 三个玩家的双向链表中 CChainItem<ViewLinkNode>的指针位置，删除即可。

对 EnterList 中的玩家，需要按照优先级高低放到不同的桶里，比如队友的优先级比其他玩家优先级高。然后按照优先级高低的顺序加入视野列表，如果视野列表已满，优先级高的玩家仍然没进入视野列表，需要从视野列表中删除优先级低的玩家，以便腾出空间将优先级高的玩家加入。对 EnterList 的两个玩家 User7、User8，新建 ViewLinkNodePair，并将 ViewLinkNode 插入对应的双向链表即可。

1.3 玩家属性同步的优化

1.3.1 背景介绍

本文适用于所有脏标记遍历功能，提升性能几十倍，本文以游戏中玩家的属性同步作为例子进行介绍。

MMORPG 游戏中玩家有大量属性需要从服务器同步给客户端，例如名字、血量、战力、等级、速度等等。每当某个玩家的某个属性发生变化时，需要将玩家该属性的标志位置脏，在心跳或者其他需要发送的时机，调用同步属性函数，判断置脏的属性并同步给所有客户端。需要注意，不能每次在某个玩家的某个属性发生变化时，均同步给所有客户端，因为开销太大，一般在心跳里（固定时间间隔）同步。为了保证玩家属性在所有客户端及时更新，心跳间隔一般在 500ms 左右，如此频繁的调用以及同步属性函数内大量的脏标记判断导致同步属性函数成为性能瓶颈之一，大约占后台性能消耗的 10%。

假设玩家有 n 个属性，k 个属性置脏，最直观的做法是遍历所有 n 个属性，逐个判断是否置脏，置脏的属性同步给客户端，需要判断 n 次；本文作者之前提过一种优化算法，根据属性置脏的频率，优先判断置脏频率高的属性，当判断出的脏属性数目等于 k，则停止，该优化效率显著优于遍历 n 个属性，但也不是最优的算法。

还有一种优化思路是用 set 记录置脏的属性，set 插入时间复杂度为 O(logn)，置脏 m 个属性，set 插入操作总时间复杂度为 O(log(1*2*..*m))，再考虑某个属性频繁置脏的情况，比如第 m 个属性在同步之前置脏 p 次，则 set 操作时间复杂度为 O(log(1*2..*m*p)，虽然一般情况下 m 比较小，但不排除某些情况 m 很大，就会导致出现性能瓶颈，实测中发现采用 set，o2 编译，如图 1.3.1.1 所示插入 50 个脏属性耗时 165639 微秒，如图 1.3.1.2 所示置脏 50 个标记位耗时 1522 微秒，set 效率低 108 倍，虽然同步的时间间隔内更新的属性数目很少，但某个属性更新的可能非常频繁，所以同步时间间隔内假设插入 50 个脏属性模拟很合理。

显然，如果算法无论属性更新是否频繁，无论属性数目是多是少，都能在 k 的时间判断出哪些属性置脏，则该算法为最优算法。因为置脏的属性非常稀疏，k 往往是个位数，所以最优算法可以将效率提升几十倍。

```
for (int i = 0; i < 10000; i++)
{
    dirtyset.clear();
    for (int j = 0; j < 50; j++)
    {
        dirtyset.insert(j);
    }
}
```

图 1.3.1.1

```
for (int i = 0; i < 10000; i++)
{
    dirtybit = 0;
    for (int j = 0; j < 50; j++)
    {
        dirtybit = dirtybit | (1 << j);
    }
}
```

图 1.3.1.2

1.3.2 优化算法

为引出本文的优化算法，本文首先介绍一个问题：计算一个整数的二进制表示中有多少个 1。基本做法如图 1.3.2.1 所示。假设该整数 n 为 10001100，则需要右移 7 次。但计算机里的数字本来就是用二进制存的，所以计算过程也都是二进制计算。利用一些位运算的特性，可以很容易计算 1 的个数。该例子中 n - 1 为 10001011，n & (n - 1)为 10001000，可以看到低 3 位都变成了 0。多验证几个例子，可以看出结论，要消除整数 n 最低位的 1，可以使用 n = n & (n-1)。从而计算 n 的二进制表示有多少个 1 的算法如图 1.3.2.2 所示。

```
int count = 0;
while (n != 0)
{
    if (n & 1)
    {
        count++;
    }
    n = n >> 1;
}
```

图 1.3.2.1

```
int count = 0;
while (n != 0)
{
    n = n & (n - 1);
    count++;
}
```

图 1.3.2.2

本文介绍一个 gcc 的内置函数，__builtin_ffs(uint32 n)，返回 n 最后一个为 1 的位是从后向前的第几位，例如若 n 为 10001100，则__builtin_ffs(n)为 3，该函数对应的 x86 汇编代码，O0 编译结果如图 1.3.2.3 所示，仅有四条汇编指令。该内置函数 64 位的版本为__builtin_ffsll(uint64 n)。

```
0x00000000004006ee <+14>:    mov    $0xffffffff,%edx
0x00000000004006f3 <+19>:    bsf    %eax,%eax
0x00000000004006f6 <+22>:    cmove  %edx,%eax
0x00000000004006f9 <+25>:    add    $0x1,%eax
```

图 1.3.2.3

接下来正式介绍本文的优化算法，假设有 64 个属性，该 64 个属性用 bit 位表示是否置脏，共需 64bit，将 64bit 转成 uint64 的 n，每次用__builtin_ffsll 找出最低位的 1 的位置，即可找出一个置脏的属性，然后用 n&(n-1)消除最低位的 1，当 n 为 0 时算法终止。假设 64 个属性第 2 个属性和第 9 个属性置脏，则该

64bit 编码为 0...0100000010，首先用__builtin_ffsll 找出低第 2 位的置脏属性，然后 n=n&(n-1)，即 0...0100000000，再用__builtin_ffsll 找出低第 9 位的置脏属性，然后 n=n&(n-1)即为 0，算法终止，可看出从 64 个属性里找出置脏的两个属性，只需要两次操作。

找出置脏属性后，使用 switch 语句而不是 if 语句处理需要同步的属性。因为随着 if 语句数目增多，效率会线性降低。而 switch 语句在 case 多的情况下（gcc 在 O0 编译，case 大于等于 5 个），翻译的汇编代码会使用查找表，在 O(1)时间内找到对应的 case，所以 switch 语句不会在 case 规模很大的情况下降低效率；在 case 很少的情况下（gcc 在 O0 编译，case 小于 5 个），仍然会逐个比较。需要注意，为尽可能提升性能，脏标记数目最好为 64 位整数倍，否则假如脏标记数目为 120 位，取出 64 位后，需要将剩下的 63 位拆分成 32 位、16 位、8 位，分三次取出，再计算脏标记位置，性能会降低。

1.3.3 性能对比

num 类型为 uint64，记录所有脏标记，num 低第 2 位、低第 64 位置脏，即 num 为 100...010。图 1.3.3.1 所示为遍历所有属性，并逐个判断是否置脏，num & 1 不为 0 时，表示第 j 个属性被置脏，图 1.3.3.2 所示为采用本文的优化算法，index 为最低位的脏标记的位置。实测中图 1.3.3.1 耗时 2011 微秒，图 1.3.3.2 耗时 26 微秒，性能提升 77 倍。本文的优化算法不仅适用于属性同步，所有脏标记相关的功能均可采用，例如数据的判断置脏并落地存储。

```
for (int i = 0; i < 10000; i++)
{
    for (int j = 0; j < 64; j++)
    {
        if (num & 1)
        {
        }
        num = num >> 1;
    }
}
```

图 1.3.3.1

```
for (int i = 0; i < 10000; i++)
{
    while (num != 0)
    {
        index = __builtin_ffsll(num) - 1;
        num = num & (num - 1);
    }
}
```

图 1.3.3.2

1.4 跨服战团匹配算法的优化

游戏中存在跨服战等需求，将一些队伍组成固定规模战团，然后战团之间战斗，战团匹配过程应当尽量高效、快速，同时实现战力均衡。为了实现尽可能公平均衡的战团匹配，直观的做法当然是搜索所有队伍组成固定规模的战团，然而其时间复杂度为指数级，以极高的计算代价换取最优匹配显然是不现实的，因此，战团匹配问题也成为制约服务器性能瓶颈问题。

为此，本文利用压桶法实现了快速组成固定规模为 K 的战团，在尽量保证匹配战团的战力均衡前提下，将战团匹配的时间复杂度降为 O(n)。具体上，当有玩家或队伍（实际中，队伍人数为 1 到 5）申请组成战团时，将其加入队列尾部。然后在游戏的心跳中遍历队列，如果当前遍历的玩家或队伍的人数加上桶里的人数小于等于 K，则将当前玩家或队伍加入桶，并从队列中删除；否则新建一个桶，将玩家或队伍加入新桶中。在战团匹配时，找出所有人数为 K 的桶，将桶里所有玩家的战力值的和设为对应桶的权重值，并以此对所有的满的桶进行排序，每次选择战力值最接近的两个战团进行战力均衡的调整，然后传送到战斗服务器进行战斗。

然而，由于压桶法是为了节省时间而选择的贪心算法，可能会遗漏能组成战团的玩家或队伍，因此在压桶法后，本文采用查表法对剩余的玩家和队伍再次尝试组成固定规模的战团。首先在服务器启动时做些预处理，群举所有能组成两个固定规模战团的组合（队伍人数为一的队伍数目，队伍人数为二的队伍数目，...，队伍人数为五的队伍数目），并存在 set 里，该步骤是全局唯一的，并且只需要做一次。然后统计剩下的玩家和队伍，计算人数为一、二、...、五的队伍数目。如果剩下的玩家或队伍队伍人数为一到五的队伍数目均大于 set 中某个元素，则 set 中该元素表示能组成两个固定规模战团的组合，并从剩下的玩家或队伍中删除。此步骤重复进行，直到剩下的玩家或队伍无法组成两个固定规模的战团。

通过上述方法，本文实现尽可能合理的战团组成，在此之后，为了尽可能实现两个战团战力均衡，本文进一步设计战团玩家调整方法。当两个固定规模的战团组成后，通过查询预先生成的表，调整两个战团里的玩家，使得两个战团的战力尽可能相等，增强战斗刺激性。具体上，首先群举所有的能组成两个固定规模战团的可能，并存在 map 里，map 的 key 为字符串，key 唯一标识战团的构成（队伍人数为一的队伍数目，队伍人数为二的队伍数目，...，队伍人数为五的队伍数

目)，map 的 value 为一个 vector 容器，vector 中存储当前 key 拆分成两个固定规模战团的所有可能。当调整两个战团的战力时，通过 map 查询当前两个战团能重组成的所有两个战团的可能，然后遍历所有的可能，找出两个战团战力最接近的组合。

1.5 发包逻辑拆分

在服务器给客户端发包时，需要将数据包打包成流，在 Encode 操作中存在大量 memcpy 操作，而每个 memcpy 时间复杂度为 O(n)，因此打包过程也成为性能瓶颈之一。因此，本文认为有必要将数据包打包并发包的过程从游戏主线程中拆分，另开一个线程处理。在具体的多线程实现过程中，在主线程和发包线程之间需要有通信的数据结构，主线程负责将数据包写进该数据结构，发包线程负责将包从该数据结构中读取、打包、发送，合理的通信数据结构的使用将对多线程效率影响很大。

简单的实现可以采用 c++的 queue 作为两个线程通信的数据结构，然而需要对 queue 加锁，才能保证两个线程读写安全，但锁的开销会降低性能。因此，本文采用基于字节流的无锁循环队列。因为每个数据包大小不一样，小的几个 Byte，大的几百 K，因此不能采用基于对象的无锁循环队列，否则几个 Byte 的数据包也会占据几百 K 的空间。

另外本文采用 Tconnd 作为网络通信组件，Tconnd 发包需要携带 TFRAMEHEAD，该结构体大小为 12K，除了 TFRAMEHEAD_CMD_START 等管理包的该 TFRAMEHEAD 是从 Tconnd 接收然后再自己填充一些字段外，其他数据包的 TFRAMEHEAD 完全是自己拼的，因此知道需要填充哪些字段。 所以除了 TFRAMEHEAD_CMD_START 等管理包的 TFRAMEHEAD 是在主线程收到并拼完，然后 12K 完整的写进无锁循环队列，其他的 TFRAMEHEAD 均是主线程将需要的参数写进无锁循环队列，发包线程再根据这些参数拼 TFRAMEHEAD。这种优化可以大大降低无锁循环队列所占内存。

1.6 玩家 Cache

在当前多进程的服务器的架构下，查看玩家信息亟需优化。当一台服务器（GameServer1)的玩家要查看另一台服务器(GameServer2)的玩家信息时，需要经过多个服务器的多次查询中转。譬如，上述请求的流程可能为

GameServer1 -> WorldServer -> GameServer2 -> WorldServer -> GameServer1，请求消息共需要转发四次。同时，查看离线玩家信息时，通常需要对数据库进行操作，这都导致查看玩家信息需要被优化。

为此，本文通过对玩家信息进行缓存（玩家 cache）实现玩家信息查看优化。具体上，1）同一游戏服务器下的用户不需缓存，譬如，GameServer1 玩家 A 查看 GameServer1 的玩家 B 不通过缓存就可以很容易地获取玩家 B 信息；2）不同游戏服务器下需进行玩家缓存，且缓存按需更新。譬如，GameServer1 玩家 A 查看 GameServer2 的玩家 C，在 GameServer1 的玩家 cache 中查看玩家 C，如果玩家 C 进入缓存时间超过 10 秒，更新缓存中玩家 C 信息，如果玩家 C 不在缓存中，则从数据库加载；3）查看离线玩家同样利用玩家 cache，每逢玩家离线，通知所有 GameServer 更新玩家 cache 中该玩家信息。

因为被查看的玩家往往都是等级、战力等特别高的玩家，这些玩家会被大量玩家查看，所以本文提出的玩家缓存机制往往命中率极高，实测中缓存命中率达到 80%-90%左右。

二、服务器内存优化

2.1 内存统计

在对内存优化之前，需要先确定程序每个模块的内存分配。程序的性能有 perf、gprof 等分析工具，但内存没有较好的分析工具，因此需要自行统计。在 linux 下/proc/self/statm 有当前进程的内存占用情况，共有七项：指标 vsize 虚拟内存页数、resident 物理内存页数、share 共享内存页数、text 代码段内存页数，lib 引用库内存页数、data_stack 数据/堆栈段内存页数、dt 脏页数，七项指标的数字是内存的页数，因此需要乘以 getpagesize()转换为 byte。在每个模块结束后统计 vsize 的增加，即可知该模块占用的内存大小。

在面向对象开发中，内存的消耗由对象的消耗组成，因此需要统计每个类的成员变量的占用内存大小。使用 CLion 或者 visual studio 都可以导出类中定义的所有成员变量，然后在 gdb 使用命令：
p ((unsigned long)(&((ClassName*)0)->MemberName))，即可打印出类 ClassName 的成员变量 MemberName 相对类基地址的偏移，根据偏移从小

到大排序后，变量的顺序即为定义的顺序，根据偏移相减即可得出每个成员变量大小，然后优化占用内存大的成员变量。

2.2 内存泄露

内存泄露是指程序中已动态分配的堆内存由于某种原因程序未释放或无法释放，导致内存一直增长。虽然有 valgrind 等工具可以检查内存泄露，但 valgrind 虚拟出一个 CPU 环境，在该环境上运行，会导致内存增大、效率降低，对于大规模程序，基本无法在 valgrind 上运行。因此需要自行检查内存泄露，glibc 提供的内存管理器的钩子函数可以监控内存的分配、释放。如图 2.2.2、2.2.3 所示，分别为钩子函数的分配内存和释放内存。因为服务器启动时需要预先分配很多内存，比如内存池，这些内存是在服务器停止时才释放，因此为了避免这些内存的干扰，在服务器启动之后才能开始内存泄露的统计。

首先申请固定大小的 vec_stack，记录所有分配的内存，如果有释放，则从 vec_stack 中删除，最后 vec_stack 中的元素即为泄露的内存，vec_stack 必须为固定大小，否则 vector 扩容中会有内存分配，也不可以用 map，map 的红黑树旋转也会有内存分配，会造成干扰；然后通过图 2.2.1 所示的 my_back_hook 记录原有的 malloc、free；并通过图 2.2.2 所示的 my_init_hook 将 malloc、free 换成自定义的钩子函数。

每次分配内存时，都会进入自定义钩子函数 my_malloc_hook 中，如图 2.2.2 所示。在 my_malloc_hook 中首先通过 my_recover_hook 将 malloc 恢复成默认的，否则会造成死递归，然后通过默认的 malloc 分配大小为 size 的空间，为了分线程统计内存泄露，还需要对线程号做判断，在 stTrace.m_pAttr 记录内存分配的地址，m_nSize 记录大小，m_szCallTrace 记录调用栈，如果 vec_stack 已满，需要根据 m_nSize 从大到小排序，如果当前分配内存大于 vec_stack 记录的最小的分配内存，则替换；如果未满，则直接加入 vec_stack，在 my_malloc_hook 结束时，将 malloc 替换成自定义的 malloc。

每次释放内存时，都会进入自定义钩子函数 my_malloc_free 中，如图 2.2.3 所示。在 my_malloc_free 中首先通过 my_recover_hook 将 free 恢复成默认的，否则会造成死递归，对线程号判断，然后在 vec_stack 中删除对应的分配，并将 free 替换成自定义 free。

```
static void my_backup_hook (void)
{
    old_malloc_hook = __malloc_hook;
    old_free_hook = __free_hook;
}

static void my_init_malloc (void)
{
    __malloc_hook = my_malloc_hook;
    __free_hook = my_free_hook;
}

static void my_recover_hook (void)
{
    __malloc_hook = old_malloc_hook;
    __free_hook = old_free_hook;
}
```

图 2.2.1

```
static void *my_malloc_hook (size_t size, const void *caller)
{
    void *result;
    my_recover_hook();
    result = malloc (size);
    if (pthread_equal(gs_MainThreadID, pthread_self()))
    {
        stMallocCallStack stTrace;
        stTrace.m_pAttr = result;
        stTrace.m_nSize = (int)size;
        memset(stTrace.m_szCallTrace, 0, sizeof(stTrace.m_szCallTrace));
        backtrace(stTrace.m_szCallTrace, MAX_MEMLEAK_STACK_LV);
        if (likely(vec_stack.size() >= MAX_MEMLEAK_RECORDE_NUM))
        {
            if (int(size) > s_nMinMallocSize)
            {
                std::sort(vec_stack.begin(), vec_stack.end(),
                    [](const stMallocCallStack& a, const stMallocCallStack& b) -> bool
                {
                    return a.m_nSize > b.m_nSize;
                });
                s_nMinMallocSize = vec_stack[MAX_MEMLEAK_RECORDE_NUM-1].m_nSize;
                if (int(size) > s_nMinMallocSize)
                {
                    vec_stack[MAX_MEMLEAK_RECORDE_NUM-1] = stTrace;
                }
            }
        }
        else
        {
            vec_stack.emplace_back(stTrace);
        }
    }
    my_init_hook();
    return result;
}
```

图 2.2.2

```
static void my_free_hook (void *ptr, const void *caller)
{
    my_recover_hook();
    free (ptr);

    if (pthread_equal(gs_MainThreadID, pthread_self()))
    {
        for( auto it = vec_stack.begin() ; it != vec_stack.end(); it++ )
        {
            if ((*it).m_pAttr == ptr)
            {
                vec_stack.erase(it);
                break;
            }
        }
    }
    my_init_hook();
}
```

图 2.2.3

2.3 内存池

在游戏服务器内存分配过程中，glibc 中的 malloc 采用的是 ptmalloc，ptmalloc 在对小对象进行分配时会产生大量内部碎片，同时也会有外部碎片的产生。因此往往需要自行设计并实现模板化的内存池，采用内存池的好处是，避免内部、外部碎片，对象 New/Delete 均为 O(1)复杂度，同时有利于监控。例如当模板参数为宠物时，首先 new 出 60 个宠物的空间，对这 60 个宠物空间，当 60 个空间被全部使用时，再分配 60 个空间。本文在内存池设计中实现两个数据管理器，分别是空闲数据管理器 freeList 和使用空间管理器 usedList，freeList 采用 queue 实现即可，usedList 采用 vector。

由于这部分较复杂，因此用代码辅以说明。如图 2.3.1，在 Init 函数中传入要创建的对象数目 capacity，pointer 根据 capacity 预先分配空间，用于建立所有对象索引，pointer 主要用来校验。本文按块 Chunk 创建对象，每块 Chunk 有 objectperchunk 个 T 类型对象。chunksize 记录当前总共创建了多少对象。虽然 Init 函数设定了 capacity，但是初始并没有创建 capacity 个 T 对象，而是先创建 objectperchunk 个 T 对象供使用，并记录到 freelist 中。

