容器类

软件学院
马玲 张圣林

容器类

Qt库提供了一组通用的基于模板的容器类（container classes）。这些容器类可以用来存储指定类型的项目（items），例如，如果大家需要一个QString类型的可变大小的数组，那么可以使用QVector(QString)。与STL(Standard Template Library，C++的标准模板库)中的容器类相比，Qt中的这些容器类更轻量，更安全，更容易使用。

- Qt的容器类简介
- 遍历容器
- 通用算法
- QString
- QByteArray和QVariant

Qt的容器类简介

QSet<T>	它提供了一个快速查询单值的数学集。
QMap<Key,T>	它提供了一个字典（关联数组），将 Key 类型的键值映射到 T 类型的值上。一般每一个键关联一个单一的值。QMap 使用键顺序来存储它的数据；如果不关心存储顺序，那么可以使用 QHash 来代替它，因为 QHash 更快速。
QMultiMap<Key,T>	它是 QMap 的一个便捷类，提供了实现多值映射的方便的接口函数，例如一个键可以关联多个值。
QHash<Key,T>	它与 QMap 拥有基本相同的接口，但是它的查找速度更快。QHash 的数据是以任意的顺序存储的。
QMultiHash<Key,T>	它是 QHash 的一个便捷类，提供了实现多值散列的方便的接口函数。
	进行了插入或者删除操作，那么这个迭代器就无效了。）
QVector<T>	它在内存的相邻位置存储给定类型的值的一个数组。在 vector 的前面或者中间插入项目是非常缓慢的，因为这样可能导致大量的项目要在内存中移动一个位置。
QStack<T>	它是 QVector 的一个便捷子类，提供了后进先出（LIFO）语义。它添加了 push()，pop()和 top()等函数。
QQueue<T>	它是 QList 的一个便捷子类，提供了先进先出（FIFO）语义。它添加了 enqueue()，dequeue()和 head()等函数。

QList是一个模板类，它提供了一个列表。QList<T>实际上是一个T类型项目的指针数组，所以它支持基于索引的访问，而且当项目的数目小于1000时，可以实现在列表中间进行快速的插入操作。QList提供了很多方便的接口函数来操作列表中的项目，例如：

- 插入操作insert()；
- 替换操作replace()；
- 移除操作removeAt()；
- 移动操作move()；
- 交换操作swap()；
- 在表尾添加项目append()；
- 在表头添加项目prepend()；
- 移除第一个项目removeFirst()；
- 移除最后一个项目removeLast()；
- 从列表中移除一项并获取这个项目takeAt()，还有相应的takeFirst()和takeLast()；
- 获取一个项目的索引indexOf()；
- 判断是否含有相应的项目contains()；
- 获取一个项目出现的次数count()。

对于QList，可以使用“<<”操作符来向列表中插入项目，也可以使用“[ ]”操作符通过索引来访问一个项目，其中项目是从0开始编号的。不过，对于只读的访问，另一种方法是使用at()函数，它比“[ ]”操作符要快很多。

例如：

```
QList<QString> list;
list << "aa" << "bb" << "cc"; // 插入项目
if(list[1] == "bb") list[1] = "ab";
list.replace(2,"bc");        // 将“cc”换为“bc”
qDebug() << "the list is: ";  // 输出整个列表
for(int i=0; i<list.size(); ++i){
   qDebug() << list.at(i);   // 现在列表为aa ab bc
}
list.append("dd");           // 在列表尾部添加
list.prepend("mm");          // 在列表头部添加
QString str = list.takeAt(2); // 从列表中删除第3个项目，并获取它
qDebug() << "at(2) item is: " << str;
qDebug() << "the list is: ";
for(int i=0; i<list.size(); ++i)
{
   qDebug() << list.at(i);   // 现在列表为mm aa bc dd
}
```