如图 2.3.2 所示，每次新创建对象时调用 NewObj 函数，如果 freelist 为空，说明所有已创建的对象均被使用，则再调用 NewChunk 创建 objectperchunk 个

T 对象；如果不为空，直接返回 freeList 头部的指针。

如图 2.3.3 所示，每次删除对象时调用 DeleteObj 函数，参数为待删除对象指针，除了必要的校验以外，将对象指针从 usedList 删除，并将 usedList 最后一个对象指针移动到删除位置，记录删除位置 deleteusedlistindex。同时将对象指针所指的对象清空加入 freeList，以便再次使用，好处时内存不回收可以重复使用。

如图 2.3.4 所示，对象的遍历在 usedList 上进行，但在遍历中可能会删除对象，此时 deleteusedlistindex == iterateindex，因为删除对象后，会将 usedList 最后一个对象指针移动到 deleteusedlistindex 位置，因此 iterateindex 不能增加。

```
std::vector<PTR> pointer;           //下标是PoolID, pointer[poolid]指向Object
std::vector<Chunk> chunklist;       //分块列表
std::deque<PTR> freelist;           //空闲列表
std::vector<PTR> usedlist;          //使用列表

unsigned objectperchunk = OBJECT_PER_CHUNK;
unsigned capacity = 0;              //容量
unsigned peakcount = 0;             //使用峰值
unsigned deleteusedlistindex = -1;  //删除位置,用于遍历
unsigned iterateindex = -1;         //当前遍历位置

struct Chunk
{
    std::shared_ptr<T> list;
    Chunk(unsigned count) : list(new T[count], std::default_delete<T[]>()) {}
};

//自分配内存模式调用此Init接口
bool Init(size_t capacity)
{
    if (this->capacity != 0 || capacity == 0)
        return false;

    this->capacity = capacity;
    pointer.resize(capacity);

    NewChunk();
    return true;
}

void NewChunk()
{
    size_t chunksize = chunklist.size() * objectperchunk;
    Chunk c(objectperchunk);
    for (size_t i = 0; i < objectperchunk; ++i)
    {
        T* t = c.list.get() + i;
        unsigned poolid = chunksize + i;
        t->SetPoolID(poolid);
        if (freelist.size() + usedlist.size() < capacity)
        {
            AddToFreeList(t);
            pointer[poolid] = t;
        }
    }
    chunklist.push_back(c);
}
```

图 2.3.1

```
T *NewObj()
{
    if (capacity == 0 && !Init(objectperchunk))
        return nullptr;
    if (usedlist.size() >= capacity)
        return nullptr;
    if (freelist.empty())
        NewChunk();
    auto front = freelist.front();
    freelist.pop_front();
    AddToUsedList(front);
    front->CleanUp();
    return front;
}
```

图 2.3.2

```
bool DeleteObj(T *t)
{
    if (t == nullptr)
        return false;

    auto usedlistindex = t->GetUsedListIndex();
    if (usedlistindex == -1u)
    {
        t->CleanUp();
        std::unique_ptr<char[]> up(new char[10240]);
        GetCurrentStack(up.get(), 10240);
        ERRORLOG << "==========[fatal error] object:" << (unsigned long)t << " double delete==========" << std::endl;
        ERRORLOG << up.get() << std::endl;
        return false;
    }

    size_t index = t->GetPoolID();
    assert(index < capacity);
    T *p = Get(index);
    if (p == nullptr || p != t)
    {
        assert(false);
        return false;
    }
    DeleteFromUsedList(t);
    AddToFreeList(t);
    t->CleanUp();
    return true;
}
void DeleteFromUsedList(T* t)
{
    assert(!usedlist.empty());
    auto lastindex = usedlist.size() - 1;
    deleteusedlistindex = t->GetUsedListIndex();
    if (deleteusedlistindex != lastindex)
    {
        usedlist[deleteusedlistindex] = usedlist[lastindex];
        usedlist[deleteusedlistindex]->SetUsedListIndex(deleteusedlistindex);
    }
    usedlist.pop_back();
}
void AddToFreeList(T* t)
{
    t->SetInPoolFlag(false);
    t->SetUsedListIndex(-1);
    freelist.push_back(t);
}
```

图 2.3.3

```
 * 遍历推荐写法
 * T* p = nullptr;
 * while ((p = NextObj(p)))
 * {
 *     do something
 * }
 */
T *Next(T *p = NULL)
{
    if (p == nullptr)
    {
        iterateindex = -1;
        deleteusedlistindex = -1;
    }

    if (!(deleteusedlistindex == iterateindex))
        ++iterateindex;

    if (iterateindex < usedlist.size())
    {
        deleteusedlistindex = -1;
        return usedlist[iterateindex];
    }

    return nullptr;
}
```

图 2.3.4

2.4 内存分配（ptmalloc vs tcmalloc）

即使使用内存池，程序中仍然需要大量使用 malloc 分配空间。但主流使用的 ptmalloc 存在如下缺点：一，ptmalloc 采用多个线程轮询加锁主分配区和非主分配的方式，造成锁的开销；二，多线程间的空闲内存无法复用，利用线程 A 释放一块内存，由于并不一定会释放给操作系统，线程 B 申请同样大小的内存时，不能复用 A 释放的内存，导致使用内存增加；三，ptmalloc 分配的每块 chunk 需要 8 个字节记录前一个空闲块大小和当前块大小以及一些标记位。

为此，本文提倡使用 tcmalloc 进行内存分配。tcmalloc 对每个线程单独维护 ThreadCache 分配小内存；针对 ptmalloc 多线程内存无法复用的问题，tcmalloc 为进程内的所有线程维护公共的 CentralCache，ThreadCache 会阶段性的回收内存到 CentralCache；针对 ptmalloc 每块 chunk 使用 8 个字节表示其他信息，tcmalloc 对每块 chunk 使用大概百分之一的空间表示其他信息，对小对象分配，空间利用率远高于 ptmalloc。

3 服务器启动时间优化

3.1 读取阻挡文件优化

服务器启动时，需要读大量文件，其中最大的文件就是阻挡文件，每个阻挡文件大约有几 M，采用 c++的 ifstream 可以一次读取一个字节，也可以一次读取整个文件，后者的速度比前者快几十倍以上。如图 3.1.1 所示，MATRIX_LEN 为 4096，阻挡文件为 4096*4096byte，共 16M，图 3.1.1 按 byte 读取阻挡文件需要 2564ms，图 3.1.2 一次性读取阻挡文件需要 50ms。

```
for (int i = 0; i < MATRIX_LEN; i++)
{
    for (int j = 0; j < MATRIX_LEN; j++)
    {
        infile.read((char*)&read_matrix[i][j], 1);
    }
}
```

图 3.1.1

```
infile.read((char*)read_matrix, MATRIX_LEN * MATRIX_LEN);
```

图 3.1.2

《御龙在天移动版》服务器性能优化

腾讯互娱高级工程师-蔡铭福 ,刘林，杨岳军，向熠

一、游戏介绍

《御龙在天移动版》是一款 3D MMORPG 手游，以三国为背景，移植《御龙在天端游》经典玩法，主打手机上的实时万人国战，同时通过社交系统和主播语音来增进玩家间的交互。

游戏特点：合纵连横，多国万人同时参与国战；开放式场景，随时随地遭遇战；国家语音指挥，大量玩家集体行动。

二、性能优化背景

《御龙在天》手游的主打玩法是实时万人国战。

由于国战的参与人数非常多，国战的流程是分布在不同的地图场景中。玩家在不同的场景间进行切换时，需要给玩家带来良好的体验。同时由于涉及多场景，如何做到降低逻辑编码和正式环境部署的复杂度，提升正式环境扩容缩容的灵活性，考虑容灾效果和负载均衡，需要一个设计合理的架构和场景切换流程。

国战参与人数多带来的另一个问题是流量消耗大。由于国战期间，玩家是在地图场景中实时进行移动和战斗，大量释放技能，玩家视野和属性更新频繁，势必会有大量的网络包需要发给手机客户端。手机玩家对流量比较敏感，而玩家手机也参差不齐，大量的网络包也会给手机端带来性能和电量消耗，影响游戏流畅度。因此需要一些方法，对流量进行优化。

MMORPG 游戏通常玩法和系统的数量都非常多，因此代码量也是非常大的。当出现性能问题需要优化时，如何从百万行级别代码的工程里，发现那些性能浪费最严重的代码，进行优化后，如何检验性能优化的效果，需要从工具、原因定位方法、优化策略几个方面入手来解决问题。

基于这些问题背景，本文将从四个方面来介绍御龙在天手游的服务器优化经验：

1.场景切换流程优化；

2.国战架构优化；

3.流量消耗优化；

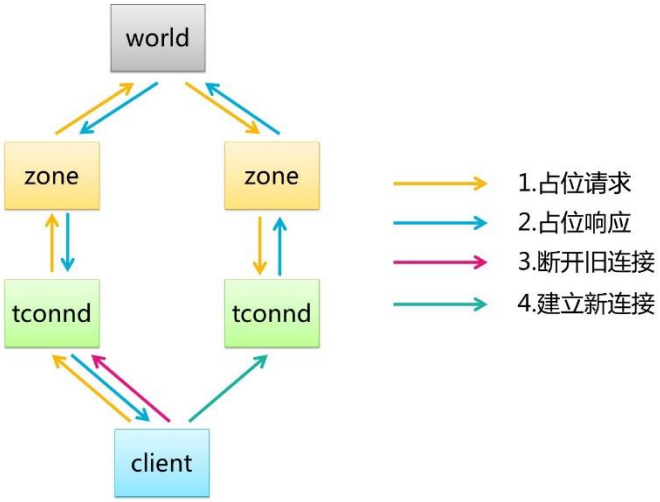
4.代码级性能优化工具、原因定位方法、优化策略。

三、场景切换流程优化

3.1 背景介绍

《御龙在天》手游是一款分场景地图的 MMORPG，不同国家的地图场景，可能会分布在不同的 zone 场景进程上。当玩家从一个场景进入另一个场景时，如果两个场景是分布在不同进程上，这时需要将玩家对象和相关数据、状态信息，从原来所在进程同步到新的进程上，也就是我们说的跳 zone 流程。在这个流程中，由于涉及到玩家所在进程的切换，在网络状况欠佳的情况下，可能会导致重连失败，影响玩家体验。尤其是在移动网络的环境下，受到运营商信号和网络切换的影响，增加了跳 zone 失败的几率。基于以上原因，需要对跳 zone 流程进行优化，以降低跳 zone 失败的几率，提升玩家体验。

3.2 原有方案



步骤如下：

原 zone 向目标 zone 发起占位请求；

目标 zone 向原 zone 回服占位响应；

占位成功后，原 zone 通知客户端断开原有连接；

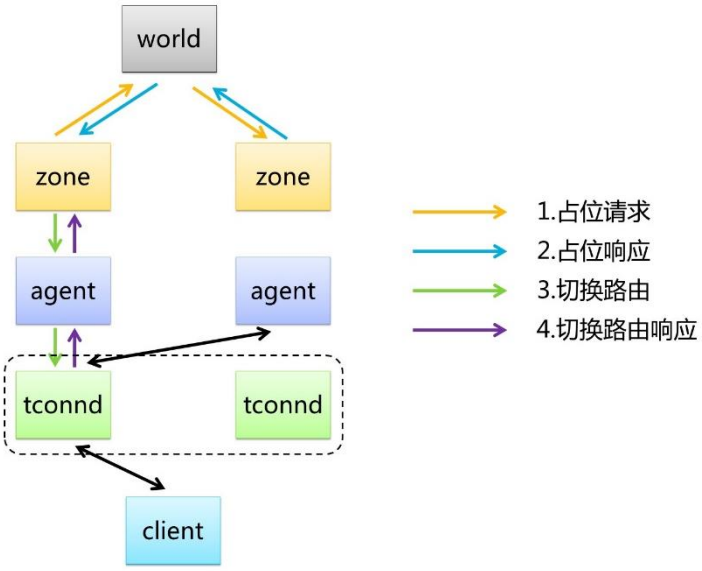
客户端发起新的网络连接到目标 zone。

方案缺点：

需要客户端参与配合流程；

客户端需要重新建立连接。

3.3 改进方案



步骤如下：

原 zone 向目标 zone 发起占位请求；

目标 zone 向原 zone 回服占位响应；

占位成功后，原 zone 通过 agent，切换路由到目标 zone 的 agent；

收到切换路由响应，数据同步到目标 zone。

3.4 改进结果

目前项目使用改进后的方案，有 2 个优点：

不需要客户端参与和配合进行跳 zone 流程，客户端侧完全无感知；

不需要重新建立网络连接；

优化后的跳 zone 失败率，从千分之一下降到了万分之一以下，效果较为明显。

四、国战架构优化

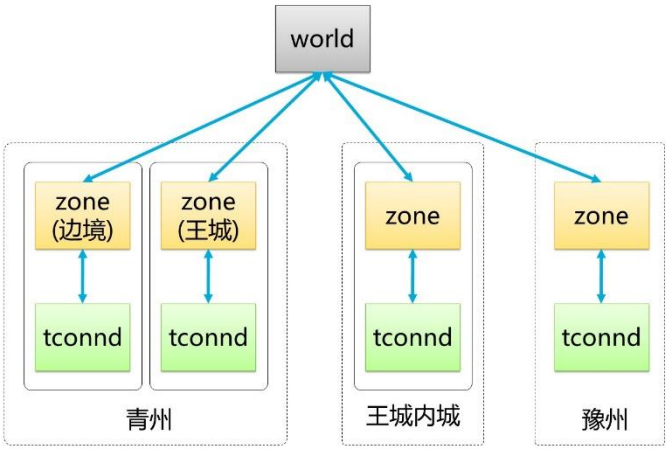
4.1 背景介绍

国战是《御龙在天》手游的核心玩法之一。由各国国王发起国战，进攻方在指定国战时间内，逐一占领敌国的边境、阳平关、王城郊外，最后在王城内城杀死国家神兽，取得国战胜利。每一个地图都有一个要占领的目标点，进攻方通过杀死镇守每个地图的目标 NPC，来占领该地图。若在指定时间内，守方成功守卫本国神兽，则守方获胜。国战的架构设计和优化需要考虑几个问题：

由于国战涉及到多个场景和地图，而不同的场景和地图可能分布在不同的进程上，如何设计架构，以便在逻辑开发和外网部署时，不会依赖于具体的地图，减少编

码复杂度，同时也可以根据运营数据灵活调整进程和机器数量；
由于王城内城的场景地图，在非国战期间是无法进入的，通过什么方式来实现，可以达到既节约机器资源，又能较好达到容灾效果；
国战期间参与玩家人数非常多，大量玩家聚集在同一个地图和场景上，网络流量陡增，如何设计场景进程 zone 和网络连接管理进程 tconnd 的关系，使得在网络层面减少对机器的压力。

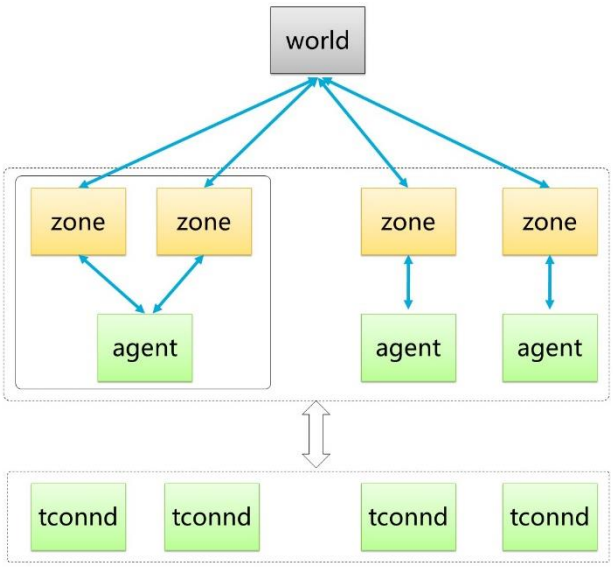
4.2 原有方案



场景进程 zone 与国家地图绑定，通过配置进行关联；
王城内城通过普通地图方式实现，需要配置绑定到某个特定 zone 上；
每个 zone 进程绑定 2 个 tconnd 进程，每个 zone 进程上的所有客户端连接都通过这 2 个 tconnd。

缺点：
场景进程 zone 与国家地图绑定，在国战逻辑的开发时，对不同的逻辑需要做分支判断，是否是在对应的地图上；
王城通过普通地图方式实现，在运营后期单区人数下降的时候，缩减机器也必须保留王城内城的 zone。容灾能力也较弱，如果王城内城的 zone 出故障，就会影响国战的流程，导致国战无法正常进行；
zone 与 tconnd 是绑定的，在国战期间大量玩家聚集到一个 zone 上，会对对应的 tconnd 带来很大压力。

4.3 改进方案



场景进程 zone 与国家地图不绑定，通过 world 进程在启动时进行分配和创建；王城内城通过副本的方式实现，在国战开始时，通过副本创建策略，动态选择 zone 进程来创建副本；

tconnd 使用集群化模式，与 zone 进程不绑定，通过 agent 进程来路由消息；单区内所有 zone 进程的网络连接，根据负载平均分布在所有 tconnd 上。

4.4 改进结果

场景进程 zone 与国家地图不绑定，在国战逻辑开发时，不需要再根据地图来进行逻辑分支的判断。在运营后期缩减机器和 zone 数量时，也非常灵活。

王城内城通过副本的方式实现，可以灵活分配选择 zone 进程创建。同时也增强了容灾能力，在国战发起时，可以将王城内城副本创建在非故障的 zone 上。

tconnd 使用集群化模式，与 zone 进程不绑定，玩家集中在某个 zone 上时，不会对 tconnd 造成压力。同时还可以减少对外的 ip 和端口数量。

五、流量消耗优化

5.1 背景介绍

在移动网络下，网络流量也是 RPG 手游面临的重要问题，特别在御龙在天国战类 RPG 手游中显得更为突出。御龙手游流量消耗的压力主要来自两个方面：a) 实时 PVP，游戏场景实时更新，技能，属性，移动广播量大，发包频繁; b) 国战手游需要有宏大的场面，看到更多的人，但是视野大意味广播量大，广播包多。

国战 PVP 中，手机客户端会收到大量的技能包，视野包等，由于玩家手机参差不齐，手机上玩家对流量比较敏感，大量的网络包不仅会带来流量消耗，同时也会给手机端带来 CPU,GPU 消耗，影响游戏流畅度。

因此需要我们在实时 PVP 中降低手机流量消耗，让玩家不用担心流量消耗，可以放心的在手机上尽情享受万人国战的快感。同时通过降低网络流量，也能够有效降低手机客户端网络包处理压力，减少电量消耗，带给玩家更流畅的游戏体验。

5.2 流量优化措施

5.2.1 业务下行包合包

5.2.1.1 游戏下行包特点

发包频率高，单个业务包小。以一测国战数据为例，单个下行网络包有效载荷不足 50%。

5.2.1.2 合包策略

tconnd 网关增加下行缓冲进行业务包合并，当缓冲区大小超过配置阈值或者等待时间超过配置时间时，将缓冲区包发送到网络，多个业务包共用 appolo 头，合并后压缩。

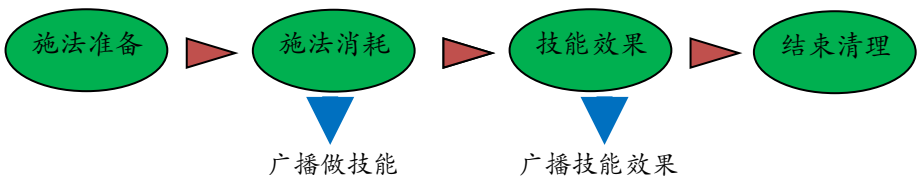
对于需要实时下发的包，通过设置好实时下发标志，可以不用等待缓冲区满或者超时，直接实时下发。

5.2.2 技能流量优化

一测下行数据占比

从统计数据来看，包量和包个数占比前三位的分别是技能效果，属性通知，做技能包。属性通知通常是伴随着技能产生的，比如做技能中的 BUFF 对攻防的影响等等。通过对技能进行优化，可以有效降低网络流量和包量。

5.2.2.1 技能施法流程



技能施法流程

5.2.2.2 技能优化措施

1.技能效果分场景去广播化处理，收敛广播包量

如果是死亡包，因为需要视野内所有人看到，此时的效果包需要在视野内进行广播。对于非死亡效果包，只广播给攻击者和受击者。对于非攻击和受击玩家，血量同步通过主动 pick 查询。

2.群战做技能设定广播阈值，到达域值广播降级

如果是位移技能，视野中玩家需要依赖做技能包更新对应玩家位置信息，因此该技能包需要在视野中进行广播。而对于非位移技能，在国战中大量人群聚集群战时，无需全视野广播，只需要选择部分玩家广播做技能包，已经可以体现出战场的激烈。具体措施如下：如果视野玩家不超过 n 人，做技能包按视野进行广播，如果视野中玩家超过 n 人，则随机从视野中挑选 n 个人进行做技能包广播（包含技能受击目标）。

3.技能效果和表现分离（一段伤害多段表现）

对于服务器下发的单次技能效果，客户端通过分段的形式表现多段伤害，这样可以在一次网络包的情况下表现出多次受击伤害效果，从而节省下发的技能效果包个数。

4.持续技能多段分时上报，去除按碰撞检测上报

对于多段技能，采用分时上报的方式而不采用实时碰撞上报，通过分时合并上报目标，减少技能攻击时上报的技能包个数。

5.2.3 移动流量优化

移动包降频率

降低客户端请求移动包频率，客户端行走预表现，增加服务器的移动误差容忍值（初始容忍 n 秒误差，大约 m 米距离误差容忍）

反作弊

加入移动反作弊，防止客户端外挂，作弊次数多，降低容忍值

5.3 流量优化结果

经过以上介绍的流量优化方法和措施，在视野人数增加了一倍的情况下，点将测试相比于第一次测试的流量消耗情况也得到了较大程度的优化。属性包、技能包、技能效果包数量的占比，都下降为第一次测试时的三分之一到一半。同时，每秒业务包的数量减少了一半，每秒下行的流量减少为之前的大约四分之一。

六、代码性能优化

6.1 背景介绍

MMORPG 游戏通常玩法和系统的数量都非常多，因此代码量也是非常大的。当出现性能问题需要优化时，如何从百万行级别代码的工程里，发现那些性能浪费最严重的代码，进行优化后，如何检验性能优化的效果，需要从工具、原因定位方法、优化策略几个方面入手来解决问题。

6.2 性能优化基础工具

要在茫茫码海中发现那些性能热点，处理掉，再检验成效，需要一些基础设施的支撑。御龙主要利用程序内部的性能统计系统以及一些常用的性能分析工具来发现代码的性能热点，使用灵活的机器人模型代码优化迭代测试，并利用监控系统发现运营环境中的不易觉察的性能毛刺。

6.2.1 程序内部性能统计系统

游戏服务器代码的执行一般有玩家行为驱动和服务器定时驱动两种方式：
由玩家行为驱动时，可以在消息包处理的统一入口处统计消息包的数量、大小、处理时间等信息。
由定时器驱动时，可以把定时器做成动态注册集中管理的方式，这样也可以在定时器执行入口处统计各个定时器的执行次数和执行时间。
所有消息处理时间和定时器处理时间的总和基本上就是整个程序的总执行时间。

构建程序内部性能统计系统的好处有几个：
有了各个消息包和定时器的执行时间，可以在业务层上进行调整尽可能降低 cpu 消耗，例如降低移动频率和服务器定时驱动频率等。
在使用性能分析工具对函数消耗进行排序时，会发现大量消耗时间很少的函数，形成长尾，不易逐一优化，程序内部的统计信息有利于在更高执行层次上进行整体优化。
另外，消息包和定时器的执行数量统计有利于建立机器人模型进行测试、对具体代码的总执行时间进行预估等，程序的总执行时间的统计结果也有利于程序根据 cpu 情况进行自动调整服务。

6.2.2 性能分析工具

良好的性能分析工具可以让性能优化事半功倍。御龙后台在性能优化中尝试过很多性能分析工具，包括 gprof、oprofile、valgrind、perf、vtune 等。使用哪种性能分析工具受到获取简便性、学习成本大小、是否满足性能分析需求等影响，各种分析工具之间有不少差异，需要根据实际情况来选择合适的分析工具。

选择性能分析工具时有几个因素需要特别考虑：

工具本身消耗的 cpu 大小。例如 valgrind 本身 cpu 消耗非常大，不适合使用于服务器负载较大时的性能分析，只适合于性能是线性增长的情况并作低负载测试。Oprofile、perf 等受到系统内核支持的一般 cpu 消耗较低。

能否用于运营环境的性能分析。Gprof 是使用编译时嵌入代码的方式，而且需要程序正常退出才能得到统计结果，oprofile 需要重编系统内核支持，运营环境下使用不方便。

基于功能上的考虑。Gprof 不能统计内核调用消耗，perf、vtune 等基于硬件性能计数器的方式可以提供多种事件的统计分析结果，还有需要考虑可视化操作是否简便等。

6.2.3 机器人测试验证系统

服务器代码执行是玩家行为驱动的，像视野广播等性能消耗与玩家数量非线性关系的情况下，需要模拟大量玩家的群体性行为来测试代码的性能优化结果。把机器人做到灵活可配置，可以根据程序内部的性能统计信息，配置机器人的登录数量、各消息包发送频率和外网一致，模拟运营环境的各种实际场景对优化结果进行迭代测试。

6.2.4 性能监控系统

除了国战、集体过边任务等可预见的场景会出现性能消耗高峰外，还可能在其它不易发现的场景中出现性能毛刺。Mmog 游戏的功能非常丰富复杂，玩家群体行为难以完全预估，服务器机器数量有一两千台，这些因素都导致服务器性能毛刺发现困难。这时候就要通过加入服务器性能监控来发现潜在的性能问题。机器整体的 cpu 负载情况以及程序内部的性能统计信息都可以上报到公司的网管平台，方便我们对各类性能信息做针对性监控。

6.3 原因定位方法

程序内部性能统计系统以及性能分析工具的运用，有助于我们更快发现程序的热点所在，但是并不能帮助我们定位到哪一行需要修改的代码，也不会告诉我们该如何作代码修改，因此借助工具以外，我们仍然需要积累代码优化的经验。

6.3.1 热点粒度

大多数性能分析工具都是以函数为单位进行性能统计的，有些分析工具如 perf 支持定位热点到指令级上，但是要求有较高的采样精度，需要系统内核的支持，并且带来较高的性能损耗，不适宜使用到运营环境上。以函数作为性能统计单位时，如果一个函数代码量过于庞大，就为定位性能热点到代码行带来困难，因此热点区域的代码适宜写成简短的函数。

当无法定位热点到函数中具体代码行时，可以利用程序内部的性能统计系统在函数中若干位置插入统计代码，根据统计日志逐步缩小热点区域的范围。

6.3.2 代码分析

确定热点代码行的另一个重要手段是仔细分析代码。一个函数成为热点的原因可能有很多，包括函数本身的指令数量多且调用次数多、高频调用带来的栈管理开销、磁盘操作、缓冲命中率低、分支预测失败率高、缺页、上下文切换等，需要根据具体代码仔细分析性能消耗原因。也可以使用 perf 等性能事件统计结果进行辅助分析。

6.3.3 经验数据

一行代码是否可能够成热点，可以通过积累一些经验数据进行计算确定。例如 memset 1MB 需要消耗多少时钟周期、各种时间操作函数消耗的时间、文件操作消耗的时间等。积累各种代码消耗的时间经验数据后，可以根据统计日志等信息得到代码的执行频率，再计算代码占用的 cpu 百分比。

6.4 代码优化策略

定位到性能热点并分析原因以后，需要根据具体代码选择合适的优化方向。下面将根据御龙后台优化经验谈谈常用的优化策略。

6.4.1 内存操作

MMOG 后台充斥着大量数据，御龙 zone 进程的虚拟存储空间已经达到几 G 级别，各种数据的存储、修改、回写、大数据包发送等潜伏着大量内存操作，对大内存的频繁操作将会消耗较多 cpu，因此内存操作的优化是经常使用的优化手段。下面以御龙遇到的其中一个内存操作的例子做说明。

在运营环境使用 perf 进行性能分析时发现，memset 操作在所有函数性能消耗排名中排在第一位，其中技能定时器使用的 memset 耗时最多。

分析发现技能定时器中有定时发送持续伤害技能效果包的功能，在消息包字段填充前有 memset 整个包的操作，而由于这个包比较大，只要每秒发送几百个技能效果包，就占用整体 cpu 的 10%。这样的技能包数据是比较容易达到的。
直接去掉 memset 可能隐含部分字段未初始化的风险，我们发现这个技能效果包的最大目标对象数定义为 100 个，而实际使用中是对单个对象使用的，最后我们通过缩小包的大小解决了这个问题。

6.4.2 时间操作

游戏服务器中经常会使用到一些时间操作函数，特别是在服务器定时检查和驱动的操作中，可能需要遍历每一个数据对象进行时间消逝检查，时间操作函数是比较耗 cpu 的，如果对象数很多，而且检查每个对象都调用 localtime、time 等获取时间的函数，就会浪费较多 cpu。

对一些对时间精度要求不是很高的操作，可以在一个定时器中获取到时间并保存起来，其它需要使用时间的地方直接获取这个缓存的时间即可。

时间函数的不合理使用将会耗费大量 cpu，下面以御龙的一个例子作说明。
我们在检查御龙国运时的性能统计日志时，发现一个任务相关的定时器处理耗时非常高，使得进程整体 cpu 在国运时也很高，使用 vtune 采集性能数据发现任务定时器中调用的一个时间相关的转换函数耗时占整体进程的 60%以上：

使用 perf 分析，发现_IO_vfscanf_internal 和__offtime 两个函数耗时最高。但这两个函数是标准库函数，由于去掉了 debug 信息，无法看到调用栈情况。
使用 pstack 多次打印进程的栈信息，发现进程经常性执行到这两个函数上，并

且是从这个时间转换函数中的__mktime_internal 调用的。
因此热点的耗时原因就较为清晰了，因为国运玩法中需要不断定时检查是否任务超时，在计算消逝时间时调用了开发框架库中的一个时间转换函数，这个函数实现时调用了标准库的时候转换函数__mktime_internal，而这个函数在字符串操作和时间计算等较为耗时。

最后我们使用了自己编写的 mktime 时间转换函数来解决这个问题，它是通过计算与已知时间点的差异来实现的，效率提升在 20 倍以上，完全消除了这个性能热点。

6.4.3 cache-miss

由于游戏后台存在大量的数据处理，很容易出现 cpu 缓存的 cache-miss，造成程序执行效率的下降。使用 perf 的 stat 工具分析御龙国战的 zone 性能，发现 cache-miss 情况比较严重。

这是与大量的内存数据引用和内存拷贝有关的。减少 cache-miss 的方法是使用局部性原理，减少存储器变量的引用，或者降低整个内存操作的频率，需要针对具体代码进行优化。
下面是 cache-miss 引起性能严重下降的一个例子。在分析御龙聊天服性能的过程中，发现在消息广播中一个非常简单的获取 player 身上一个字段的函数非常耗 cpu。

把这个函数改成内联后，cpu 有所下降，但是整体 cpu 还比较高。Perf stat 发现进程的 cache-miss 情况很严重。分析代码发现聊天服上有几万个 player 对象，每次对一个国家范围进行广播时会遍历所有几万个玩家找到对应国家的玩家进行广播，以 cache-miss 消耗的时钟周期和使用的 cpu 主频来计算，每秒最高发几百个消息包，由于国战时文本消息广播数量很多，每秒过百的消息包广播请求是可能达到的。解决办法是对玩家进行分国家管理，减少广播时遍历玩家的数量。

6.4.4 分支预测

分支预测失败会破坏 cpu 的流水线操作，降低代码执行效率。从御龙 zone 进程的 perf stat 分析结果来看，该进程有较高的分支预测失败率：
可以使用 perf 工具指定 branch-misses 事件可以查看哪些函数的分支预测失败率最高并进行优化。御龙使用 perf 对非国战 zone 进行性能分析时发现，某个时间比较函数占用的时候排在第二位。

这个函数在 branch-misses 事件排序中排在前列。查看代码发现函数实现非常简单，但是里面包含一个计算和条件判断较多的分支判断，优化时我们把这个把函数里的分支判断去掉，由上层调用保证传入参数的正确性。优化后，这个函数就退出函数耗时榜前面位置。

6.4.5 文件操作

文件数据读写涉及磁盘操作，效率较低，这个容易预估。数据应该尽量在内存中进行缓存，减少频繁的文件存取操作。另外有些隐秘的文件操作也可能导致性能问题。例如 zone 的 vtune 分析结果发现某个函数占用 cpu 较高。这个函数是判断一个操作是否是敏感操作，对每一个上行数据包都会调用这个函数进行检查，函数中有判断一个含敏感操作列表文件是否存在的 access 操作。测试发现 access 操作在国战高峰中调用次数非常多。最后我们通过分析后优化掉了这个 access 文件操作，使这个函数的 cpu 占用下降到可以忽略了。

6.4.6 高热点代码精简

性能热点的存在可以跟代码本身的指令执行数量有关，因此精简热点函数的代码是非常有用的优化措施。

例如御龙的视野广播函数原来占用了较多 cpu。在函数内部，遍历视野列表、获取视野对象、包发送等有较多安全性检查以及异常处理的操作，便这些安全性检查正常情况下不会出现，或者在其它代码中已经做了安全性保证，或者出现异常不影响后续逻辑的继续执行，所以可以直接把这个代码精简掉。由于视野广播执行很频繁，每次广播都要遍历整个视野代码，循环内的代码执行频次非常高，所以精简代码的优化措施非常有效。

6.4.7 内联

简短函数的高繁次调用会带来栈管理的开销，很容易想到把这些函数改成内联函数。一些日志打印、状态检查等函数调用频次非常高，如果使用的总 cpu 较高，就可以改为内联函数，御龙在这方面的例子包括前面 6.4.3 中的 GetStatus 函数和 6.4.4 中的时间比较函数，都取得了较好的优化效果。

七、结语

通过以上的优化方法、工具和策略，《御龙在天》手游服务器在以下几个方面都达到了理想的优化效果：

通过场景切换流程优化，大大降低了玩家场景切换失败的几率；

通过国战架构优化，降低了逻辑开发复杂度，提升了正式环境部署的灵活性和容灾效果，同时也均衡了网络流量对机器带来的压力；

通过合包策略和技能、移动流量优化，降低了客户端侧的流量消耗，也提升了客户端的运行流畅度；

在代码级性能优化部分，通过结合御龙在天手游的实际运营案例，介绍了在服务器侧常用的性能优化工具、原因定位方法和优化策略，希望能给各位读者在遇到性能优化问题时提供相关帮助。

《天天爱消除》服务器性能优化

腾讯互娱高级工程师-罗雄威

一、概述

《天天爱消除》服务器已经在外网稳定运行四年多了，日积月累服务器方面出现了一些问题。主要包括内存，强校验性能，异步开发效率，登录等问题。本文记录这些问题的解决方案和优化效果。

二、服务器进程内存优化

2.1 服务器进程内存现状

《天天爱消除》外网机器负载表现为内存占用率较高，CPU 使用率较低，同时因为是弱交互的手游，网卡流量并不会存在瓶颈。玩家同时在线高峰期，机器内存使用出现告警。核心进程组 gamesvr 和 checksvr 占用内存过多，下图 1 是机器内存使用情况。

```
top - 13:11:30 up 52 days, 14:31,  2 users,  load average: 1.53, 1.31, 1.29
Tasks:  76 total,   3 running,  73 sleeping,   0 stopped,   0 zombie
Cpu(s): 15.3%us,  1.7%sy,  0.0%ni, 82.3%id,  0.0%wa,  0.7%hi,  0.0%si,  0.0%st
Mem:  15728640k total, 15712824k used,    15816k free,        0k buffers
Swap:  2088956k total,  1647856k used,   441100k free,  4575340k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
28481 user00    20   0 8127m 7.0g 356m R 29.9 46.3  32:08.78 redgamesvr
28450 user00    20   0  599m 472m  27m R  8.7  3.1   4:13.69 strongchecksvr
28448 user00    20   0  600m 470m  27m S  7.4  3.1   4:14.56 strongchecksvr
28447 user00    20   0  599m 468m  27m S  6.1  3.0   4:16.75 strongchecksvr
28444 user00    20   0  600m 467m  28m S  4.8  3.0   4:13.53 strongchecksvr
```

图 1：服务器进程内存使用图

分析发现 checksvr 进程内存主要消耗在加载关卡资源，在一个 set 集中，同时启动了 4 个 checksvr 进程，每个进程都加载了所有版本对应的关卡资源。gamesvr 内存使用量正相关于玩家同时在线人数，主要消耗在排行榜。爱消除排行榜是直接实现在 gamesvr 中，并在进程内部缓存。图 2 和图 3 是 checksvr 和 gamesvr 内存使用占比示意图。

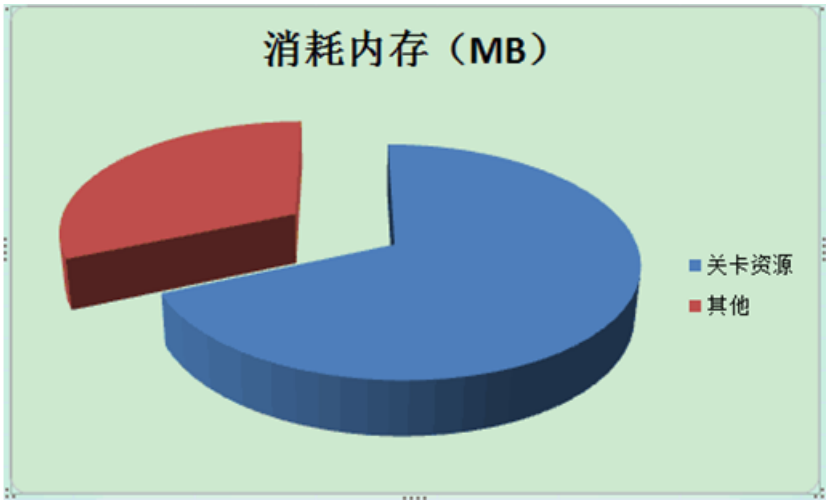


图 2：checksvr 内存使用占比示意图

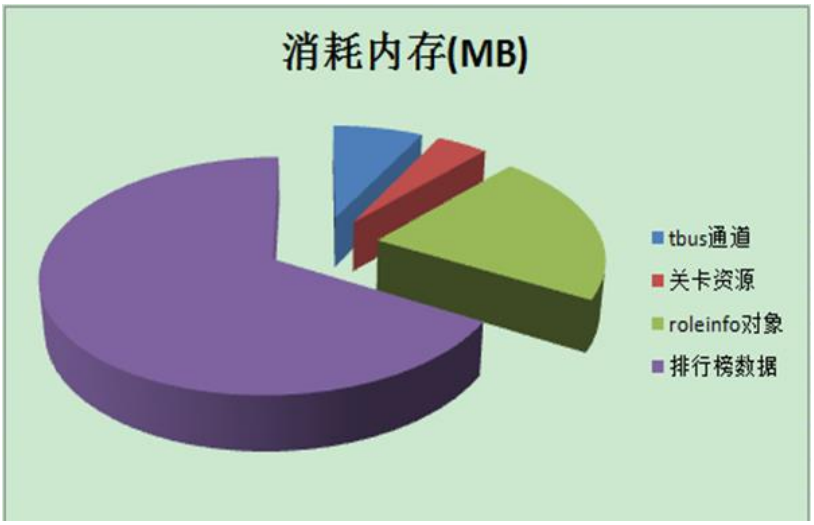


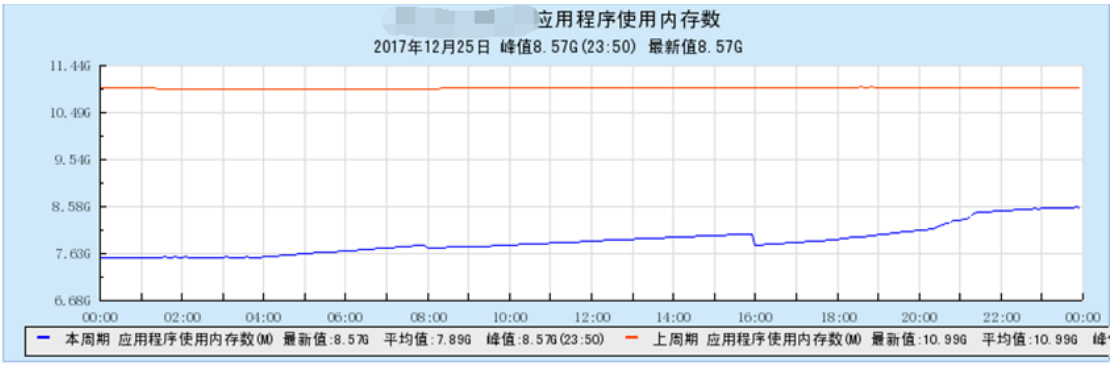
图 3：gamesvr 内存使用占比示意图

2.2 服务器进程优化方案

针对 checksvr 关卡资源冗余带来的内存开销，新方案中结算校验时通过 gamesvr 在请求包中把关卡资源带过来，从而减少 checksvr 常驻内存使用量。为此会带来一点额外的 CPU 拷贝开销。