QMap类是一个容器类，它提供了一个基于跳跃列表的字典（a skip-list-based dictionary）。QMap<Key,T>是Qt的通用容器类之一，它存储（键，值）对并提供了与键相关的值的快速查找。QMap中提供了很多方便的接口函数，例如：

- 插入操作insert()；
- 获取值value()；
- 是否包含一个键contains()；
- 删除一个键remove()；
- 删除一个键并获取该键对应的值take()；
- 清空操作clear()；
- 插入一键多值insertMulti()。

可以使用“[ ]”操作符插入一个键值对或者获取一个键的值，不过当使用该操作符获取一个不存在的键的值时，会默认向map中插入该键，为了避免这个情况，可以使用value()函数来获取键的值。当使用value()函数时，如果指定的键不存在，那么默认会返回0，可以在使用该函数时提供参数来更改这个默认返回的值。QMap默认是一个键对应一个值的，但是也可以使用insertMulti()进行一键多值的插入，对于一键多值的情况，更方便的是使用QMap的子类QMultiMap。

例如：

```
QMap<QString,int> map;
 map["one"] = 1;        // 向map中插入("one",1)
 map["three"] = 3;
map.insert("seven",7);  // 使用insert()函数进行插入
 // 获取键的值，使用“[ ]”操作符时，如果map中没有该键，那么会自动插入
 int value1 = map["six"];
 qDebug() << "value1:" << value1;
qDebug() << "contains 'six' ?" << map.contains("six");
 // 使用value()函数获取键的值，这样当键不存在时不会自动插入
 int value2 = map.value("five");
 qDebug() << "value2:" << value2;
qDebug() << "contains 'five' ?" << map.contains("five");
 // 当键不存在时，value()默认返回0，这里可以设定该值，比如这里设置为9
 int value3 = map.value("nine",9);
 qDebug() << "value3:" << value3;
```

嵌套和赋值

- 容器也可以嵌套使用，例如QMap<QString,QList<int> >，这里键的类型是QString，而值的类型是QList<int>，需要注意，在后面的“> >”符号之间要有一个空格，不然编译器会将它当做“>>”操作符对待。

- 在各种容器中所存储的值的类型可以是任何的可赋值的数据类型，该类型需要有一个默认的构造函数，一个拷贝构造函数和一个赋值操作运算符，像基本的类型double，指针类型，Qt的数据类型如QString、QDate、QTime等。但是QObject以及QObject的子类都不能存储在容器中，不过，可以存储这些类的指针，例如QList<QWidget*>。

遍历容器

遍历一个容器可以使用迭代器（iterators）来完成，迭代器提供了一个统一的方法来访问容器中的项目。Qt的容器类提供了两种类型的迭代器：

- Java风格迭代器
- STL风格迭代器

如果只是想按顺序遍历一个容器中的项目，那么还可以使用Qt的：

- foreach关键字

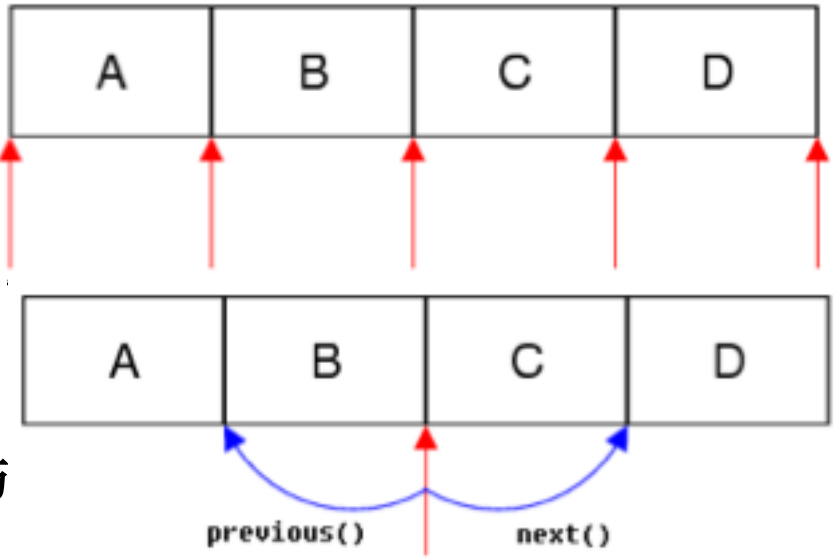
Java风格迭代器

Java风格迭代器在使用时比STL风格迭代器要方便很多，但是在性能上稍微弱于后者。对于每一个容器类，都有两个Java风格迭代器数据类型：一个提供只读访问，一个提供读写访问。

容器	只读迭代器	读写迭代器
QList<T>, QQueue<T>	QListIterator<T>	QMutableListIterator<T>
QLinkedList<T>	QLinkedListIterator<T>	QMutableLinkedListIterator<T>
QVector<T>, QStack<T>	QVectorIterator<T>	QMutableVectorIterator<T>
QSet<T>	QSetIterator<T>	QMutableSetIterator<T>
QMap<Key, T>, QMultiMap<Key, T>	QMapIterator<Key, T>	QMutableMapIterator<Key, T>
QHash<Key, T>, QMultiHash<Key, T>	QHashIterator<Key, T>	QMutableHashIterator<Key, T>

QList示例：

```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QListIterator<QString> i(list); // 创建列
qDebug() << "the forward is :";
while (i.hasNext())          // 正向遍历
    qDebug() << i.next();
qDebug() << "the backward is :";
while (i.hasPrevious())      // 反向遍历
    qDebug() << i.previous();
```



这里先创建了一个QList列表list，然后使用list作为参数创建了一个列表的只读迭代器。这时，迭代器指向列表的第一个项目的前面（这里是指向项目“A”的前面）。然后使用hasNext()函数来检查在该迭代器后面是否还有项目，如果还有项目，那么使用next()来跳过这个项目，next()函数会返回它所跳过的项目。当正向遍历结束后，迭代器会指向列表最后一个项目的后面，这时可以使用hasPrevious()和previous()来进行反向遍历。可以看到，Java风格迭代器是指向项目之间的，而不是直接指向项目。所以，迭代器或者指向容器的最前面，或者指向两个项目之间，或者指向容器的最后面。

QMap示例：

```
QMap<QString, QString> map;
map.insert("Paris", "France");
map.insert("Guatemala City", "Guatemala");
map.insert("Mexico City", "Mexico");
map.insert("Moscow", "Russia");
QMapIterator<QString,QString> i(map);
while(i.hasNext()) {
    i.next();
   qDebug() << i.key() << " : " << i.value();
}
 if(i.findPrevious("Mexico"))
       qDebug() << “find ‘Mexico’”;  // 向前查找键的值
```

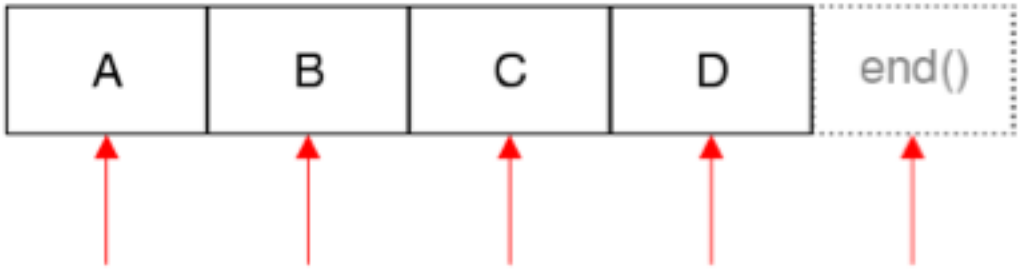
这里在QMap中存储了一些（首都，国家）键值对，然后删除了包含以“City”字符串结尾的键的项目。对于QMap的遍历，可以先使用next()函数，然后再使用key()和value()来获取键和值的信息。