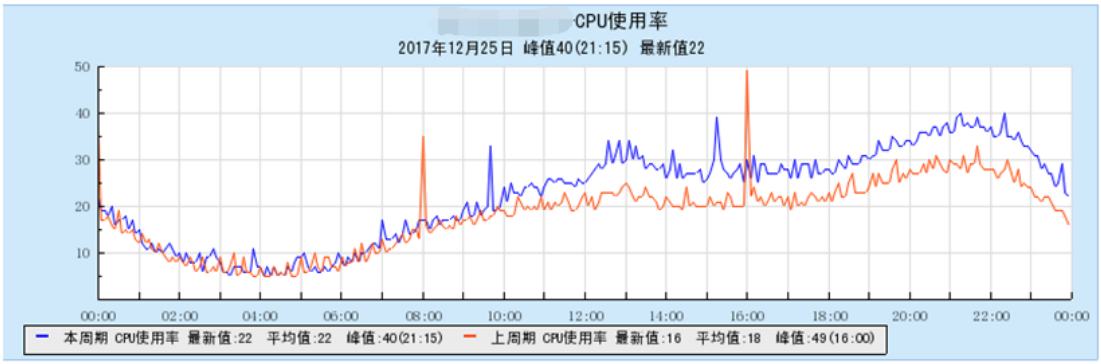
针对 gamesvr 排行榜内存消耗，有两种解决方案。第一种就是把排行榜独立出 gamesvr，形成单独的排行榜服务。但这相当于把内存转嫁到其他机器上面去，因为新排行榜服务需要申请新机器。另外独立排行榜方案需要对 gamesvr 逻辑代码大量修改。第二种方案是采用 CPU 换内存的方式。我们前文提到，核心进程组机器 CPU 使用率比较低。所以最终我们采用压缩排行榜数据的方式优化掉 gamesvr 内存。图 4 和图 5 是核心进程组内存优化效果。从两图中可以看出我

们用不到 10%的 CPU 增长节省了 2.4GB 左右的内存。



行号	时间 (Desc)	应用程序使用内存数(M) 本周期/上周期
1	12-25 23:55 / 12-20 23:55	8.57G / 10.99G
2	12-25 23:50 / 12-20 23:50	8.57G (Max) / 10.99G

图 4：内存优化效果图



行号	时间 (Desc)	CPU使用率 本周期/上周期
1	12-25 23:55 / 12-20 23:55	22 / 16
2	12-25 23:50 / 12-20 23:50	23 / 18

图 5：CPU 消耗对比图

三、checksvr 架构和性能优化

3.1 checksvr 功能概述

checksvr 是爱消除后台用来进行强校验的服务。该服务对玩家每次结算请求，服务器后台运行跟客户端一样的消除库，得出结果，跟客户端上报结果进行对比，从而判断玩家是否作弊。该服务在保证游戏公平性方面有着重要作用。

客户端每个版本对应一个滑动库 so，单个 checksvr 进程加载所有版本对应的校验 so。图 6 是优化前 gamesvr 和 checksvr 通信示意图。

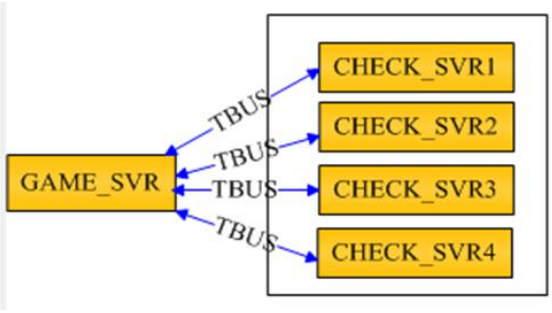


图 6：gamesvr 和 checksvr 通信示意图

3.2 checksvr 存在问题

目前 checksvr 架构主要存在如下几个问题：

编译耗时。每次客户端发布新版本，后台需要把所有客户端版本对应的 so 全部编译。

维护困难。因为后台和客户端共用滑动逻辑库代码，后台每次需要重新编译所有 so，客户端在开发新版本时，滑动库相关代码只能增加不能删除，否则会导致低版本校验 so 编译不过。

性能问题。tbus 轮询收包方式导致校验请求包在队列延迟大,同时单局较验耗时较长。

3.3 checksvr 架构和性能优化方案和效果

针对 3.2 小结提到的问题，我们采用如下优化方法：

1.单进程加载单个版本较验 so，可以解决每次客户端开发新版本，低版本校验 so 需要全部重新编译的问题，但是这样会导致 checksvr 进程数量变多。

2.根据外网玩家在各个版本的分布情况，合并多版本较验 so 到单进程，减少进程数。

3.在 tbus 组件基础上，引入事件通知机制，减少校验包在队列延时。

4.精简滑动库在服务器端的逻辑，perf 工具分析热点函数，提高较验单局性能。

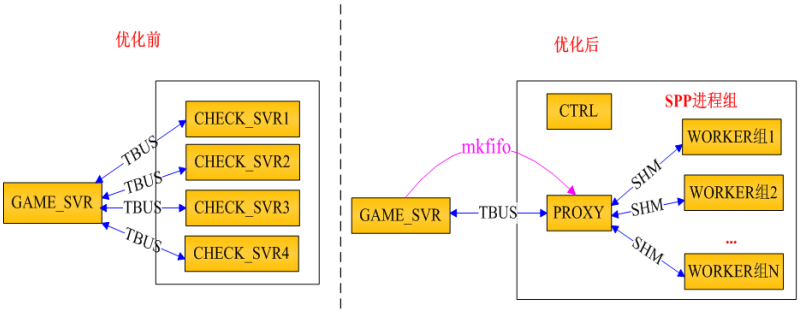


图 7：checksvr 架构优化前后对比示意图

经过上述优化之后，客户端在开发新版本时不再需要重新编译所有低版本 so，同时性能也得到了提升，图 8 和图 9 是优化前后的对比示意图。优化后队列延时平均从 10ms 降低到了 4ms，单局校验平均耗时从 45ms 降到了 30ms。

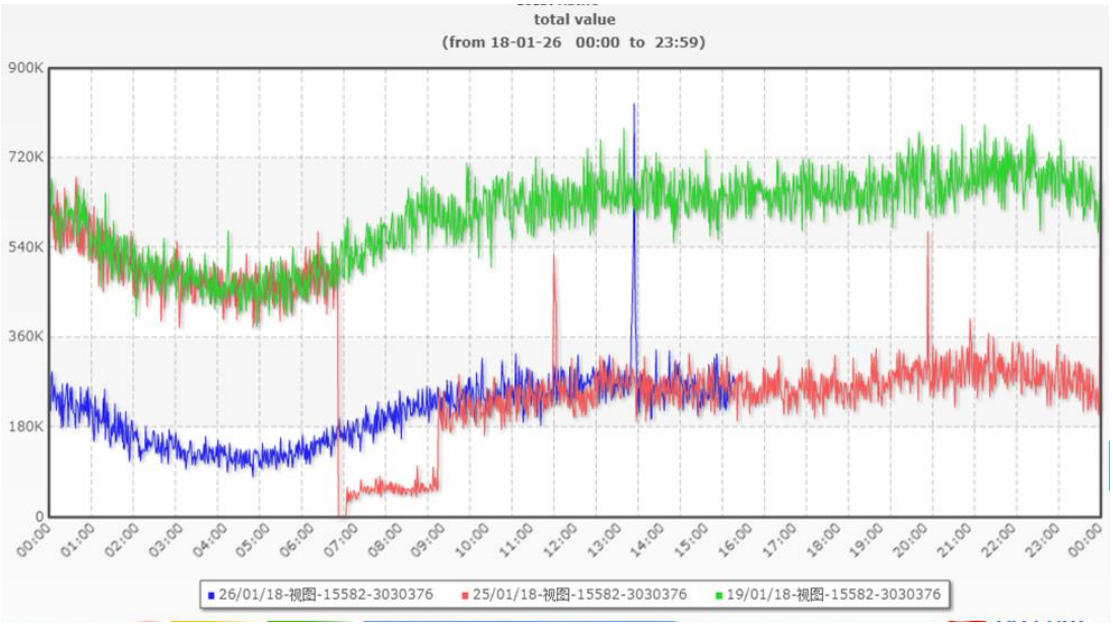


图 8：优化前后校验请求包在队列延时对比示意图

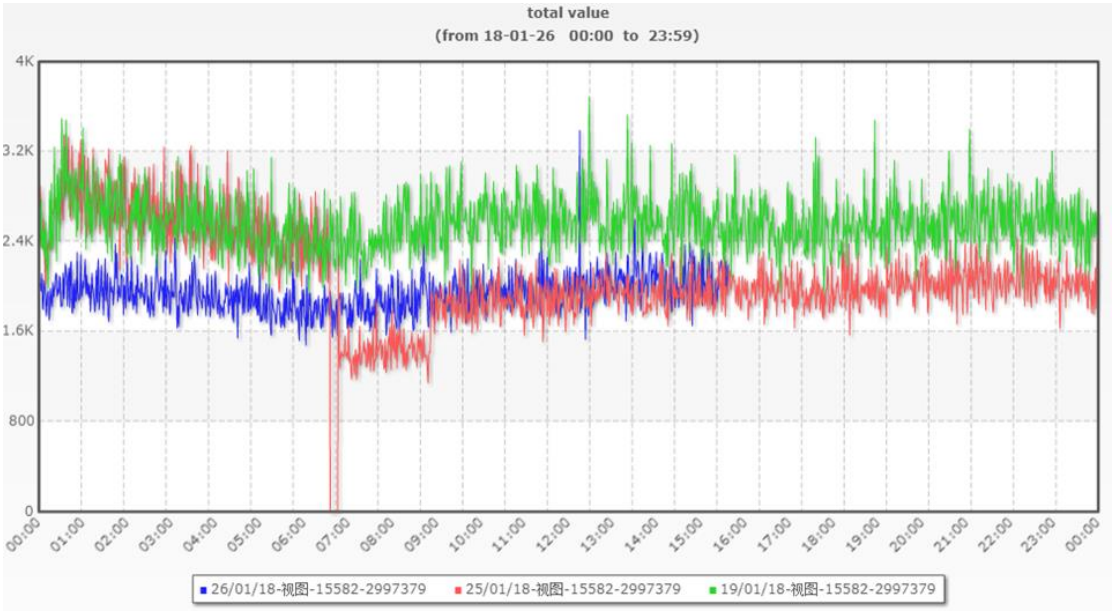


图 9：优化前后单局强校验耗时对比示意图

3.4 http 类服务开发效率优化

《天天爱消除》后台存在一些这样的服务，它们通过 http 协议与外部平台通信完成某些交互。比如 friendsvr 跟微信、手 Q 开放平台通信，完成玩家好友关系链的操作；dealsvr 跟数平通信，完成游戏一级货币钻石的各种操作。这类服务基于 tapp 框架，采用 curl 异步编程模式来并发，它们普遍存在代码维护困难，开发效率不高等问题。

为解决上述问题，我们在这类服务中引入协程库。对代码进行改造，我们选用公司内部开源协程库 libmt。图 10 是在 tapp 框架中引入协程之后的调度机制示意图。

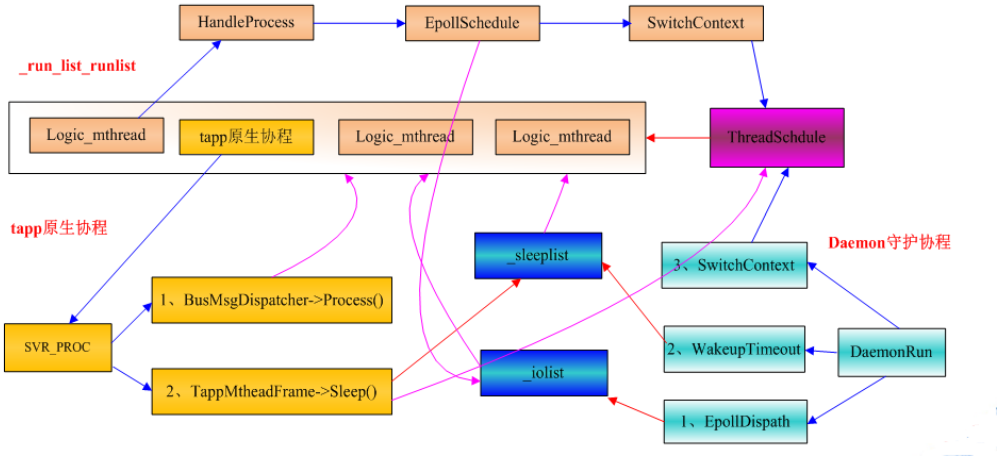


图 10：tapp 框架引入协程调度示意图

引入协程后，这类服务代码开发和维护就像写同步代码一样，结束了异步代码满天飞的局面。图 11 是优化前后两种模式下完成一个需求的开发对比图。该需求就是简单统计下，玩家在登录和非登录阶段去数平拉取钻石成功和失败量。

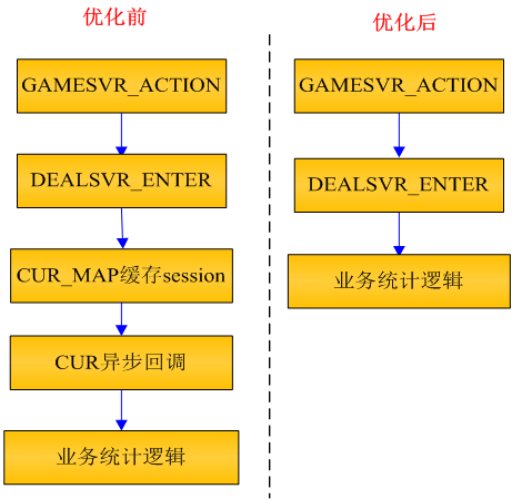


图 11：协程引入前后业务开发对比示意图

3.5 登录过程优化

《天天爱消除》后台已经在外网运行四年多了，游戏业务逻辑不断增加使得游戏在登录过程中需要下发给客户端的包越来越来多。有部分玩家投诉游戏登录不上，客户端表现认为是后台超时。

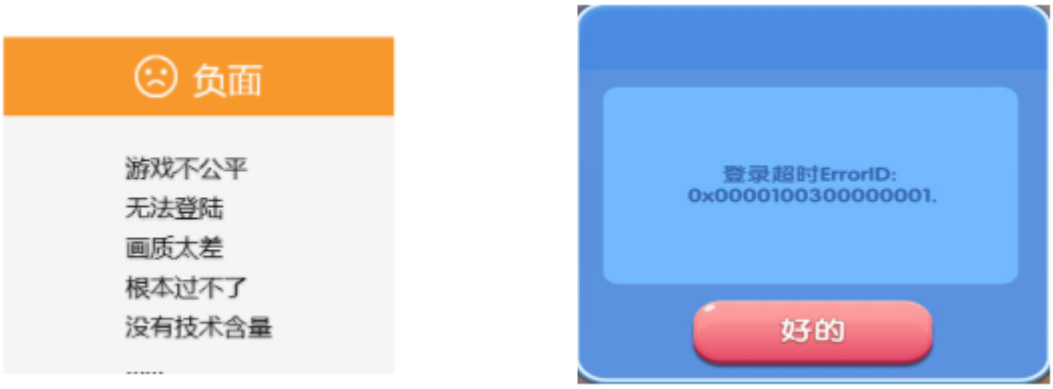


图 12：登录超时表现

外网统计数据表明登录过程中后台下发包量平均在 260 个，并且玩家过关数越多，包量越多。针对登录过程中下发包过多的问题，有两个方案可以考虑，其一是启用 Tconnd 的压缩和合包功能，但爱消除的 Tconnd 运行在 TDR 模式，而 Tconnd 只有在 GCP 模式下面才支持压缩和合包。GCP 模式需要客户端引入新 API，这明显不适用于爱消除的情况，因为我们需要兼容版本众多的老客户端。其二是游戏 gamesvr 自己修改逻辑，针对其中影响比较大的协议按照如下方式分别进行修改：

协议类型	优化方法	优化前包量	优化后包量	包量减少比例
配置	客户端读取本地配置，资源更新	63	0	100%
玩家数据	客户端启用新协议，触发压缩算法	63	13	79%
带逻辑的配置	变更下发时机	20	0	100%

图 13：登录包量优化方案

优化过后外网登录下发包量减少效果比较明显，如下图 14 所示。在我们使用的方案中，新版本客户端基本不需要修改逻辑代码，同时可以通过修改后台服务器逻辑代码做到兼容低版本客户端。

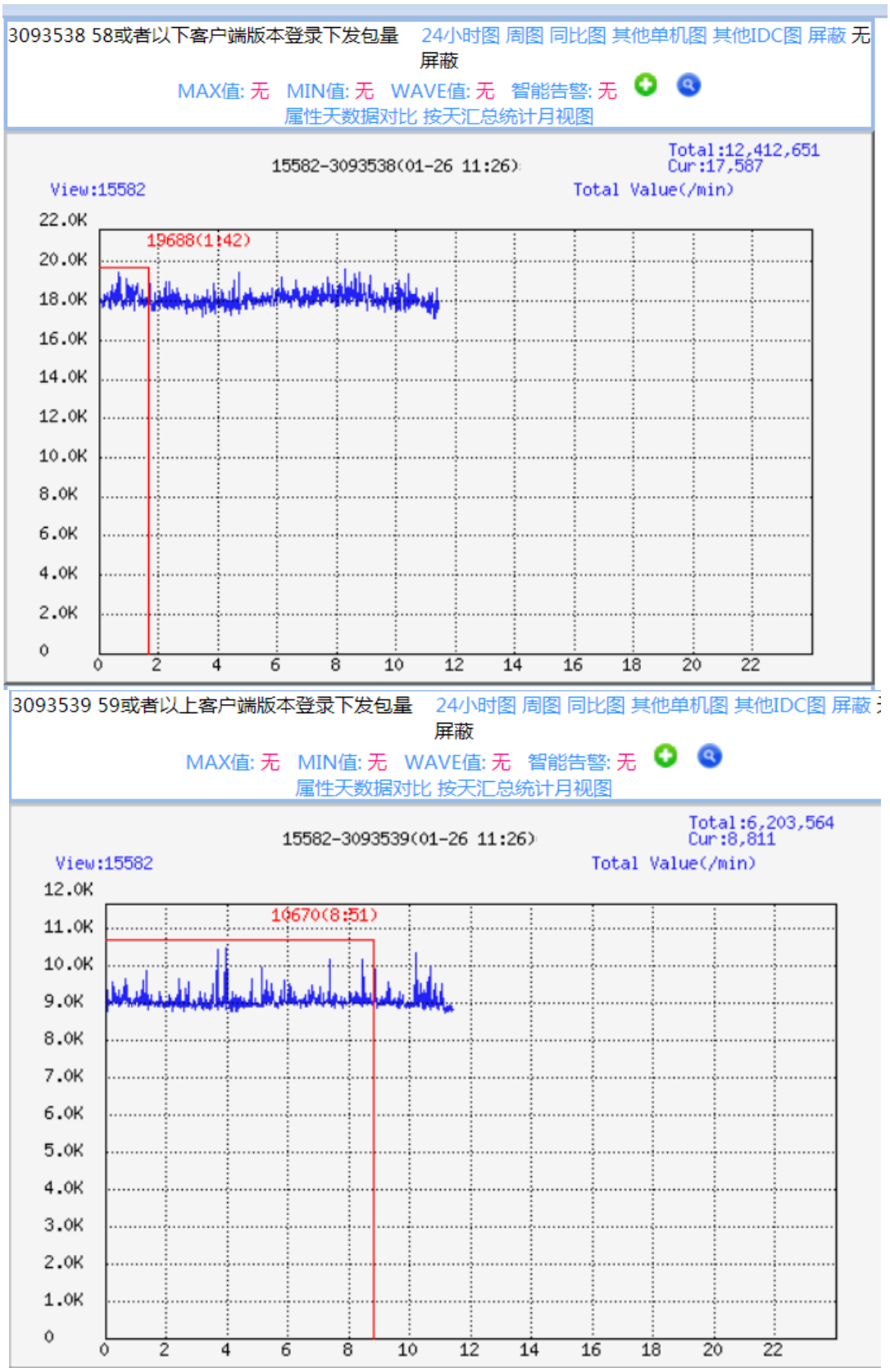


图 14：登录包量优化对比图

移动游戏实现技术优化的解决方案详解

腾讯互娱专家工程师-张丹

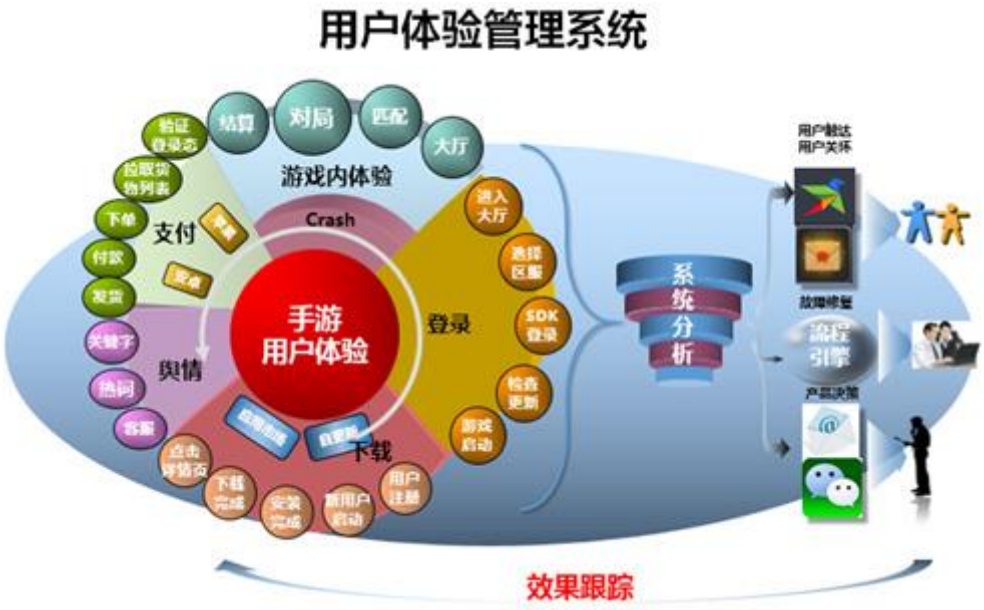
手游经过多年来的高速发展，如今已经进入白热化的红海竞争时代，仅凭一个好的创意就能大获成功的产品已经越来越少，必须经过不断精细化的打磨，追求极致的用户体验，才能做出 S 级大作。