STL风格迭代器

STL风格迭代器兼容Qt和STL的通用算法（generic algorithms），而且在速度上进行了优化。对于每一个容器类，都有两个STL风格迭代器类型：一个提供了只读访问，另一个提供了读写访问。因为只读迭代器比读写迭代器要快很多，所以应尽可能使用只读迭代器。

容器	只读迭代器	读写迭代器
QList<T>, QQueue<T>	QList<T>::const_iterator	QList<T>::iterator
QLinkedList<T>	QLinkedList<T>::const_iterator	QLinkedList<T>::iterator
QVector<T>, QStack<T>	QVector<T>::const_iterator	QVector<T>::iterator
QSet<T>	QSet<T>::const_iterator	QSet<T>::iterator
QMap<Key, T>, QMultiMap<Key, T>	QMap<Key, T>::const_iterator	QMap<Key, T>::iterator
QHash<Key, T>, QMultiHash<Key, T>	QHash<Key, T>::const_iterator	QHash<Key, T>::iterator

QList示例：




```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QList<QString>::iterator i;    // 使用读写迭
qDebug() << "the forward is :";
for (i = list.begin(); i != list.end(); ++i) {
   *i = (*i).toLower();       // 使用QString的toLower()函数转换为小写
   qDebug() << *i;            // 结果为a，b，c，d
}
qDebug() << "the backward is :";
while (i != list.begin()) {
   --i;
   qDebug() << *i;            // 结果为d，c，b，a
}
QList<QString>::const_iterator j; // 使用只读迭代器
qDebug() << "the forward is :";
for (j = list.constBegin(); j != list.constEnd(); ++j)
   qDebug() << *j;            // 结果为a，b，c，d
```

STL风格迭代器的API模仿了数组的指针，例如，使用“++”操作符来向后移动迭代器使其指向下一个项目；使用“*”操作符返回迭代器指向的项目等。需要说明的是，不同于Java风格迭代器，STL风格迭代器是直接指向项目的。其中一个容器的begin()函数返回了一个指向该容器中第一个项目的迭代器，end()函数也返回一个迭代器，但是这个迭代器指向该容器的最后一个项目的下一个假想的虚项目，end()标志着一个无效的位置，当列表为空时，begin()函数等价于end()函数。

在STL风格迭代器中“++”和“--”操作符即可以作为前缀（++i，--i）操作符，也可以作为后缀（i++，i--）操作符。当作为前缀时会先修改迭代器，然后返回修改后的迭代器的一个引用；当作为后缀时，在修改迭代器以前会对其进行复制，然后返回这个复制。如果在表达式中不会对返回值进行处理，那么最好使用前缀操作符（++i，--i），这样会更快一些。对于非const迭代器类型，使用一元操作符“*”获得的返回值可以用在赋值运算符的左侧。STL风格迭代器的常用API如下表所示。

表达式	行为
*i	返回当前项目
++i	前移迭代器到下一个项目
i += n	使迭代器前移 n 个项目
--i	使迭代器往回移动一个项目
i -= n	使迭代器往回移动 n 个项目
i - j	返回迭代器 i 和迭代器 j 之间的项目的数目

QMap示例：

```
QMap<QString, int> map;
map.insert("one",1);
map.insert("two",2);
map.insert("three",3);
QMap<QString, int>::const_iterator p;
qDebug() << "the forward is :";
for (p = map.constBegin(); p != map.constEnd(); ++p)
  qDebug() << p.key() << ":" << p.value();// 结果为(one,1),(three,3),(two,2)
```