下面，我试着讲述如何从技术角度追求手游(特别是多人对战手游)用户体验提升的解决方案。

我们将用户体验从用户行为划分为五大模块：

下载→登录→游戏内体验→支付→舆情

以下分别从这五个方面描述手游基础技术上，特有的问题和解决方案：



一、下载

移动游戏下载包括两种：

1.应用市场或者官网整包下载。

2.游戏内自更新升级。

剔除掉“用户主动取消”和“无网络”的情况下，成功率可以达到 80%-99%。

下载简单示意图：

下载主要失败原因有(比例较小的原因就不详细列出了。

客户端出现问题的主要原因：

1.有网络但下载不了。

可能原因：

a.用户禁用了该应用市场 APP 的非 WiFi 网络访问权限(在非 WiFi 情况下)。

b.连上了 WiFi，但此 WiFi 需要验证或者此 WiFi 无法访问网络。

c.该网络处于网络拥塞或者信号较弱的情况。

解决方案：

检测网络连通性，提醒用户检查系统设置等。

2.磁盘空间不够。

解决方案：

下载前先检查磁盘空间是否足够，提醒用户清理。

3.创建文件异常/写入文件异常。

解决方案：

此情况可能是存储卡 IO 异常，一般先进行判断 writeleft>0 和进行 fwrite 几次重试即可恢复。

少部分用户可能是手机内存卡有写保护，提醒用户进行设置(一般安卓系统里没有，需要在 PC 里面设置或者 root 用户通过系统软件设置)。

网络问题的主要原因：

1.整包或部分文件被劫持或者 Cache。

此问题移动网络下比较常见，劫持或 Cache 导致的失败率占一半以上，当地运营商(各省策略不一样)由于各种原因，将某些下载文件劫持，导致下载失败。

解决方案：

提供备份 BGP 下载地址，由于 CDN 边缘节点较多，整个 CDN 节点都进行备份，资源浪费比较严重，由于失败用户占比较小，所以采取集中的 BGP IP 下载备份源，当 CDN 下载失败时，直接连接 BGP IP 的 nginx 代理重试下载。

效果：

在某些游戏上，失败用户使用了 BGP IP 直连方式，整体成功率提升了 20~30%。

2.移动网络下，数据文件被运营商网关截断。

解决方案：

少部分运营商 WAP 网关会对单数据包大于 10M 的请求进行截断导致下载失败，解决方法是采用分片下载的方式，当一个下载资源包大于 10M 时，将资源包分成数个小于 10M 的资源包进行下载，本地再进行组合。

3.跨网下载很慢或者失败，由于下载过程中网络发生切换(移动网络切换 WiFi 或者 WiFi 切换另一个运营商 WiFi)导致。

解决方案：

及时判断网络的制式以及和后台通讯看用户的 IP 是否发生变化，如发生变化，重新解析 CDN 域名获取与用户 IP 所属运营商一致的 CDN 资源，并支持断点重连技术。

CDN 问题主要原因：

CDN 的问题主要是和各个 CDN 厂商相关，比如有些 CDN 厂商的 CDN 池分两种：

小资源 CDN 池：整体带宽相对小，支持频繁更新(回源频率快)，边缘节点多，离用户近，下载更快。

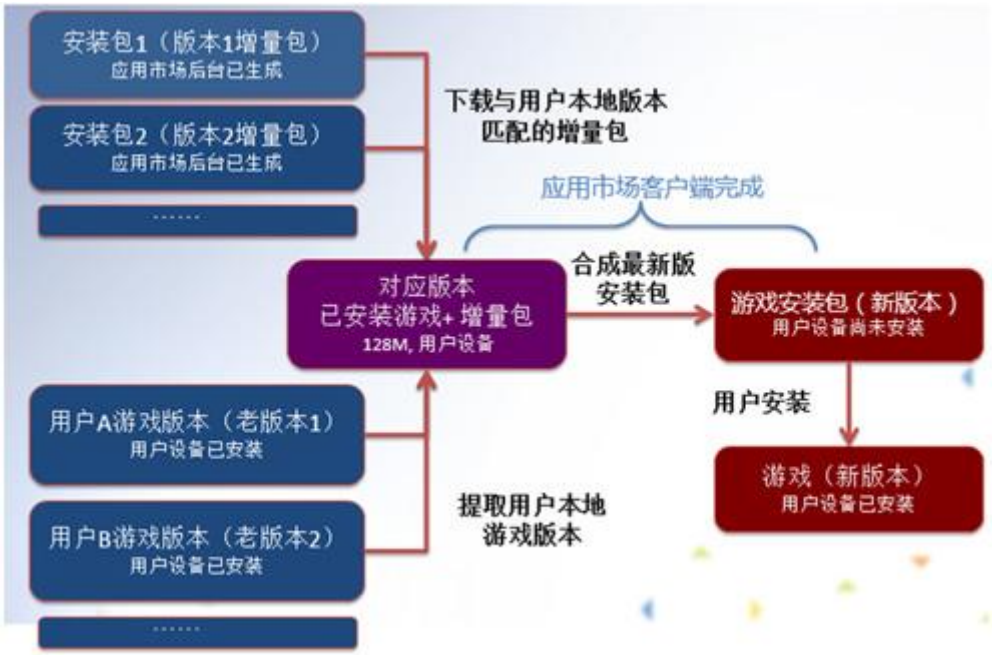
大资源 CDN 池：整体带宽大，不支持频繁更新(大资源对回源服务器压力大)，

边缘节点少，主要集中在大城市，下载相对较慢。
这里就要注意：
1.大的整包资源尽量不要放在小资源 CDN 池，假如该 CDN 每十分钟清理文件并到回源服务器去拉取最新文件，会导致每十分钟你的整包资源会被删掉，并且在回源完成的过程中无法下载成功。

2.图片、配置文件、公告等小资源可放在小资源 CDN 池，这样配置文件，公告等经常更新的资源更新后会更快速及时的被用户下载到。

3.一般 CDN 边缘节点会有 LRU（Least Recently Used）近期最少使用算法，如果你的资源老被淘汰到磁盘上而非内存中，必然导致下载速度相对较慢，如果 CDN 边缘节点负载过高，也会导致下载较慢或者失败，这些需要找 CDN 厂商帮你定位解决。

另外，文件越大，下载时间越长，整体失败率也相对越高(用户看到包太大也会有所顾虑)，而针对游戏版本升级，应用市场还有一种增量更新的方案可以减少文件大小和下载时长。



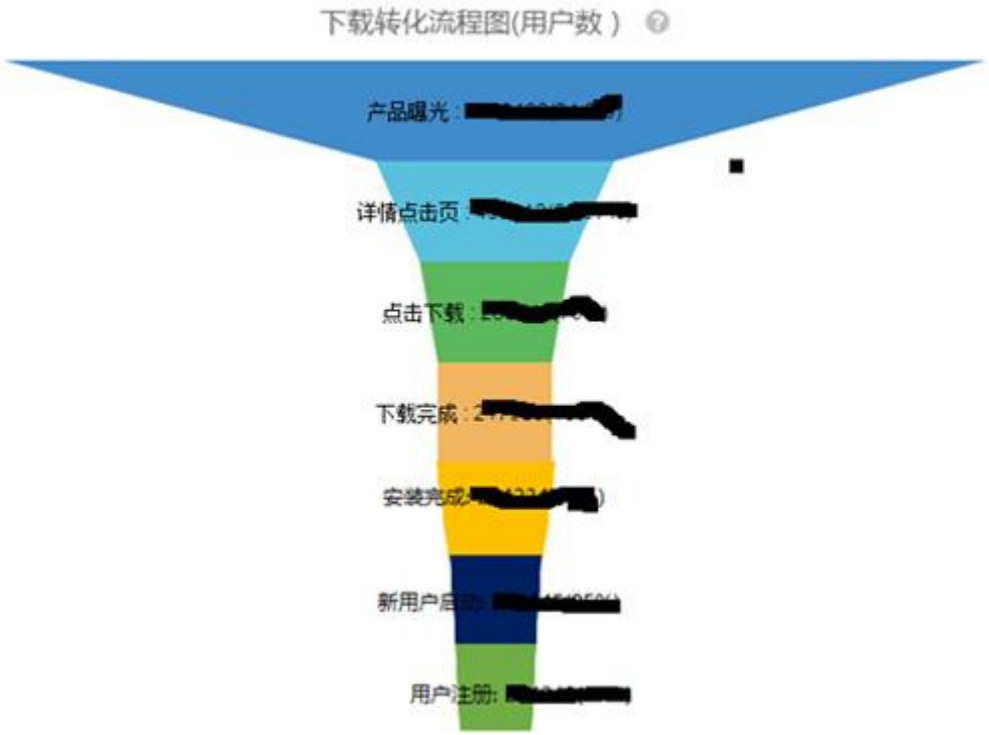
增量更新原理图

这些优化完之后，整体的下载成功率可以提高到 98%以上(剔除掉“用户主动取消”和“无网络”)。

针对应用市场下载，仅仅看下载成功率是不够的，应用市场下载到注册的流程是：

产品曝光→点击产品详情页→点击产品下载→下载完成→安装完成→游戏启动→用户注册

这里要有一个详细的漏斗图，了解到每一个步骤用户流失的情况是怎么样的，和大盘比是否很差，可能在哪些方面有问题：是否产品的曝光标题不够吸引人?是否产品详情页不能打动用户?是否游戏启动后，新手引导设计不合理，导致用户没有注册?……根据每步的数据有针对性的对每个环节进行优化。



二、登录

DNS 解析：

DNS 解析是登录的第一步，也是使用域名业务最常遇到的问题，这里介绍移动网络特有的几个 DNS 的问题：

1.移动网络 LocalDNS 劫持

DNS 劫持是大家遇到最多的问题，移动运营商由于算法、调度和网间流量等种种原因，将域名的 IP 解析成自己网内的一个 IP 或者一个错误 IP，导致玩家无法登录服务器，突发的时候 DNS 劫持率可以达到 2%以上。这个状况大家应该都比较了解，就不再细讲。

2.移动网络 DNS 解析 IP 非本运营商 IP

原因一：

部分运营商骨干网带宽较小，访问 LocalDNS 得到的是本运营商的接入点 IP(因为我们做了分运营商接入)，但是用户访问的时候，运营商判断出口带宽不够，就把流量调度到租用的第三方出口，也就是说获取 DNS 的网络路径和流量的网络路径不一致，我们最终看到用户的出口 IP 是第三方的 IP，这样就会导致用户出现跨网访问。

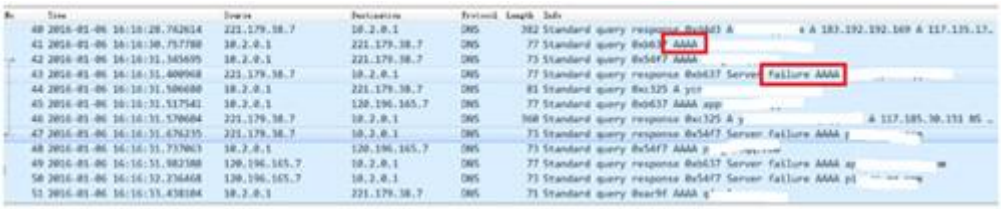
原因二(非常少量)：

运营商为了防止 DNS 服务器故障，做了第三方 DNS 服务器备份，当 LocalDNS 故障时，用户访问到第三方 DNS 服务器得到其他运营商的接入点 IP，导致跨网访问。

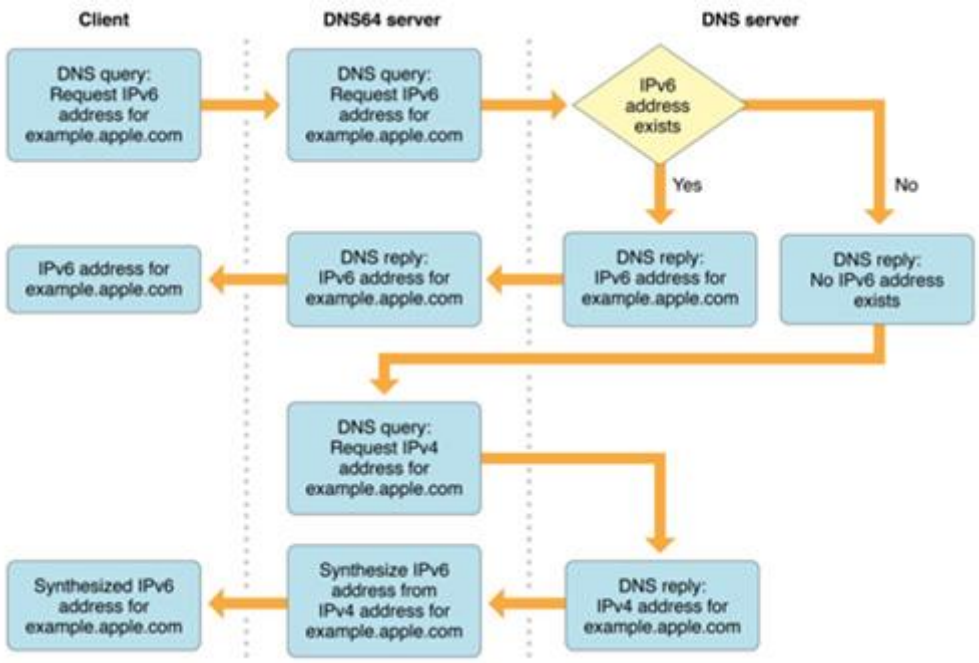
3.移动 4G 网络 DNS 解析很慢

这是 4G 网络的一个特有问题。

通过一些用户投诉我们发现有些游戏登陆非常慢，而实际上他的网络质量还是很好的，玩的过程中即便延时也很快。于是，我们抓包发现：



移动 4G 用户 DNS 先去查询 4A 地址(IPV6 地址)，大部分业务现在都没有配置 4A 地址，所以正常的应该返回 4A 地址为空，然后就会去查询 A 地址，获得正常的 IPV4 地址。一般流程如下：



可是这里 DNS 服务出现问题，不是返回 IPV6 地址不存在，而是返回 4A 地址查询错误：

```
▸ Frame 47: 73 bytes on wire (584 bits), 73 bytes captured (584 bits)
▸ Ethernet II, Src: Harmonix_30:30:30 (00:30:30:30:30:30), Dst: 00:71:7a:6f:6e:65 (00:71:7a:6f:6e:65)
▸ Internet Protocol Version 4, Src: 221.179.38.7, Dst: 10.2.0.1
▸ User Datagram Protocol, Src Port: 53 (53), Dst Port: 60346 (60346)
◂ Domain Name System (response)
    [Request In: 42]
    [Time: 0.330540000 seconds]
    Transaction ID: 0x54f7
  ▸ Flags: 0x8182 Standard query response, Server failure
```

这个时候部分安卓系统就会出现问题，系统认为查询错误不是没有 IPV6 地址，而是系统错误，就会不停地重试去访问 4A 地址，甚至重试几分钟后才去获取 IPV4 地址，导致业务一直卡在 DNS 获取地方，登陆游戏很慢。

既然有这种问题，那么为什么没有大面积投诉呢?是不是可能只是部分网络有这个问题?

经过查询相关文档，我们发现，目前只有 4G 网络是支持 IPV6 的。

于是我们对移动，联通，电信 2/3/4G，WiFi 网络都进行了测试，发现只有移动 4G 有这个问题，可是当我们在另外一个省移动 4G 进行测试时，我们发现没有去进行 4A 访问。

难道查询 4A 还和省份有关系?最终经过我们重重跟进，我们发现查询 4A 地址(IPV6)，不仅仅和网络制式有关系，和手机型号或定制系统也有关系。因为我们另外一个省是用小米 note 去测试的，而我们自己使用华为 mate 7 测试；另外一个省用华为 mate 7 测试也是要先访问 4A，而小米 note 无论哪个省移动 4G 都不去查询 4A 地址。

于是，我们得出结论：

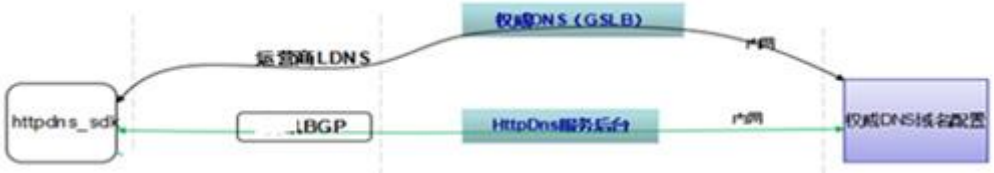
网络制式	查询 4A 地址
2G，3G，WiFi	×
电信 4G	×
联通 4G	×
移动 4G	√

支持 4A 查询的移动 4G 网络，也只有手机支持查询 4A 地址，才会去查询 4A，这也是用户投诉面积不大的原因。

另外，我们通过抓包，同样在移动，联通，电信 4G 情况下，支持 4A(IPV6)的手机也没有去查询网络是否支持 4A(IPV6)，就直接发起 4A(移动)或者 A(联通/电信)查询，我猜测网络是否支持 4G，应该是核心网会知会手机，这个知会并不是走网络协议，而是 3GPP 核心网内部的一个协议，所以我们抓不到包(仅是我的猜测，还未得到官方证实)。

说了这么多，那么解决这些 DNS 问题有什么办法?

我们主要采用 httpdns+LDNS 结合的方式：



技术原理：

A、SDK 通过高效的 BGP 线路直接访问 HTTPDNS 后台，获取域名的最优 IP，客户端获取到业务 IP 后，就直接往此 IP 发送业务协议请求。

B、基于容灾考虑，保留使用运营商 LocalDNS 解析域名的方式作为备份线路。(因为我们发现，有些 BGP IP 也会被极少数运营商劫持)

由于优先取 HTTPDNS,而不是 LDNS,跨运营商和 4A 的问题也相应得到解决。

通过搭建我们自己的 HTTPDNS 服务和 LDNS 双备份服务，发现域名劫持率基本由 1.5%降为 0.1%，跨运营商访问率也由 4%基本降为 0，4A 导致的登录慢情况也不再出现。

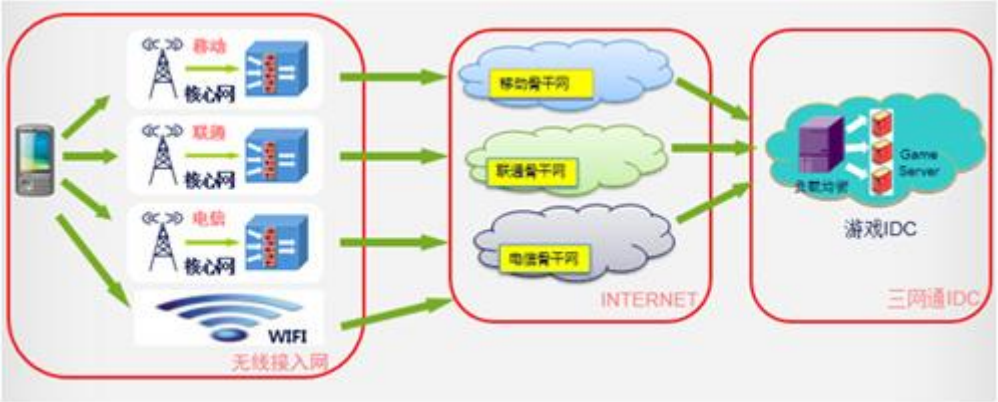
网络连接：

网络连接主要有以下几个问题：

1.运营商之间出口带宽小，网络不稳定，如业务部署的服务器和用户不在一个运营商，会导致连接超时或丢包严重。

这个是最基本的问题，大家应该都有解决方案，一般有 2 种：

分运营商接入：



使用 BGP IP：

百度搜索 BGP 的解释：边界网关协议(BGP)是运行于 TCP 上的一种自治系统的路由协议。 BGP 是唯一一个用来处理像因特网大小的网络协议，也是唯一能够妥善处理好不相关路由域间的多路连接的协议。

BGP 实现单一 IP 覆盖所有运营商用户的效果，利于运营资源利用率的提升，并能彻底解决用户跨网穿越的问题。

对于解决中国这么复杂的运营商网络，BGP 是一个不错的解决方案，但 BGP 资源一般都比较贵。

2.手游用户经常在国外访问我们的服务器。

手游用户由于出国游玩或者居住国外，经常会从国外连入我们国内的服务器，由于中国对国际出口的管制，导致网络质量会非常差。

有能力的公司可以增加国外接入点(建议在香港)，然后调度国外用户接入国外接入点，再通过 vpn 或者专线接入国内服务器(这个 vpn 或者专线也会受到国家监管)，提高国外用户访问质量。

这些问题相信大家都知道，这里不再细说。

除了这些基础问题外，我们来看看一个典型的手游登录流程：

启动游戏→检查更新→下载更新包→sdk 登录→选择区服→进入大厅

用户启动游戏后，一般先检查是否有更新包，如果有的话就下载更新(选择更新

或强制更新)，然后调用个平台的 sdk 进行登录授权，通过后选择区服，然后进入游戏大厅。

登录成功率低可能有很多原因，我们要知道每个环节的成功率和流失率是怎么样的才能细致的去做优化，比如用户是不是检查更新失败，是不是我们的更新服务器出了问题?用户是不是更新下载失败(前面已讲过原因和方法)?是不是登录 sdk 出了问题?是不是大区列表拉取失败?上次登录服和推荐服是否错误?最后才是连接游戏服务器失败。

三、游戏内体验

Crash 问题(闪退)：

手游 Crash 对用户带来的体验伤害，大家应该都已经非常清楚了。Crash 严重会导致大量的用户流失，所以对手游进行 Crash 监控和优化非常重要!