这里创建了一个QMap，然后使用STL风格的只读迭代器对其进行了遍历，输出了其中所有项目的键和值 。

Foreach关键字

foreach是Qt向C++语言中添加的一个用来进行容器的顺序遍历的关键字，它使用预处理器来进行实施。 例如：

```
QList<QString> list;
list.insert(0, "A");
list.insert(1, "B");
list.insert(2, "C");
qDebug() <<"the list is :";
foreach (QString str,list) {    // 从list中获取每一项
    qDebug() << str;          // 结果为A，B，C
}
QMap<QString,int> map;
map.insert("first", 1);
map.insert("second", 2);
map.insert("third", 3);
qDebug() << endl << "the map is :";
foreach (QString str, map.keys())   // 从map中获取每一个键
    qDebug() << str << " : " << map.value(str);
// 输出键和对应的值，结果为(first,1),(second,2),(third,3)
```

通用算法

在<QtAlgorithms>头文件中，Qt提供了一些全局的模板函数，这些函数是可以使用在容器上的十分常用的算法。我们可以在任何提供了STL风格迭代器的容器类上使用这些算法，包括QList、QLinkedList、QVector、QMap和QHash。

如果在目标平台上可以使用STL，那么可以使用STL的算法来代替Qt的这些算法，因为STL提供了更多的算法，而Qt只提供了其中最重要的一些算法。

示例：

```
QStringList list;
 list << "one" << "two" << "three";
 qDebug() << QObject::tr("qCopy()算法： ");
QVector<QString> vect(3);
// 将list中所有项目复制到vect中
 std:copy(list.begin(), list.end(), vect.begin());
qDebug() << vect; //结果为one,two,three

qDebug() << endl << QObject::tr("qEqual()算法： ");
// 从list的开始到结束的所有项目与vect的开始及其后面的等数量的项目进行比较，
// 全部相同则返回true
bool ret1 = std:equal(list.begin(), list.end(), vect.begin());
qDebug() << "euqal: " << ret1; //结果为true

qDebug() << endl << QObject::tr("qFind()算法： ");
// 从list中查找"two",返回第一个对应的值的迭代器，如果没有找到则返回end()
QList<QString>::iterator i = std:find(list.begin(), list.end(), "two");
qDebug() << *i;    // 结果为"two"
```

QString

- QString类提供了一个Unicode（Unicode是一种支持大部分文字系统的国际字符编码标准）字符串。其实在第一个Hello World程序就用到了它，而几乎所有的程序中都会使用到它。

- QString存储了一串QChar，而QChar提供了一个16位的Unicode 4.0字符。在后台，QString使用隐式共享（implicit sharing）来减少内存使用和避免不必要的数据拷贝，这也有助于减少存储16位字符的固有开销。

隐式共享

- 隐式共享（Implicit Sharing）又称为写时复制（copy-on-write）。Qt中很多C++类使用隐式数据共享来尽可能的提高资源使用率和尽可能的减少复制操作。使用隐式共享类作为参数传递是既安全又有效的，因为只有一个指向该数据的指针被传递了，只有当函数向它写入时才会复制该数据。

- 共享的好处是程序不需要进行不必要的数据复制，这样可以减少数据的拷贝和使用更少的内存，对象也可以很容易地被分配，或者作为参数被传递，或者从函数被返回。隐式共享在后台进行，在实际编程中我们不必去关注它。Qt中主要的隐式共享类有：QByteArray、QCursor、QFont、QPixmap、QString、QUrl、QVariant和所有的容器类等等 。

例如：

```
QPixmap p1, p2;
p1.load("image.bmp");
p2 = p1;                    // p1与p2共享数据
QPainter paint;
paint.begin(&p2);           // p2被修改
paint.drawText(0,50, "Hi");
paint.end();
```

例如这里QPixmap类是一个隐式共享类，开始时p1和p2的引用计数都为1。

- 深拷贝意味着复制一个对象，而浅拷贝则是复制一个引用（仅仅是一个指向共享数据块的指针）这个共享数据块的引用计数就会增加。
- 当隐式共享类使用“=”操作符时就是使用浅拷贝，如上面的“p2 = p1；”语句。但是当一个对象被修改时，就必须进行一次深拷贝，比如上面程序中“paint.begin(&p2);”语句要对p2进行修改，这时就要对数据进行深拷贝，使p2和p1指向不同的数据结构，然后将p1的引用计数设为1，p2的引用计数也设为1。

编辑操作

在QString中提供了多个方便的函数来操作字符串，例如：

- append()和prepend()分别实现了在字符串后面和前面添加字符串或者字符；replace()替换指定位置的多个字符；
- insert()在指定位置添加字符串或者字符；
- remove()在指定位置移除多个字符；
- trimmed()除去字符串两端的空白字符，这包括‘\t’、‘\n’、‘\v’、‘\f’、‘\r’和‘ ’；
- simplified()不仅除去字符串两端的空白字符，还将字符串中间的空白字符序列替换为一个空格；
- split()可以将一个字符串分割为多个子字符串的列表等等。

对于一个字符串，也可以使用“[ ]”操作符来获取或者修改其中的一个字符，还可以使用“+”操作符来组合两个字符串。在QString类中一个null字符串和一个空字符串并不是完全一样的。一个null字符串是使用QString的默认构造函数或者在构造函数中传递了0来初始化的字符串；而一个空字符串是指大小为0的字符串。一般null字符串都是空字符串，但一个空字符串不一定是一个null字符串，在实际编程中一般使用isEmpty()来判断一个字符串是否为空。

示例：

```
QString str = "hello";
qDebug() << QObject::tr("字符串大小： ") << str.size(); // 大小为5
str[0] = QChar('H');     // 将第一个字符换为‘H'
qDebug() << QObject::tr("第一个字符： ") << str[0]; // 结果为‘H'
str.append(" Qt");       // 向字符串后添加"Qt"
str.replace(1,4,"i");    // 将第1个字符开始的后面4个字符替换为字符串"i"
str.insert(2," my");     // 在第2个字符后插入" my"
qDebug() << QObject::tr("str为： ") << str; // 结果为Hi my Qt
str = str + "!!!";       // 将两个字符串组合
qDebug() << QObject::tr(“str为： ”) << str; // 结果为Hi my Qt！ ！ ！

str = " hi\r\n Qt!\n  ";
qDebug() << QObject::tr("str为： ") << str;
QString str1 = str.trimmed();   // 除去字符串两端的空白字符
qDebug() << QObject::tr("str1为： ") << str1;
QString str2 = str.simplified(); // 除去字符串两端和中间多余的空白字符
qDebug() << QObject::tr("str2为： ") << str2; //结果为hi Qt！
```

查询操作

- 在QString中提供了right()、left()和mid()函数分别来提取一个字符串的最右面，最左面和中间的含有多个字符的子字符串；
- 使用indexOf()函数来获取一个字符或者子字符串在该字符串中的位置；
- 使用at()函数可以获取一个指定位置的字符，它比“[ ]”操作符要快很多，因为它不会引起深拷贝；
- 可以使用contains()函数来判断该字符串是否包含一个指定的字符或者字符串；
- 可以使用count()来获得字符串中一个字符或者子字符串出现的次数；
- 而使用startsWith()和endsWidth()函数可以用来判断该字符串是否是以一个字符或者字符串开始或者结束的；
- 对于两个字符串的比较，可以使用“>”和“<=”等操作符，也可以使用compare()函数。

示例：

```
qDebug() << endl << QObject::tr("以下是在字符串中进行查询的操作： ") <<endl;

str = "yafeilinux";
qDebug() << QObject::tr("字符串为： ") << str;

// 执行下面一行代码后，结果为linux
qDebug() << QObject::tr("包含右侧5个字符的子字符串： ") << str.right(5);

// 执行下面一行代码后，结果为fei
qDebug() << QObject::tr("包含第2个字符以后3个字符的子字符串： ") << str.mid(2,3);

qDebug() << QObject::tr("'fei'的位置： ") <<str.indexOf("fei"); //结果为2

qDebug() << QObject::tr("str的第0个字符： ") << str.at(0); //结果为y

qDebug() << QObject::tr("str中'i'字符的个数： ") << str.count('i'); //结果为2

// 执行下面一行代码后，结果为true
qDebug() << QObject::tr("str是否以”ya“开始？ ") << str.startsWith("ya");
```