这里首先大概介绍下 Crash 的捕获和获取堆栈信息的方法(方便定位问题)：

Android：

Java 层异常的捕获，一般是在启动时设置异常回调：

void java.lang.Thread.setDefaultUncaughtExceptionHandler(UncaughtExceptionHandler handler)

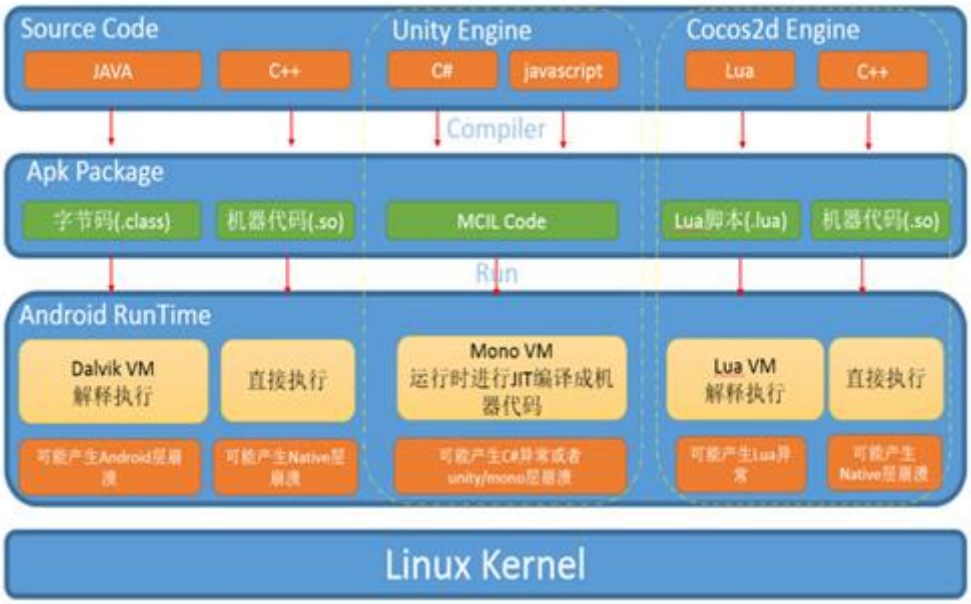
一旦发生异常会调用到 handler 进行处理，handler 参数中带有 Thread 和 Throwable，可以通过这两个变量来获取线程堆栈和异常堆栈。

public static interface UncaughtExceptionHandler { void uncaughtException(Thread thread, Throwable ex); }

但是游戏一般都不是用原生 Java 开发，都是使用游戏引擎，主流游戏引擎 Unity

是由 C++(native)开发的，一般 native 的收集方法：
启动时注册信号处理
int sigaction(int signum, const struct sigaction *act,struct sigaction *olda
ct);
一旦发生崩溃会调用到信号处理函数，只不过 native 收集会更麻烦一点，因此它把堆栈放在一个特定的内存中，需要 fork 出一个子进程去获取。

iOS：
程序启动，一般可以通过 signal 方法注册信号处理，signal(int signum, void (*shandler) (int));
系统调用 AppDelegate 的 didFinishLaunchingWithOption 方法处理，捕获 SIGABRT，SIGILL，SIGSEGV 等信号，当发生 Crash 的时候通过 backtrace 和 backtrace_symbols 这两个方法是追溯到出错堆栈。
并且，仅仅关注 Crash 还不够，我们还要关注引擎或者开发语言脚本的异常，因为这些异常可能最终导致 Crash，也可能导致黑屏，花屏，数据显示异常，画面卡顿等，要实现一个完善的 Crash 和异常收集机制：

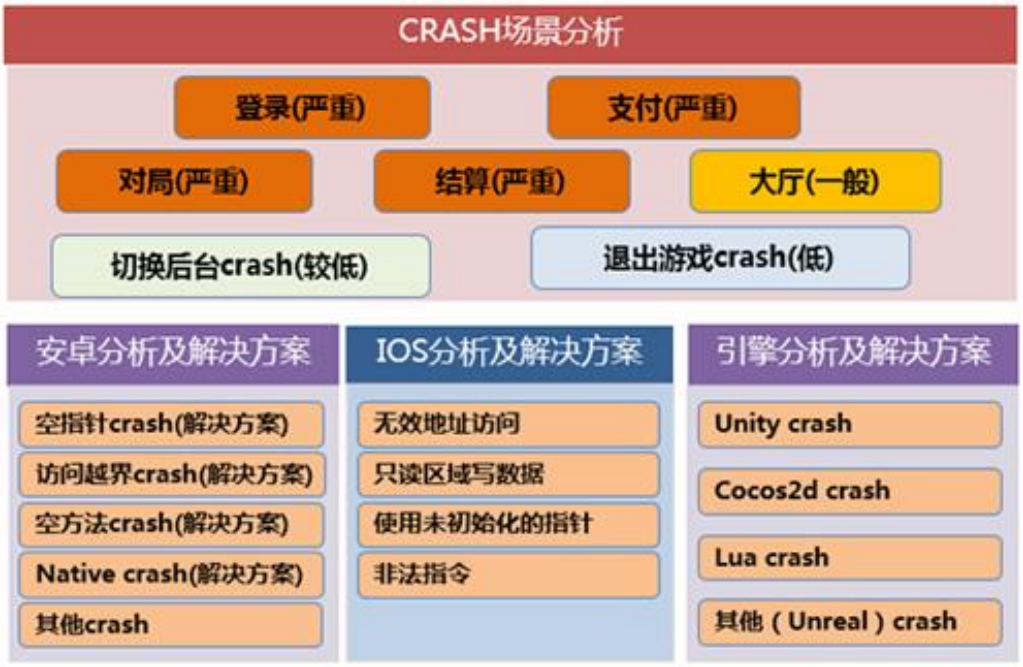


Crash 信息收集上来之后，crash 次数非常多，我们必须集中精力优先解决重要的 Crash 问题，而一些用户感觉不出来的 Crash，其实可以不用解决或者放后解决，例如游戏切换到后台或者在游戏退出的时候产生的 Crash，关于退出 Crash，我们在 unity 中就发现一个典型例子：

我们发现很多游戏 Crash 的堆栈信息是发生在 nativeDone,，nativeDone 以上是 native C++代码，通过反编译的追查，我们最终发现 Application.Quit 来调用的 nativeDone，我们只用改写 Application.Quit 函数，只要调用 Application.Quit 函数，调用 System.Diagnostics.Process.GetCurrent-Process().Kill()，这样就不用走到 nativeDone 了。

Unity 引擎游戏还有一个比较典型的 Crash：
很多 unity 游戏堆栈中，都有 nativeInjectEvent 这个函数，这个 Crash 是 unity4.x 的问题导致的，5.x 似乎不会有这个问题。经过不断重现，我们发现这个跟输入法有关系，由于游戏拉起后立即输入所致。可能的原因也许是引擎尚未初始化完整，即尝试处理某些输入导致。解决方案就是在启动后立即屏蔽输入，不将其传递给 unity，直到我们真的需要输入时为止。

这个 Activity 应该直接或者间接继承于 unity 的 UnityPlayerNativeActivity。而我们要做的就是用一个开关来禁止或者允许 android 事件传输给 unity 的命令队列，重写 dispatchKeyEvent 等几个函数就可以解决。
其实还有很多 Crash 都是比较初级的编码规范没有执行好，例如空指针导致 Crash，越界访问，调用空方法，地址无效，指针未初始化等等，解决起来也相对比较简单。



一款手游要想 Crash 原因不造成用户流失，个人觉得 Crash 率应该控制在 3%以下，如果要做 S 级精品手机游戏，Crash 率要控制在 1%以内最好。

解决了 Crash 的问题，我们就要分场景解决游戏内体验的问题：

我们主要将游戏(特别是有 pvp 对战的游戏)分为四个场景：

大厅：

大厅承载了很多内容，包括各种玩法的入口，活动，任务，商城，背包，战队，联盟等等，这里我们认为对用户体验有较大影响的主要有：

公告弹出时长(拉取公告信息时间太长会导致用户界面假死)，背包加载时长，活动加载时长，商城打开时长等……这些都是用户最常用的一些功能，且需要拉取服务器数据，打不开或太慢的话会严重影响玩家玩游戏的耐心和体验。

另外，我们还会去捕捉大厅的屏幕热力区域，看看哪个区域是玩家点击最多的地方，从而去分析玩家对哪部分玩法最感兴趣，或者说可能是哪个按钮的点击区域有问题导致经常按不到等。

匹配：

匹配一般用于多人 pvp 的游戏，即将酣畅淋漓的对战就要开始了。
这里我们主要关注匹配成功率，平均匹配时长，地图加载时长等。手游的玩家相对是比较没有耐心的，一旦匹配时间，加载时间过长，很容易就一个 home 键将游戏切到后台，导致出现用户挂机或游戏重连，严重影响游戏的对战性。

对局(重点)：

对局是 pvp 游戏的核心玩法，这里也是我们游戏内体验的核心场景，这里也是我们关注的重点：

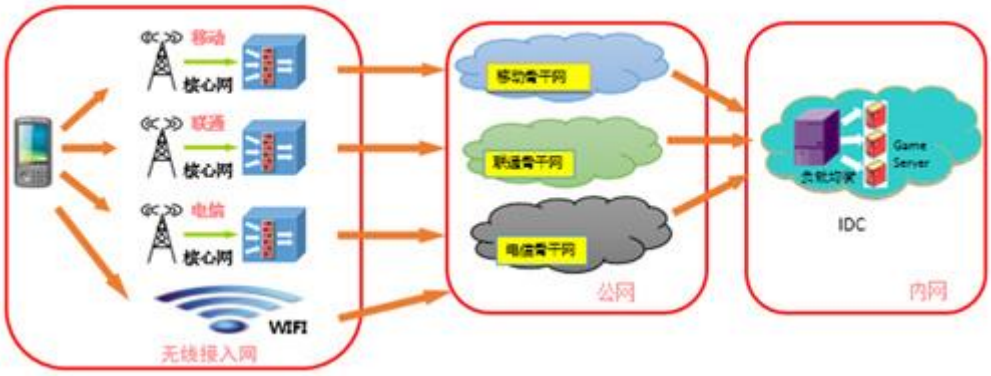
FPS：
FPS 采集方法：unity 每一帧都会执行一遍脚本的这些函数：update->lateUpdate->onGUI，典型的做法是 new 一个 gameobject 并挂载脚本。
在脚本的update函数中统计，update函数中，每次执行update函数，frame++，并累积时间，当累积时间>=1s 时，计算 fps = frame/time。
一般认为，平均 FPS>25 帧是体验较好的情况，平均 15<FPS<25 帧是体验一般的情况，平均 FPS<15 帧是体验较差的情况。如果一局当中 FPS 突然下降 10 帧以上，我们认为是一次卡顿，卡顿次数多了用户玩的体验就非常差了。
针对 FPS 较低的用户，我们通过大数据分析出一个“机能库”，把相应手机机型的推荐配置发送给用户，例如小米 note 手机关掉语音，音效，降低画质，每秒帧数，FPS 分别可以提高多少等，帮助玩家尽可能的提高游戏的体验。

延时：

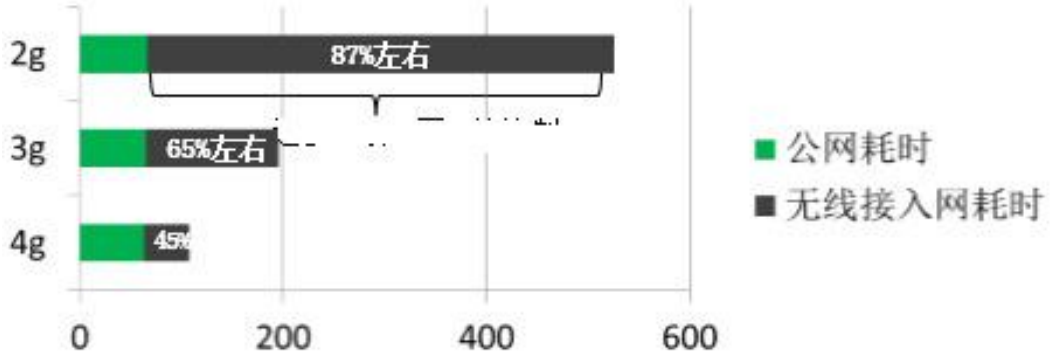
延时的采集方法：每 3~5 秒对游戏服务器进行一次延时测速，可以采用心跳包的模式，这里建议采用和游戏同样的协议，例如游戏室 TCP 连接的，那么，我们使用 TCP 心跳进行测速，游戏是 UDP 连接的，那应该使用 UDP 心跳进行测速，而不是统一使用 PING 进行测速，因为有可能和实际游戏使用的协议延时不一致，而且有些服务器禁止了 PING 协议。

网络低延时是 pvp 游戏的核心诉求！

典型的手游网络的架构图



其中在 2G 情况下，80%以上的延时是在运营商核心网部分，这里确实没有什么办法优化；3G 大概 60%的延时是在运营商接入网部分，而 4G 大概只有 40%左右的延时在核心网部分(取决于游戏服务器的部署)。



而移动手机网络相对固定网络来说，有着明显的特点：

1.网络切换，移动网络切换到 WiFi，WiFi 切换到另一个 WiFi，移动网络切换到另外一个移动网络(目前已经有双卡双待手机支持)等。

2.网络中断，当电话或者短信来的时候，移动网络会中断，有些 4G 网络(中国移动)会切换成 2G 网络，进电梯/地下室等等也会导致网络短暂中断。

3.网络拥塞，人员较多的地方，例如医院，大商场，演唱会等等，即使信号满格也会网速非常慢，偏远地区/室内信号不好的地方，只有 1，2 格信号也会导致网速变慢。

那么，怎么解决这个问题?

我们分为两个层面去解决，一个层面是降低运营商接入网延时，一个是降低公网延时。

运营商接入网部分：

3G：3GPP 协议里面 PDP 和 DSCP 都具有 QOS 保障字段，可以进行无线核心网的网络质量保障。

类型	优点	缺点
DSCP	可以对应用级别（IP，PORT）进行QOS保障	1. 不是所有的核心网设备都支持 2. 只能对下行包打标记和优先级调度
PDP	1. 基于通信隧道的协议，基本所有核心网设备都支持 2. 可以对手游上下行数据包都进行QOS保障	部分手机只支持单PDP，QOS保障是对所有应用生效

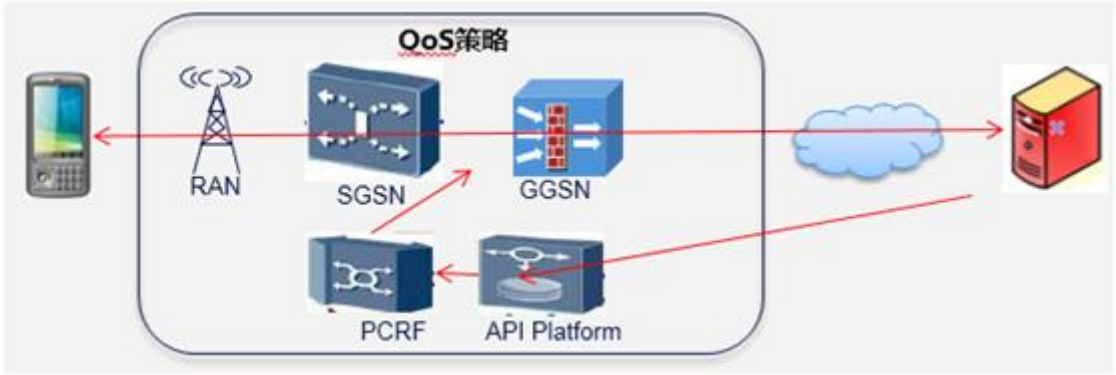
4G：LTE 网络定义了 QCI 可以对数据包进行管理和保障

QCI	资源类型	包延时预算(ms)	业务举例
1	GBR	100	传统语音
2		150	传统视频(实时流媒体)
3		50	实时游戏
4		300	非传统的视频(缓存的流媒体)
5	non-GBR	100	IMS 信令
6		100	语音，视频 Video（实时流媒体），交互游戏
7		300	视频(缓存的流媒体) TCP-based(e.g., www, e-mail, 聊天, ftp, p2p 文件共享, 渐进式视频, etc.)
8			
9			

其中还定义了 QCI 等于 3 就是用于实时游戏。

GBR:
QCI=1: Example Services: Conversational voicemscbsc
QCI=2: Conversational Video (Live streaming)
QCI=3: Real Time Gaming
QCI=4: Non-conversational voice (buffered streaming)
Non-GBR:
QCI=5: IMS signaling
QCI=6: Video (buffered streaming), TCP-based (e.g. www, email, chat, ftp, p2p file sharing, progressive video,etc)
QCI=7: Voice, Video (live streaming), interactive gaming
QCI=8: Video (buffered streaming), TCP-based (e.g. www, email, chat, ftp, p2p file sharing, progressive video,etc)
QCI=9: Video (buffered streaming), TCP-based (e.g. www, email, chat, ftp, p2p file sharing, progressive video,etc)

我们和运营商，设备商积极沟通，发现是存在调用运营商的平台去对用户核心网的 QOS 策略进行设置的可能的，不过这个推动所有运营商都开发这个能力，需要一定的时间。

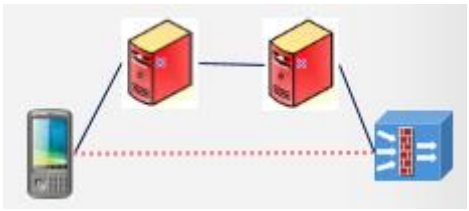


公网部分：

手游和端游在部署上一个显著特点：

端游一般是分区域部署，用户就近接入游戏大区，例如北京电信一区，上海联通一区，广东电信二区，游戏服务器都是按区部署在北京 IDC，上海 IDC，广东 IDC；手游一般都是集中部署，也不会按地区去进行分区，很多游戏还是全区全服游戏，游戏服务器一般都部署在一个 idc，所以 internet 公网的延时会比 PC 端游更高。

公网典型是通过搭建加速代理的方式进行加速：



卡顿率：

我们把 FPS 降低 10 帧以上，网络延时突然增加 100ms 以上定义为卡顿，这两种情况都会导致用户口中的“卡”，对游戏体验影响严重。

拖拽率：

拖拽的大概原因就是客户端本地会进行一定的惯性计算，当服务器负载过高，或者网络延迟严重的情况下，会导致客户端收到服务器的数据后对客户端的数据进行校正，就会出现拖拽的情况，同样，拖拽对游戏的体验影响比较严重。

掉线率：

掉线对玩家体验的影响这里就不用强调了。

除了关注以上这些数据外，我们还要关注一些基础数据，例如：
手机 Cpu 使用率，内存使用率，DRAWCALL、平均三角面(次)、电量消耗，流量等，这些也对游戏体验有不同得影响。