转换操作

- QString中的toInt()、toDouble()等函数可以很方便的将字符串转换为整型或者double型数据，当转换成功后，它们的第一个bool型参数会为true；

- 使用静态函数number()可以将数值转换为字符串，这里还可以指定要转换为哪种进制；

- 使用toLower()和toUpper()函数可以分别返回字符串小写和大写形式的副本。

示例：

```
qDebug() << endl << QObject::tr("以下是字符串的转换操作：") << endl;
str = "100";
qDebug() << QObject::tr("字符串转换为整数：") << str.toInt(); // 结果为100
int num = 45;
qDebug() << QObject::tr("整数转换为字符串：") << QString::number(num);//结果为“45”
str = "FF";
bool ok;
int hex = str.toInt(&ok,16);
qDebug() << "ok: "<< ok << QObject::tr("转换为十六进制：") << hex; // 结果为ok：true 255
num = 26;
qDebug() << QObject::tr("使用十六进制将整数转换为字符串：")
         << QString::number(num,16);//结果为1a
str = "123.456";
qDebug() << QObject::tr("字符串转换为浮点型：") << str.toFloat();//结果为123.456
str = "abc";qDebug() << QObject::tr("转换为大写：") << str.toUpper();// 结果为ABC
str = "ABC";qDebug() << QObject::tr("转换为小写：") <<str.toLower();// 结果为abc
```

arg( )函数

示例：

```
int age = 25;
QString name = "yafei";

// name代替%1，age代替%2
str = QString("name is %1, age is %2").arg(name).arg(age);
// 结果为name is yafei,age is 25
qDebug() << QObject::tr("更改后的str为： ") << str;

str = "%1 %2";
qDebug() << str.arg("%1f","hello");     // 结果为%1f hello
qDebug() << str.arg("%1f").arg("hello"); // 结果为hellof %2

str = QString("ni%1").arg("hi",5,'*');
qDebug() << QObject::tr("设置字段宽度为5，使用'*'填充： ") << str;//结果为ni***hi

qreal value = 123.456;
str = QString("number: %1").arg(value,0,'f',2);
qDebug() << QObject::tr("设置小数点位数为两位： ") << str;  //结果为"number:123.45
```

- arg()函数中的参数可以取代字符串中相应的“%1”等标记，在字符串中可以使用的标记在1到99之间，arg()函数会从最小的数字开始对应，比如QString(“%5,%2,%7”).arg(“a”).arg(“b”)，那么“a”会代替“%2”，“b”会代替“%5”，而“%7”会直接显示。
- arg()的一种重载形式是：

  arg (const QString & a1, const QString & a2)，

  它与使用str.arg(a1).arg(a2)是相同的，不过当参数a1中含有“%1”等标记时，两者的效果是不同的，这个可以在前面的程序中看到。
- 该函数的另一种重载形式为：

  arg (const QString & a, int fieldWidth = 0, const QChar & fillChar = QLatin1Char(‘ ’))，