结算：

结算是我们这局惊心动魄对战的结果，也是对玩家这局游戏的一个评定，进行排位，积分奖励，任务奖励，甚至是货币奖励，很多玩家就是冲着这个奖励而来。
这里，我们主要关注结算画面异常率，结算数据异常率，结算页面弹出时长，结算奖励/积分到账率等。

四、支付

目前手游主要支付包括两个平台：苹果支付和安卓支付。
苹果支付统一走苹果 IAP，安卓由于 Google Play 未进入中国，大多使用第三方渠道 sdk 进行支付。

支付的典型流程为：
验证登录态→拉去货物列表→下单→付款→发货

支付失败，我们需要关注每个环节的失败率，特别是区分出用户主动取消和支付过程失败。
例如 iOS 支付失败的典型原因有用户安装了 IAP 破解插件的越狱手机，越狱手机的票据非法或者被替换等。

由于支付大多都是由第三方 sdk 实现，优化方案也一般由 sdk 实现，这里就不在详述。

这里特别提醒下关于 iOS 支付，黑卡，恶意退款，汇率差三大 iOS 黑产业链，给手游的 iOS 收入带来绝大部分损失。这些产业链上每个角色分工非常明确，
黑卡：与 iTunes 账户绑定的非法信用卡，这些账户或为 APP 余额账户，或为透支账户，来源遍布美国、日本、加拿大等许多国家，倒卖者背后是破解或者盗取他人信用卡的黑客。

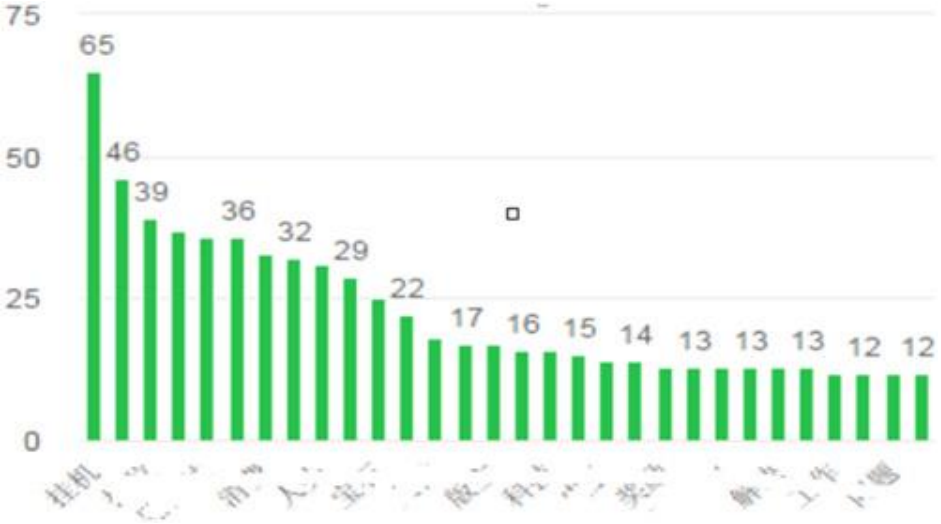
恶意退款：就是先 IAP 付款，然后打电话给苹果投诉，要求退款，空手套白狼，网上不时刷出最新有效的退款理由。
关于黑卡和恶意退款，苹果这边也在不断的优化处理方法，我们也根据用户的一些游戏行为去实时分析识别这些黑卡和恶意用户，进行封号等进一步打击。
汇率差：利用国际汇率的大幅波动，而苹果的汇率更新较慢的特点，像卢布，日元大跌的时候，淘宝上卖家利用汇率差打折售卖游戏币，赚取差价，给游戏 iOS 收入造成巨大损失。

一般做法是支付前识别异常货币，禁止进行支付，事后发现结算货币为异常货币禁止发货，甚至封号并提示用户后续处理措施。

五、舆情

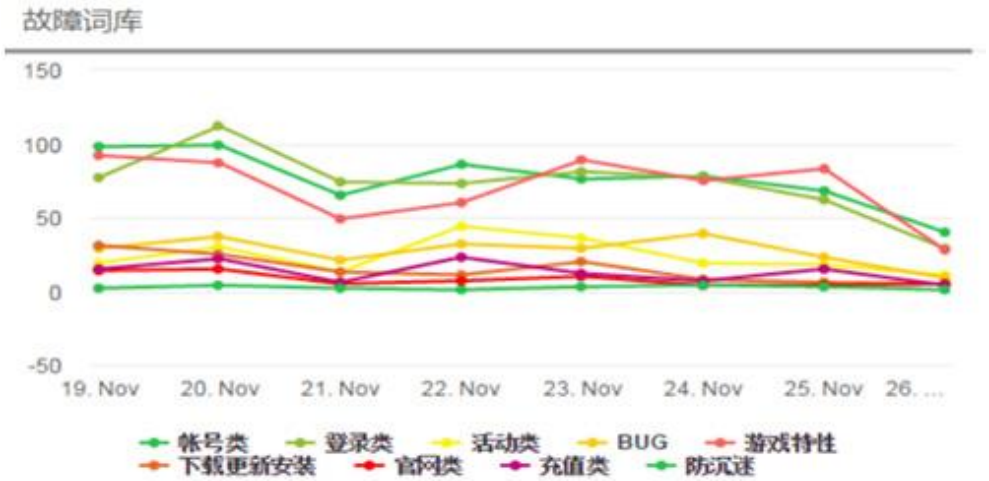
热词分析：

从官网论坛、百度贴吧、应用市场、公众号、微博……等抓取游戏相关热词。



通过这些热词监控，我们对用户当前最关注的热词有一个实时的了解，并进行热词的词义分析，分析这个词的情感，到底是正面，还是中性，还是负面，方便我们快速找到游戏体验的问题。

关键词库：



事先配置好关键词库，然后去抓取这次关键词进行分析。还有，就是对我们客服工单进行实时分析了。

一口气写了这么多，基本上一些手游用户体验和优化方案都已经讲到了，这里特别衷心感谢一起为手游技术不断探索和优化的兄弟姐妹们，感谢你们精益求精的辛苦付出和不断优化！

防劫持解决方案-回源下载

腾讯互娱工程师-赖海军

案例 1：

http 资源劫持：

2017 年 11 月，某游戏发布客户端更新包后，收到部分玩家投诉客户端更新失败，

经过排查发现，玩家网络正常，但是下载到的文件客户端检测异常，怀疑是下载资源文件时遇到了 http 劫持，联系该玩家配合进行测试，访问正常 url 和回源 url 下载到的文件对比发现，正常 url 下载的文件 md5 是错误的，通过回源 url 请求下载的文件 md5 是正确的。判定是遇到了 http 劫持，客户端先暂时通过热更 dir 模块修改资源 url 为回源 url，最终该玩家客户端资源更新正常，成功进入游戏。

案例 2：

302 劫持：

2017 年 5 月，某游戏发布客户端更新包后，收到福建某个玩家投诉客户端更新失败，开发商根据玩家反馈的报错截图发现是某个资源 url 下载异常，运维联系玩家，让玩家通过手机浏览器访问该 url，结果跳转到了错误的 url，可以判定玩家访问 url，302 重定向被运营商指向了错误的 url，客户端先暂时通过热更 dir 模块修改资源 url 为回源 url，最终该玩家客户端资源更新正常，成功进入游戏。

排查过程：

当玩家投诉客户端升级报错后，我们可以按以下步骤进行排查：

玩家客户端资源升级报错-->联系开发定位是哪个 url 下载失败-->让玩家在 (与游戏相同网络下) 手机浏览器访问该 url-->若下载失败，则让玩家用手机浏览器访问回源 url-->由于回源 url 有多个，某个不行可以切换到备用地址，提高玩家的下载成功率--> 确认访问正常后，让客户端临时使用回源 url 给玩家提供资源下载服务。

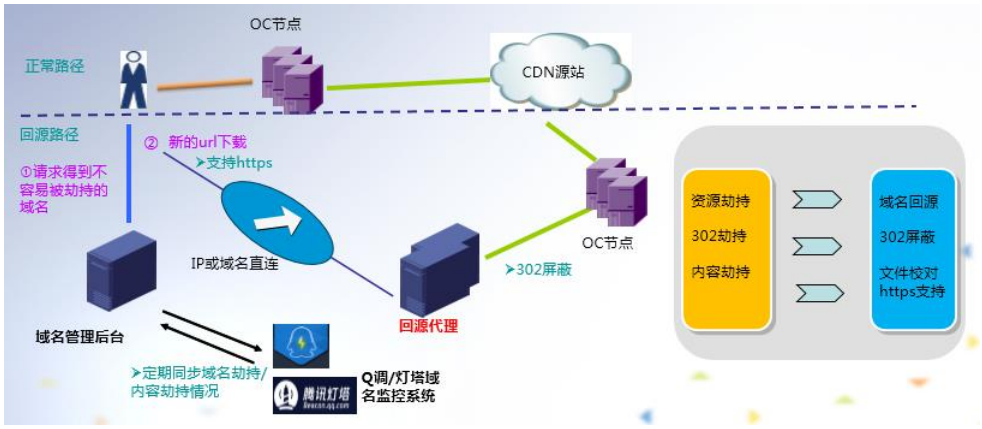
解决方案：

以上案例已在多个游戏发生过，针对 http 资源劫持的现象，我们推出了防劫持的回源代理方案，失败的用户通过直连回源代理服务器来重新获取资源，绕开了

当前异常的 cdn 节点，大幅缓解资源劫持现象。

一、方案介绍：

以下是解决方案的架构图



当用户通过正常路径访问 oc 节点失败后，我们提供了两种接入方式给用户

1.静态方式：

在业务原 url 前面加上 download.mocmna.qq.com，支持 http 和 https

调用举例如下：（若业务 url 为 http://image.xxx.qq.com/test.zip）

http 接入方式：

http://download.mocmna.qq.com/image.xxx.qq.com/test.zip

https 接入方式：

https://download.mocmna.qq.com/image.xxx.qq.com/test.zip

2.动态方式（推荐）：

客户端请求 download01.ino.qq.com/GetUrlHead?client=x.x.x.x ，获取域名或 IP，获取得到域名或 IP 后，再按照【静态方式】操作。

二、方案优势：

1.优质链路切换：根据用户出口 IP，智能选择网络路径，避开下载失败路径

2.全网劫持监测：利用 Q 调和灯塔平台探测技术，监控回源域名或热点 url

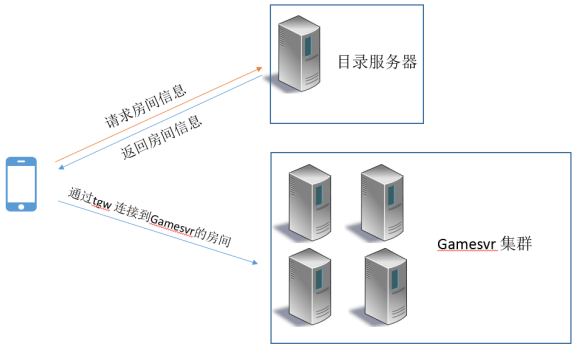
3.数据分析：完善的数据分析报表能力提供单用户的日志定位分析辅助全网监测。

棋牌类业务 Gamesr 实时扩容案例

腾讯互娱高级工程师-罗诚

导语：众所周知游戏业务大都具备有高爆发属性，因此对业务的快速扩容有着较高的要求。动态实时的扩容，不影响正在运行的业务，是当前游戏架构设计的基本条件。在此简单的分享下腾讯欢乐棋牌的业务架构以及动态 Gamesvr 扩展方法，期望能帮助业务同行设计出合理的游戏架构。

业务 Gamesvr 逻辑架构图



架构说明：

1 玩家通过目录服务器拉取到所有的房间信息

2 然后玩家通过房间信息来链接对应的 Gamesvr 进行游戏对局。

3 其中目录服务器拉取到的所有房间信息保证是最新的， **当有 Gamesvr 扩缩容的时候，目录信息会实时更新，便于玩家能获取到最新的房间信息。**

4 扩容方法：直接增加新的 Gamesvr 设备到集群中，然后通过目录服务器中房间信息的的配置和 reload，就可以将流量导入到新的 Gamesvr。以上操作均在线实时完成，对玩家而言无感知。

5 缩容方法：将需要下架的 Gamesvr 中游戏的玩家清理掉（完成当前对局后，踢出这个房间，换到其他可用房间再次游戏，不影响玩家核心体验），然后在目录服务器上将该 Gamesvr 剔除即可完成。

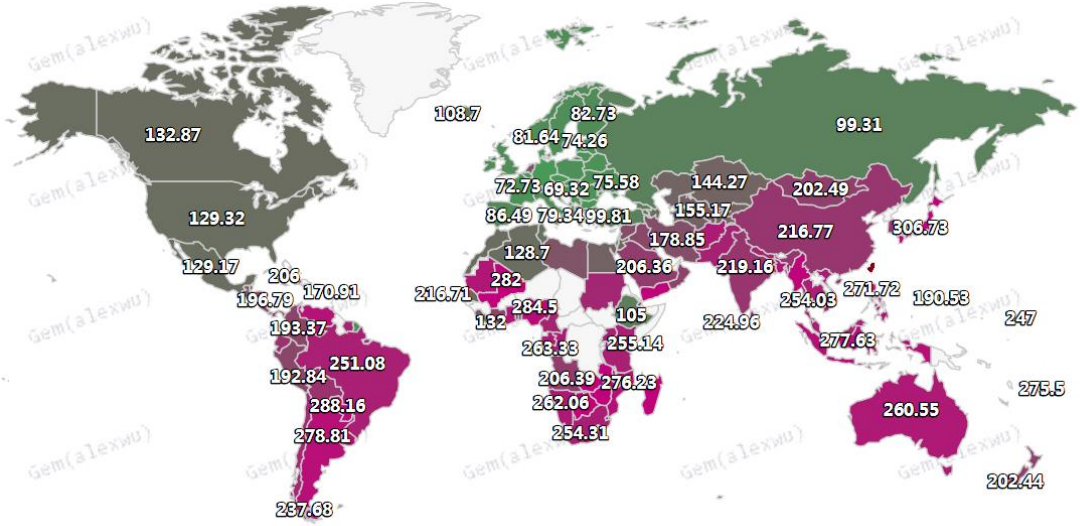
业务系统的架构设计是很重要的环节，在业务上线初期，需要充分的考虑到业务运营期间扩展的便利性，欢乐游戏沿用了 Qgame 的框架，也继承了该框架永不停机的特性，为业务的高速发展提供了良好的基础。

全球同服游戏测速支撑及网络案例分享

腾讯互娱高级工程师-吴检

从 2016 年开始随着手游业务的出海，我们就非常迫切的需要知道海外玩家的游戏体验效果，海外各区域的网络情况。

在海外由于地域的距离、网络架构的复杂，运营商的众多，如果采用单 IDC 部署的话，那么玩家的体验效果并不理想；
如下图某业务的网络平均延迟展示图：



从网络延迟数据看：
1） 南美地区、大洋洲地区基本超过 200ms，体验差；
2） 亚洲地区 100-200ms，体验一般。

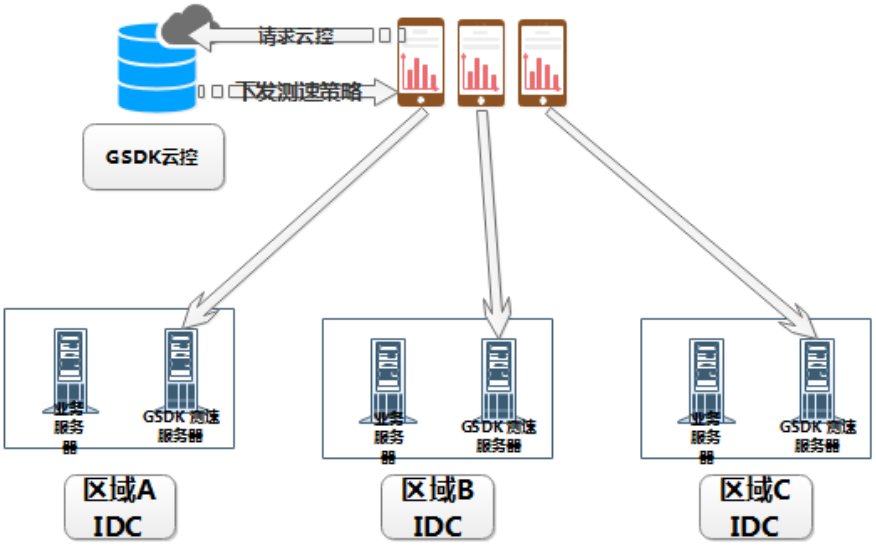
为了玩家能更好的游戏体验，业务逐步优化调整为多区域部署去解决；GSDK 也从原来单区域的网络测速进行优化调整。