  这里可以设定字段宽度，如果第一个参数a的宽度小于fieldWidth的值，那么就可以使用第三个参数设置的字符来进行填充。这里的fieldWidth如果为正值，那么文本是右对齐的，比如前面程序中的结果为“ni***hi”。而如果为负值，那么文本是左对齐的，例如将前面的程序中的fieldWidth改为-5，那么结果就应该是“nihi***”。
- arg()还有一种重载形式：

  arg (double a, int fieldWidth = 0, char format = ‘g’, int precision = -1,

  const QChar & fillChar = QLatin1Char(‘ ’))，

  它的第一个参数是double类型的，后面的format和precision分别可以指定其类型和精度。

QByteArray

- QByteArray类提供了一个字节数组，它可以用来存储原始字节（包括‘\0’）和传统的以‘\0’结尾的8位字符串。
- 使用QByteArray比使用const char * 要方便很多，在后台，它总是保证数据以一个‘\0’结尾，而且使用隐式共享来减少内存的使用和避免不必要的数据拷贝。
- 但是除了当需要存储原始二进制数据或者对内存保护要求很高（如在嵌入式Linux上）时，一般都推荐使用QString，因为QString是存储16位的Unicode字符，使得在应用程序中更容易存储非ASCII和非Latin-1字符，而且QString全部使用的是Qt的API。
- QByteArray类拥有和QString类相似的接口函数，除了arg()以外，在QByteArray中都有相同的用法。

QVariant

- QVariant类像是最常见的Qt的数据类型的一个共用体（union），一个QVariant对象在一个时间只保存一个单一类型的一个单一的值（有些类型可能是多值的，比如字符串列表）。
- 可以使用toT()（T代表一种数据类型）函数来将QVariant对象转换为T类型，并且获取它的值。这里toT()函数会复制以前的QVariant对象，然后对其进行转换，所以以前的QVariant对象并不会改变。
- QVariant是Qt中一个很重要的类，比如前面讲解属性系统时提到的QObject::property()返回的就是QVariant类型的对象。

示例：

```
QVariant v1(15);
qDebug() << v1.toInt();                 // 结果为15

QVariant v2(12.3);
qDebug() << v2.toFloat();               // 结果为12.3

QVariant v3("nihao");
qDebug() << v3.toString();              // 结果为"nihao“

QColor color = QColor(Qt::red);
QVariant v4 = color;
qDebug() << v4.type();                  // 结果为QVariant::QColor
qDebug() << v4.value<QColor>();         // 结果为QColor(ARGB 1,1,0,0)

QString str = "hello";
QVariant v5 = str;
qDebug() << v5.canConvert(QVariant::Int);  // 结果为true
qDebug() << v5.toString();              // 结果为"hello"
qDebug() << v5.convert(QVariant::Int);     // 结果为false
qDebug() << v5.toString();              // 转换失败，v5被清空，结果为"0"
```

- QVariant类的toInt()函数返回int类型的值，toFloat()函数返回float类型的值。
- 因为QVariant是QtCore库的一部分，所以它没有提供对QtGui中定义的数据类型（例如QColor、QImage等）进行转换的函数，也就是说，这里没有toColor()这样的函数。不过，我们可以使用QVariant::value()函数或者qVariantValue()模板函数来完成这样的转换，例如上面程序中对QColor类型的转换。
- 对于一个类型是否可以转换为一个特殊的类型，可以使用canConvert()函数来判断，如果可以转换，则该函数返回true。
- 也可以使用convert()函数来将一个类型转换为其他不同的类型，如果转换成功则返回true，如果无法进行转换，variant对象将会被清空，并且返回false。
- 需要说明，对于同一种转换，canConvert()和convert()函数并不一定返回同样的结果，这通常是因为canConvert()只报告QVariant进行两个类型之间转换的能力。也就是说，如果在提供了合适的数据时，这两个类型间可以进行转换，但是，如果提供的数据不合适，那么转换就会失败，这样convert()的返回值就与canConvert()不同了。例如上面程序中的QString类型的字符串str，当str中只有数字字符时，它可以转换为int类型，比如str = “123”，因为它有这个能力，所以canConvert()返回为true。但是，现在str中包含了非数字字符，真正进行转换时会失败，所以convert()返回为false。

小结

本章中学习了容器类及其相关的QString，QByteArray和QVariant等类。这些知识理论性比较强，都是非常重要的内容，要在平时编程的过程中多使用相关知识点，逐渐掌握。