优化的措施主要如下：

一、调整多区域测速服务器部署

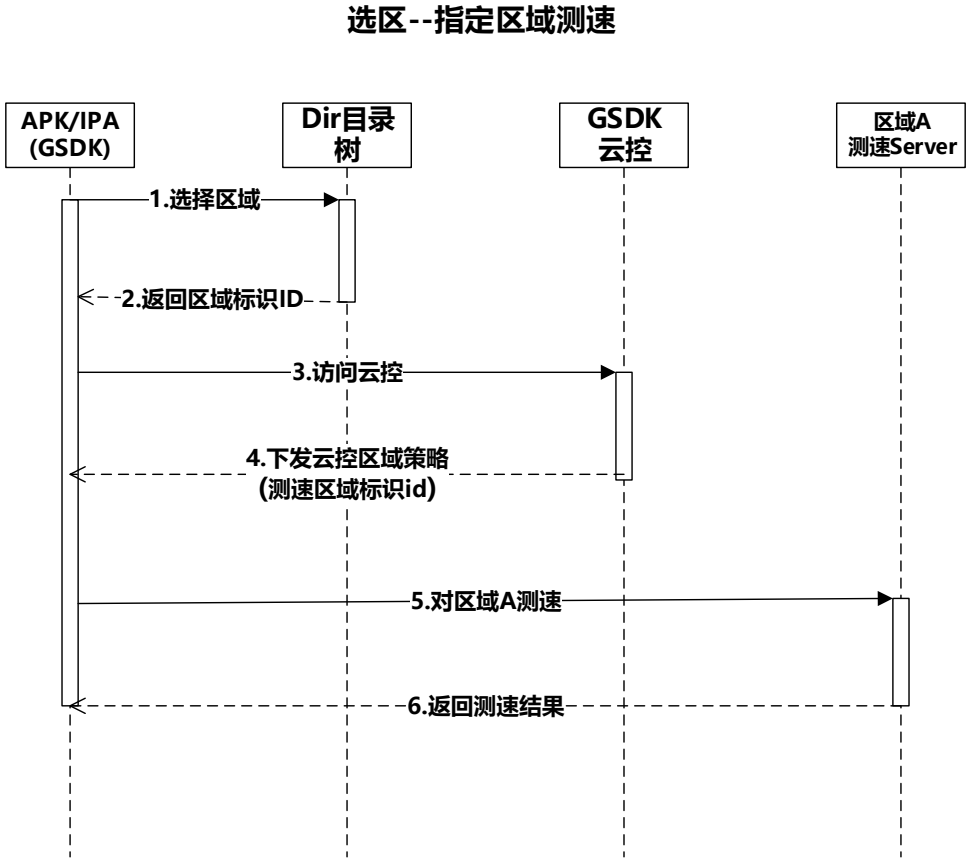
针对多区域部署，在业务部署的多区域 IDC 机房同时也部署测速服务器。

1.1 部署架构图



在业务对战 server 的区域 A、B、C 三个 IDC 机房各部署一台 GSDK 测速服务器。

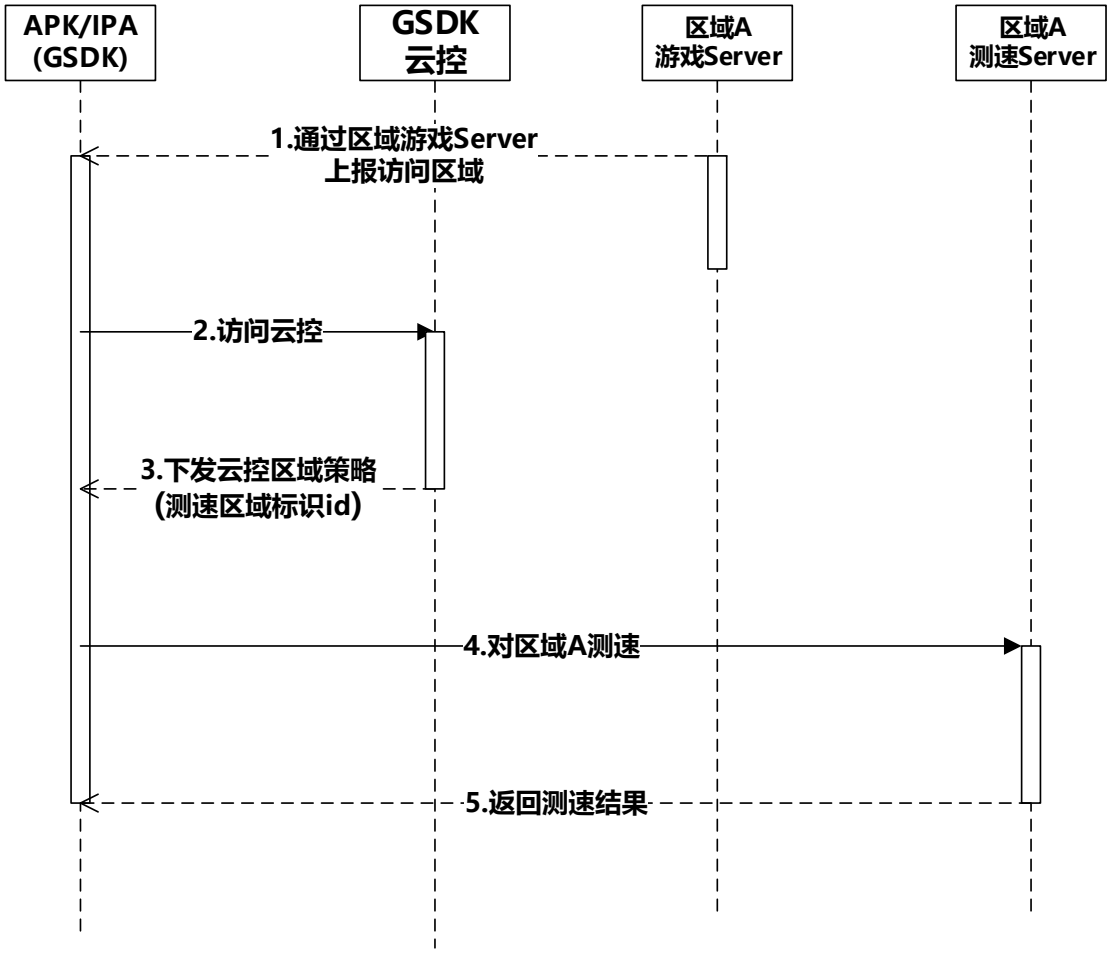
1.2 测速时序图：



通过区域标识 ID 来区分不同区域

APK/IPA 通过访问 GSDK 云控获取下发的策略，访问不同区域的测速 Server 进行测速。

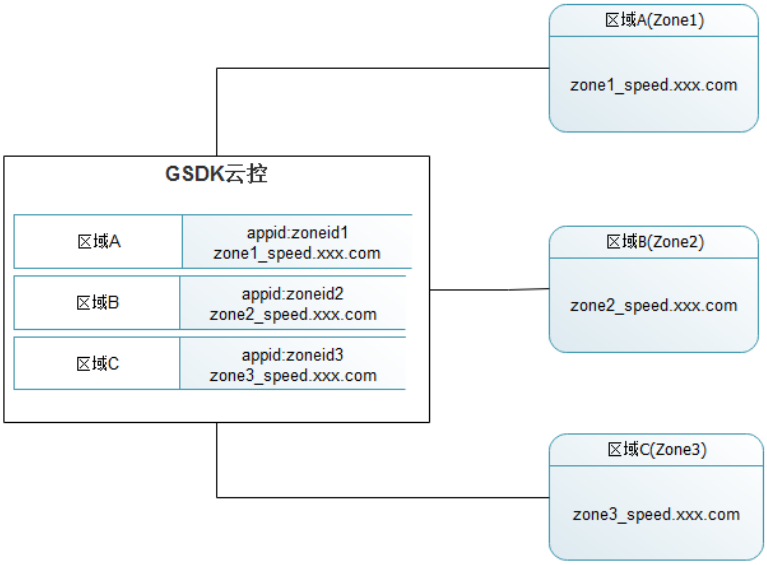
无选区--区域测速



通过区域游戏 Server 来区分不同区域

APK/IPA 通过访问 GSDK 云控获取下发的策略，访问不同区域的测速 Server 进行测速。

二、GSDK 云控调整

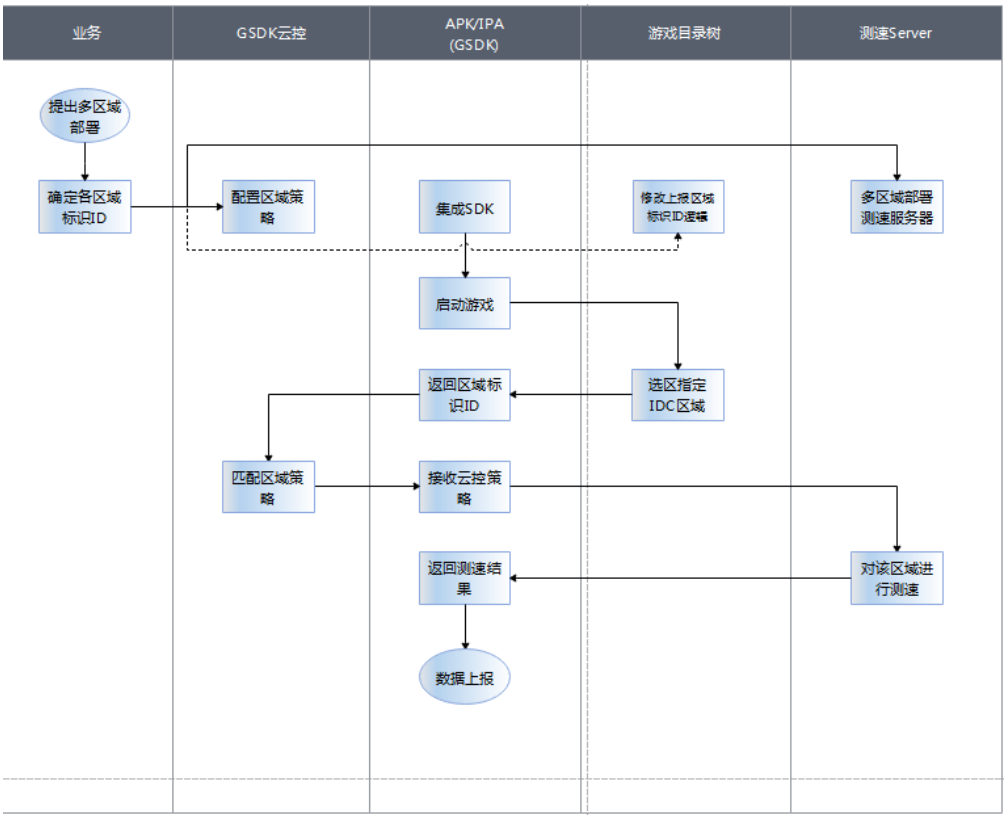


GSDK 云控按照对应的区域配置不同区域的策略。

三、业务配合调整

针对现有两种不同的架构，分别采用了两种不同方案来实现。

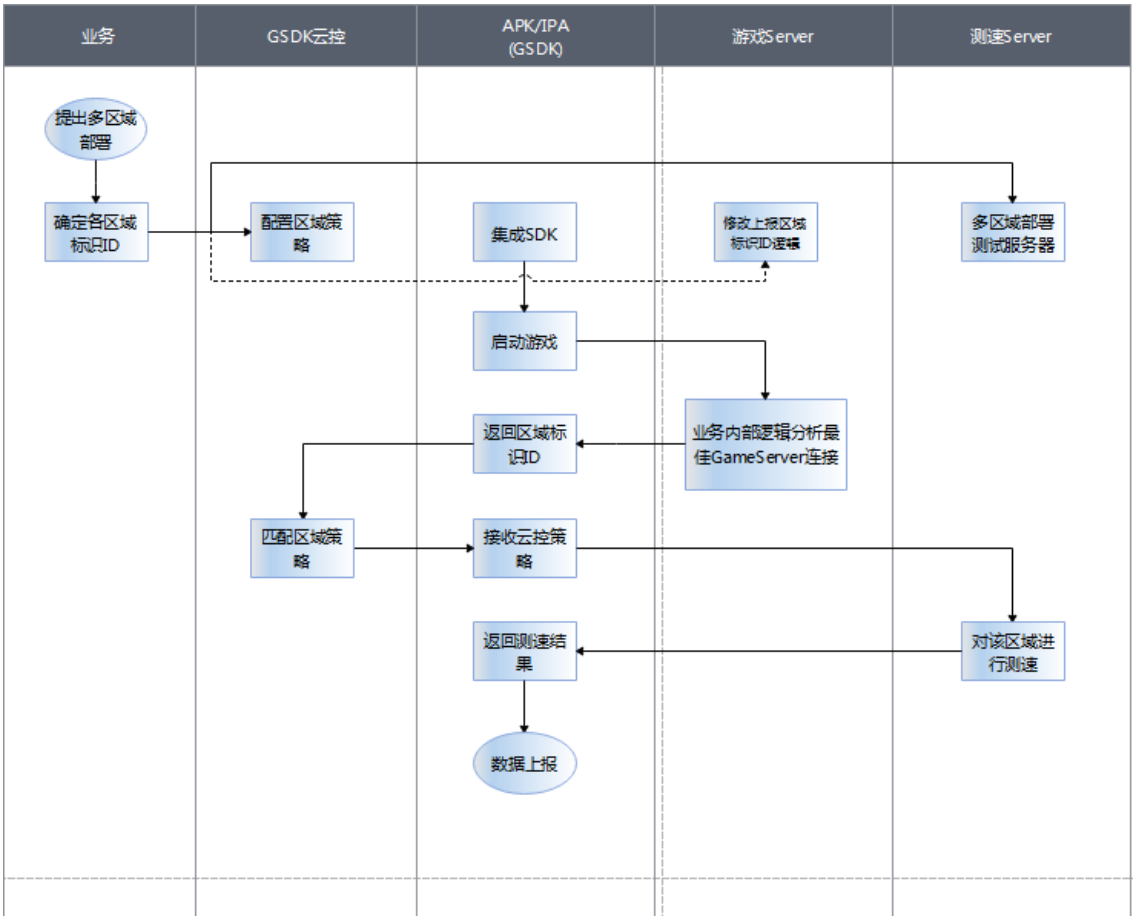
方案一：选区指定区域测速方案，该方案是指 APK/IPA 在登录时可以选择区域。



该方案主要适用于全球多区域部署。

方案二：无选区区域测速方案

该方案是指 APK/IPA 在登录时无选区。



该方案主要适用于国家内多区域部署。

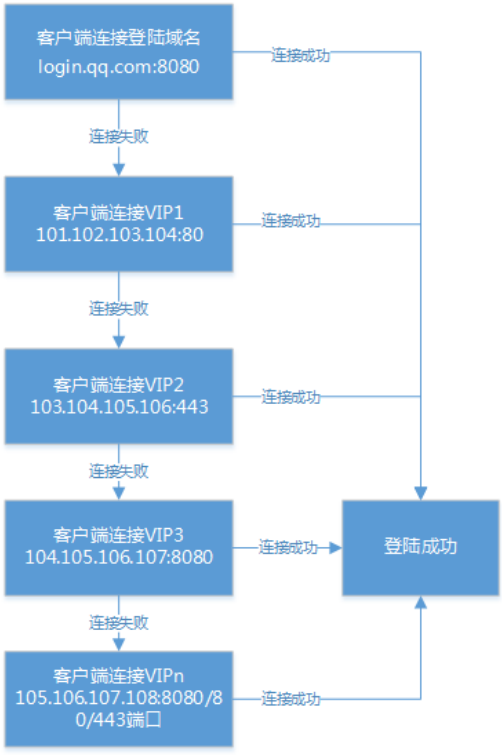
手游防域名劫持和防端口封堵方案

腾讯互娱工程师-邱杰，马帅，张丹，谭兴业

某棋牌游戏经常接到玩家投诉登陆不上游戏，经排查，原因是运营商劫持了游戏登陆域名或封堵了游戏登陆的端口。目前运营商劫持域名和封堵端口的情况十分常见，一旦被封客户端就无法登陆，严重影响业务。业界目前应对方案有 httpdns，但需客户端内嵌 httpdns SDK，且无法解决端口封堵问题。本方案主要对运营商域名劫持和端口封堵在客户端通过技术处理进行规避，能让用户在域名被劫持、端口被封堵的情况下仍可正常登陆业务。使用该方案，客户端无需内嵌任何 SDK，创新点在于：只需在客户端实现 IP+PORT 超时自动轮询逻辑，后台服务监听 3 个端口即可实现风险规避，成本低，收益高，效果好。

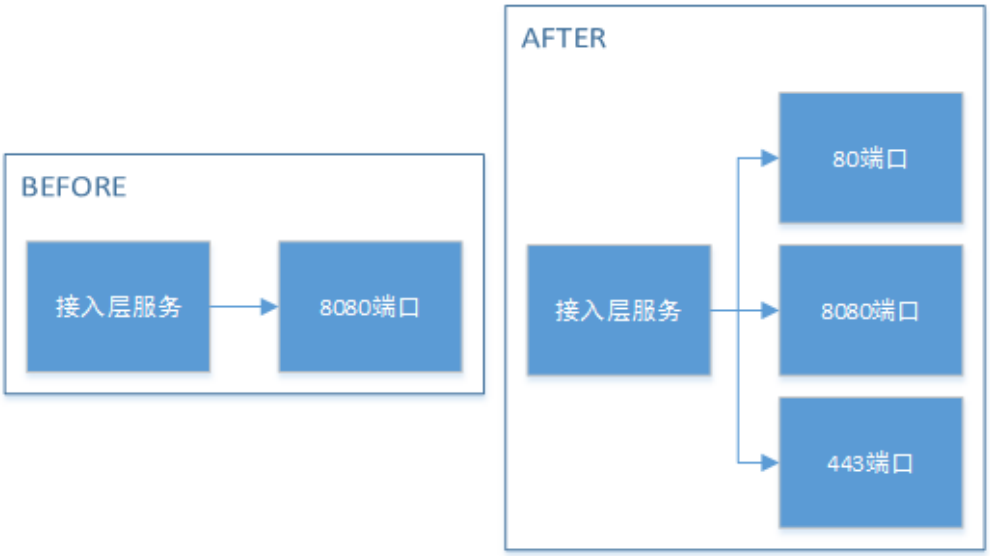
客户端在保存一份域名对应的 VIP 列表，当域名解析超时，则自动轮询连接列表中的 VIP 的 8080、80、443 端口（这三个端口是最常用的，被封堵的可能性最低）。

如：解析域名连接 8080 端口 6s 超时，自动尝试直接连接列表中的第一个 VIP 的 80 端口，若仍然 6 秒超时，则自动尝试连接第二个 VIP 的 443 端口，若仍然 6 秒超时，则自动尝试连接第三个 VIP 的 8080 端口以此类推，这样可以同时规避运营商域名劫持、端口封堵两个问题。



实现方法：

客户端：增加简单的 VIP+VPORT 轮询逻辑。域名解析失败时自动尝试 VIP 直连并轮询 VIP 和端口。
后台服务：由原来监听一个端口改成监听三个端口即可，三个端口实现同样的逻辑，改造成本低。



本方案可以用最低的改造成本和风险，实现防域名劫持、防端口封堵的功能，规避用户因运营商域名劫持、端口封堵造成无法登陆和正常使用 app 的情况。方案上线后用户没有再接到因运营商域名劫持、端口封堵造成的登录失败，该方案已推广到多款业务上使用并已成功申请专利。

手游卡顿用户分析及解决方案

腾讯互娱高级工程师-车国强，宁斌晖

随着 MOBA 游戏受到越来越多用户的喜爱和追捧，影响实时对战类手机游戏的用户体验问题也越来越多，且由于移动网络、用户终端等众多复杂问题叠加，导致用户游戏不流畅、甚至卡顿的原因也非常多元化。本文是从众多处理的用户问题中，提取了几个比较典型的用户案例，展开分析。

实时手游的卡顿、体验不流畅问题似乎比其他类型的游戏更加突出。

大部分实时类手游都采用的是 UDP 协议，游戏的特点是，大量小包、对网络延迟和稳定性要求高，哪怕出现一瞬间的网络延迟或者丢包，用户游戏感受都非常明显，甚至会影响比赛结果。

案例一、信道干扰、信道使用率高

公司内部同事反应，使用公司 wifi 网络玩游戏经常出现卡顿

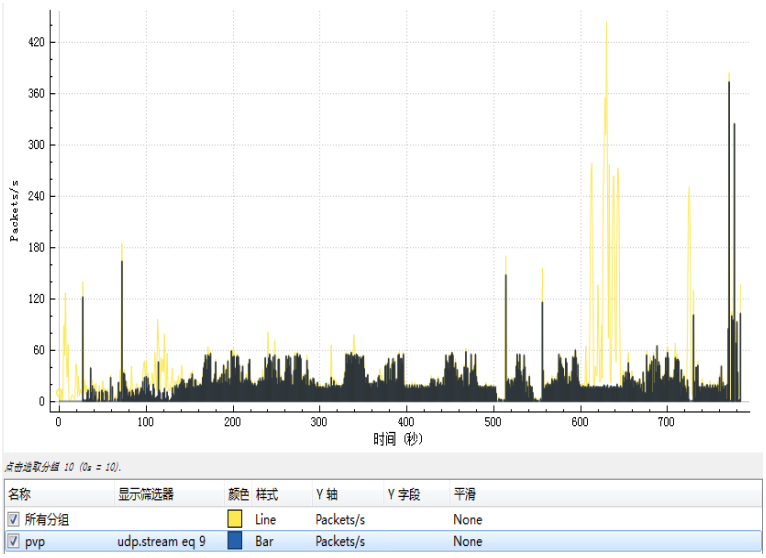
分析及处理：通过联系问题反馈同时，在对应的 wifi 下对网络进行测试，发现网络很不稳定，且有丢包。

通过使用专业的设备进行检测，发现信道使用率非常高，且多 AP 之间的信道干扰比较严重，导致 wifi 不稳定。公司 IT 同事增加接入 AP，分流终端，同时使用容量、功率更佳的路由器，同时定期对路由器做了重启后情况有所改善。

案例二、用户手机后台流量使用较高，导致游戏中出现卡顿

用户反馈玩游戏，经常会出现偶尔卡顿的现象，且 wifi 下和 4G 下都有这种情况。

分析及处理：帮用户抓包分析后发现，在用户游戏时，后台有程序在发送大量的数据包，且从网卡抓包上也看到了除了游戏之外的其他应用数据，推荐用户临时卸载对应的应用后，游戏中出现卡顿的情况减少很多。



后经过我们自己大量测试验证发现，有些应用，在后台运行，也会发送大量的数据包，当用户网卡的数据包量达到一定量级后，就出现了数据发送不稳定，错包率上升等现象，导致游戏数据交互受影响。

针对此类问题，目前在 4G 下和路由器，都有对应的 QOS 机制，保护用户在游戏时，尽量不受干扰。

案例三、澳洲用户玩国服游戏，延迟大，游戏不畅

部分澳大利亚用户反馈，玩国服游戏，延迟较大，游戏体验很差。

分析及处理：澳大利亚到中国物理距离本身就很远，用户从本地访问到国内服务

器的话，延迟稳定性会非常差。经过调研和分析后，在澳洲部署了本地接入点，打通了澳洲接入点到香港的接入点的专线链路，为用户提供骨干网加速。使用加速后，用户的延迟大大降低，保证了海外用户玩国服的游戏体验。

总结：随着腾讯游戏全球影响力的提升，包括海外的国外用户，海外华人等都有访问国服的需求，我们已经在全球重点区域部署了专线链路，通过专线，访问国服，可以大大降低网络的延迟和不稳定。

案例四、用户手机请求被劫持，导致请求跨网

用户反馈近一段时间使用 4G 网络玩游戏、高峰时间段非常卡，切换到 wifi 后正常。

分析与处理：

用户是电信 4G，使用的是安卓手机，帮助用户排查发现，用户向我们游戏服务器域名发起请求时的 clientip 是一个联通 IP，所以域名解析结果给客户返回了一个联通的 serverip，用户使用联通 IP 请求后，就出现了跨运营问题，导致延迟变大，网络不稳定。

电信的手机卡，为什么是联通的 clientip 呢？经过对用户手机进行排查发现，在用户安卓手机设置一栏中，有个手机自带的第三方功能，节省流量，当用户打开这个按钮后，用户的请求被拦截到了第三方的代理服务器上，然后代理服务器作为客户端再向我们的游戏服务器域名发送了请求。但是，第三方的代理服务器是中国联通网络，所以导致跨网。帮助用户关闭此功能后，用户请求的 clientip 恢复为中国电信的 IP 了，解析获得了同网的 serverip 后，游戏网络稳定。

总结：移动网络下，用户的网络环境非常复杂，包括骨干网、空口、设备差异、手机设置、基站容量及覆盖，用户行为等，都会对游戏运行产生影响，腾讯游戏基于海量数据分析，已经提供了一整套移动用户网络诊断、网络加速的解决方案，大大的提升了用户的游戏体验，后续我们也会输出更多经典案例及排查方法。